ML-enabled Service Discovery for Microservice Architecture: A QoS Approach

Karthik Vaidhyanathan
International Institute of Information Technology
Hyderabad, India
karthik.vaidhyanathan@iiit.ac.in

Mauro Caporuscio
Linnaeus University
Växjö, Sweden
mauro.caporuscio@lnu.se

Stefano Florio
University of L'Aquila
L'Aquila, Italy
stefa.florio@gmail.com

Henry Muccini
University of L'Aquila
L'Aquila, Italy
henry.muccini@univaq.it

ABSTRACT

Microservice architectures have gained enormous popularity due to their ability to be dynamically added/removed, replicated, and updated according to run-time needs. However, the dynamic nature of microservices introduces uncertainty, which in turn can affect the provided Quality of Service (QoS). This calls for novel service discovery mechanisms able to adapt to the variability of the QoS attributes and further perform effective service discovery and selection. To this end, this paper combines machine learning and self-adaptation techniques to perform service discovery and selection by trading off different QoS attributes. The results of our validation on a state-of-the-art microservices exemplar show that our ML-enabled approach can perform service discovery with 35% higher effectiveness with respect to existing baselines.

CCS CONCEPTS

• **Software and its engineering** → **Software architectures**; • **Computing methodologies** → **Machine learning approaches**;

KEYWORDS

Self-adaptation, Machine Learning, Service Discovery

ACM Reference Format:
Karthik Vaidhyanathan, Mauro Caporuscio, Stefano Florio, and Henry Muccini. 2024. ML-enabled Service Discovery for Microservice Architecture: A QoS Approach. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3605098.3635942

1 INTRODUCTION

In Microservice Architecture, the set of available microservice instances rapidly changes over time due to several reasons, e.g., auto-scaling, failures, and updates [17]. These factors make the *Service Discovery* mechanism a pivotal element of the microservice architecture. Indeed, *Service Discovery* is in charge of detecting and tracing microservice instances and selecting the ones needed for processing each request [21].

Uncertainties and high variability of the execution environment (such as the location of the host machine, time of the day, available resources, failures, updates, etc.) make the service discovery task challenging. Furthermore, as the achievement of multiple QoS is becoming essential in large-scale microservice applications, the service discovery should be able to select instances by also considering the trade-offs between different QoS attributes, which are transient and continuously change over time. This calls for novel service discovery mechanisms that are able to adapt to the variability of the execution environment and perform effective QoS-aware service discovery and selection.

To this end, this paper presents a *ML-enabled Service Discovery* mechanism, which extends the well-known server-side service discovery pattern [21] with machine learning capabilities. Specifically, the proposed approach exploits the MAPE-K model [16] and machine learning techniques such as *deep neural networks* [14] and *reinforcement learning* [23] to dynamically discover and select microservice instances according to multiple QoS to be achieved.

The *contribution* of this work is as follows: (*i*) it proposes a machine learning approach to select the most appropriate microservices under uncertainty; (*ii*) it presents a novel approach for achieving microservice discovery according to multiple QoS; (*iii*) it defines a microservices architecture framework that integrates multiple machine learning and deep learning mechanisms in service discovery and selection; (*iv*) it instantiates the proposed approach to trade off *response time* and *energy consumption* attributes.

The overall effectiveness and efficiency of the proposed approach have been evaluated on *SockShop*, a state-of-the-art microservices exemplar application. The results of the experimentation show that our ML-enabled approach can perform service discovery with 35% higher effectiveness with respect to existing baselines.

In the rest of the paper, Section 2 presents the *SockShop* running example focusing on the scenario of interest. Section 3 introduces our *ML-enabled Service Discovery* mechanism and its architecture, and Section 4 goes into detail on the learning mechanisms. Section 5 evaluates the effectiveness and efficiency of the proposed approach, whereas Section 6 discusses some threats to validity. Finally, Section 7 presents related work, and Section 8 concludes the paper and presents hints for future work.

SAC '24, April 8–12, 2024, Avila, Spain

ACM ISBN 979-8-4007-0243-3/24/04.
https://doi.org/10.1145/3605098.3635942

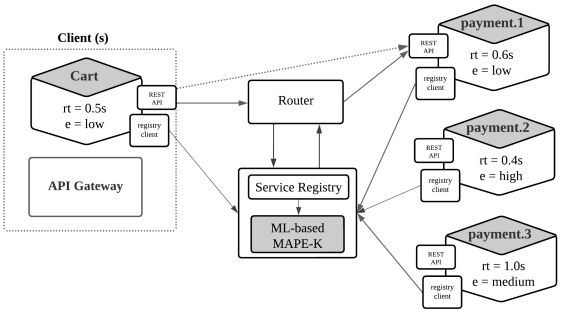

Figure 1: Microservices interaction in SockShop

2 RUNNING EXAMPLE

We will demonstrate our method using SockShop, a state-of-the-art microservices exemplar application. SockShop is an open-source demo e-commerce system composed of microservices[1]. It is designed to showcase and test microservice and cloud-native technologies. SockShop comprises six microservices built using various technology stacks such as Python, Java, NodeJS, and Go.

Figure 1 shows a scenario where one microservice (*Cart*) wants to communicate with a *Payment* microservice which is replicated into three instances. To this end, the *Router* sends a service query to the *Service Discovery*, which then returns the list of available instances of *Payment*. Traditionally, the *Router* then selects one of the instances from the list either using a round-robin mechanism (to balance load) or selects the first instance in the list. However, these approaches do not guarantee an effective discovery with respect to the QoS of the selected instances. In this scenario, we take into consideration two QoS parameters, namely response time (r) and energy consumption (e).

As depicted in Figure 1, each of the instances of the *Payment* microservice offers different values for energy and response time. However, these values do not remain static due to various uncertainties. Moreover, the response time will be calculated based on the response time offered by the microservices for a previous query, and this may change by the time a selection is made (due to, e.g., the location of the instance, RAM, etc.). These challenges result in two key requirements for service discovery:

(1) The service discovery shall adapt to the dynamic changes in the instances' QoS when making the selection.
(2) The service discovery shall select an instance by considering the trade-offs between different QoS parameters.

Hence, given: (i) response time constraints R_{max} and R_{min}, i.e., the maximum and minimum response time values acceptable for a given selection of an instance for a query q, and (ii) E_{max} being the maximum energy that can be consumed for the query q, the goal of service discovery mechanism is maximizing the following utility function that captures the energy and response time requirements of our approach:

$$U_q = w_e \cdot E_q + w_r \cdot R_q, \text{ with}$$

[1]https://microservices-demo.github.io/

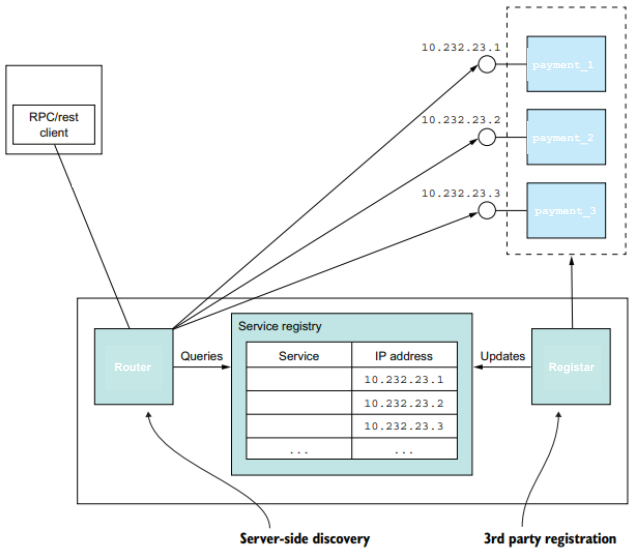

Figure 2: Platform-provided Service Discovery

$$R_q = \begin{cases} (R_{max} - r_q) \cdot p_{rv} & \text{if } r_q \geq R_{max} \\ r_q - R_{min} & \text{if } R_{max} > r_q > R_{min} \\ (r_q - R_{min}) \cdot p_{rv} & \text{if } r_q \leq R_{min} \end{cases}$$

$$E_q = \begin{cases} E_{max} - e_q & \text{if } e_q < E_{max} \\ (E_{max} - e_q) \cdot p_{ev} & \text{otherwise} \end{cases}$$

where, e_q, r_q represent the total energy and response time incurred by the selection for a given service query, q, and w_e, $w_r \in \mathbb{R}^+$ are weights that capture the priority of energy and response time savings, respectively. R_q and E_q are piece-wise functions that capture the response time and energy savings, respectively, where $p_{ev}, p_{rv} \in \mathbb{R}^+$ represent penalties for the violations of energy and response time thresholds, respectively.

3 SYSTEM ARCHITECTURE

Many microservice frameworks (e.g., Spring Boot) have a built-in *Service Discovery* mechanism, implemented as a combination of the *3rd party registration pattern* and the *server-side discovery pattern* [21], including a *Service Registry*, *Registrar*, and a *Router* component (see Figure 2). The *Service Registry* stores all microservices instances across the ecosystem [21]. The *Registrar* continuously monitors the deployed instances and updates the *Service Registry* whenever instances are started and stopped. When a service consumer (either an end-user or another microservice) invokes a given service (e.g., *payment* in Figure 2), the *Router* queries the *Service Registry* to obtain a list of available service instances and then routes requests to one of them (e.g., `payment_1`). Even though the *platform-provided service discovery* is effective and efficient, it does not consider QoS attributes, which are transient and continuously change over time, while selecting the microservices endpoints.

To this end, *ML-enabled Service Discovery* extends the platform-provided service discovery by including three additional components that implement MAPE-K model [16] (see gray elements in Figure 3). Indeed, the *ML-enabled Service Discovery* is composed

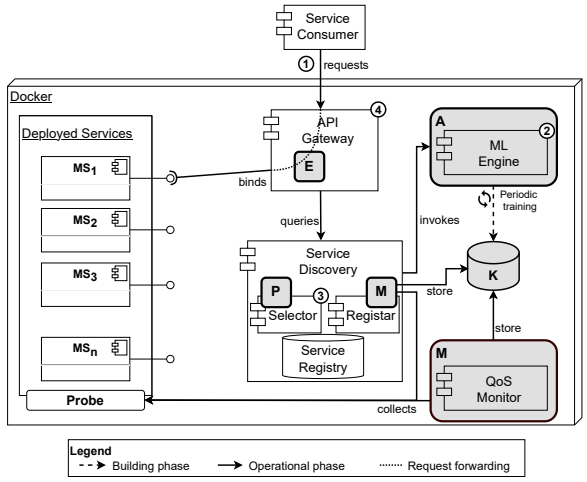

Figure 3: ML-enabled Service Discovery

of four components, namely *API Gateway*, *Service Discovery*, *QoS Monitor*, and *Machine Learning Engine*, which provide the best microservice instance for serving a given request.

The *ML-enabled Service Discovery* architecture shown in Figure 3 is built on top of Docker [20], an open-source platform for building, deploying, and managing containerized applications. It is worth noticing that due to this design choice, the *ML-enabled Service Discovery* mechanism is applicable only to microservices applications fully deployed on Docker.

API Gateway extends the role of *Router* (see Figure 2) by providing complex functionality such as authentication, service orchestration, data aggregation, etc. Indeed, all the requests issued by *Service Consumers* (see ① in Figure 3) are managed by *API Gateway*, which is in charge of forwarding the requests to the proper microservice instances and returning the results to *Service Consumers*. When *API Gateway* receives a request, it retrieves the endpoint of a proper instance by querying the *Service Discovery*.

Service Discovery is implemented according to the 3rd party registration pattern [21] and is responsible for keeping real-time information about the deployed microservices. The role of *Service Discovery* is to return a valid instance for a given microservice, consulting *Service Registry*. That is, let $M = \{m_1, \ldots, m_k\}$ be the set of microservices m registered in *Service Registry*, and let $I^M = \{i_1, \ldots, i_n\}$ be the set of all instances i deployed in the microservice framework. Given a microservice $m \in M$, the service discovery will retrieve the set $I^m \subset I^M = \{i^m\}$ containing all the available instances of m. Such a list is then forwarded to the *Machine Learning Engine* (②), which is in charge of analyzing and ranking the instances in I^m. The ranked list is processed by the *Service Selector* (③), which selects the best instance $\widehat{i^m}$ to be bound by the *API gateway* (④) for processing the consumer request.

In order to realize such behavior, the *ML-enabled Service Discovery* implements the MAPE-K activities (see Figure 3) by: *Monitoring* (*M*) the deployed microservices and their QoS, analyzing (*A*) the monitored data, planning (*P*) the needed binding, and executing (*E*) the planned binding. Further, knowledge (*K*) is used to store collected data.

Monitoring – The monitoring activity is twofold and aims at updating both the *Service Registry* and the knowledge *K*. The data is gathered via Docker's API, which acts as a probe. *QoS Monitor* collects data about the *Quality of Service* attributes of the deployed microservice instances, for example, *response time* and *energy consumption*. The *Response time* of microservices deployed within Docker can be detected by monitoring Docker's networks. Indeed, Docker's network is continuously inspected by *QoS Monitor*, which analyses all the requests, filters out the data, and computes different metrics. Further, each deployed microservice is associated with the container *energy consumption* at any given time t. The data of interest is then saved in the knowledge *K*.

Analyze – The *Machine Learning Engine* (MLE) provides a set of functionalities to forecast the QoS attributes of the microservices instances $i \in I^M$ and rank them accordingly. To this end, *MLE* exploits LSTM models (see Section 4 for further details). In particular, each microservice of the involved application is associated with one LSTM model for the *Response Time* forecasts. Each model is trained to predict the expected QoS attribute (i.e., response time) of each microservice instance in the next minute, given the monitored values in the past ten minutes.

Plan – The planning problem is modeled as an infinite horizon Markov Decision Process (MDP) [5]. In particular, each microservice $m \in M$ is represented by a different MDP. The MDP state s is modeled as a triple $s = (c^{Res}, c^E, \widehat{i^m})$, where $\widehat{i^m} \in I^m$ is the currently bound instance (the instance that is being used) among all the possible instances for m, and $c^{Res} \in C^{Res}$ and $c^E \in C^E$ are the QoS of $\widehat{i^m}$. It considers multiple QoS attributes (i.e., expected response time and instantaneous energy) to perform the selection. Therefore, the set of possible actions is $A^m = I^m = \{i^m_1, \ldots, i^m_k \ldots i^m_n\}$ where i^m_k is the action of selecting the kth instance of the microservice m. The selection is performed by the *Selector* component, which implements a Q-Learning method (see Section 4 for further details).

Execute – Once the instance has been selected, the endpoint is returned to the *API Gateway*, which executes the binding and redirects the client request to the selected instance. The performed request is captured by the monitoring component, which provides the *effective* quality of service parameters to the rewarding phase of the Q-learning algorithm.

4 LEARNING MECHANISMS

The core of our *ML-enabled Service Discovery* mechanism is continuous learning and prediction through deep neural networks and further continuous improvement of service selection through reinforcement learning. This process is implemented through two main components: (*i*) the *Machine Learning Engine* (MLE) and (*ii*) the *Service Selector* (SS). MLE and SS further consist of 6 and 4 components, respectively, as depicted in Figure 4.

MLE consists of six components (see Figure 4). Together, they are responsible for building the ML models as well as for using the machine learning (ML) models to forecast the expected QoS attributes. To this end, MLE makes use of a two-phase approach, namely, batch and real-time. As the name suggests, the batch phase is executed periodically as a batch process, and it is responsible for leveraging the historical QoS data available in the knowledge base (*K*) to train and build ML models (see *Data Processor*, *Model Builder*,

and *Model Evaluator*). The real-time phase on the other hand denotes the inference process where the model is used in real-time to predict the expected QoS attributes for each of the microservice instances (see *Real-time QoS*, *Predictor*, and *Rank Generator*).

Data Processor converts the raw historical QoS data obtained from Knowledge into the format as required by the *Model Builder* component. The raw data stored in Knowledge consists of the data of different QoS parameters (e.g., response time, utilization, throughput, etc.) of the instances of different microservices (collected by the monitor activity). This data has a temporal nature and hence can be converted into a time-series dataset [10]. Further, the problem of predicting the expected QoS can be converted into a time-series forecasting problem.

Let us assume that there are I^m instances for each of the microservices $m \in M$ and that these services were executed for a time t. Then, each of the $i^m \in I^m$ instances will generate an associated d dimensional QoS data, $Q_m \subset Q$ describing the QoS of each of the instances $i^m \in I^m$ for a service m. To this end, the data processor performs four operations, namely: (*i*) Time series Modeling, which involves converting the raw QoS data into a discrete time-series dataset of QoS values with equal intervals of time; (*ii*) Normalization ensures that there is uniformity in the scale of the data, as this forms a crucial step in any ML pipeline; (*iii*) Conversion to a supervised learning dataset, which transforms the normalized data into a primitive supervised learning problem, and (*iv*) Training and test data generation, which is responsible for dividing the data into *training* and *testing sets* for cross-validation and the evaluation of the accuracy of the generated model.

Let us assume that a given microservice m has $J = |I^m|$ instances and has been executed for N units of time. Then, the observation of a QoS parameter p at any instant of time t can be represented by a 2D Vector, $O_p \in R^{J \times N}$ where R denotes the domain of the observed features. The process of temporal aggregation results in the formation of a sequence of the form $o_1, o_2, o_3 \ldots o_t$. The problem of forecasting, O, is then reduced to predicting the most likely h-length sequence in the future given the previous l observations, which include the current one: $o_{t+1}, \ldots, o_{t+h} = \underset{o_{t+1},\ldots,o_{t+h}}{argmax}\ P(o_{t+1}, \ldots, o_{t+h} | o_{t-l+1}, \ldots, o_t)$ where P denotes the probability. Each column in q represents a feature vector, f. These features are mapped to a scale $[0, 1]$ through the process of feature scaling. Further, the data is split in the ratio 7:3, where 70% of the data forms the training set and 30% the testing set. The training dataset of QoS values is passed to the *Model Builder*, and the testing set is sent to the Model Evaluator.

Model Builder forms the key component of the MLE as it is responsible for building the forecasting model. It makes use of Long Short Term Memory (LSTM) Networks [14], a class of Recurrent Neural Networks (RNN) [12] for building the QoS forecast models. LSTMs have shown to be very effective in time-series forecasting [22] as they have the ability to handle the problem of long-term dependency better known in the literature as the Vanishing Gradient Problem [18], as compared to traditional RNN's. Here, the forecast needs to be done for each QoS parameter p of each instance $i^m \in I^m$ for every microservice m. To this end, the *Model Builder* makes use of the dataset o_{p^m} generated by the *Data processor*, which denotes the dataset which consists of the QoS data for a given parameter p for all the instances of a microservice m. This is then used to build a

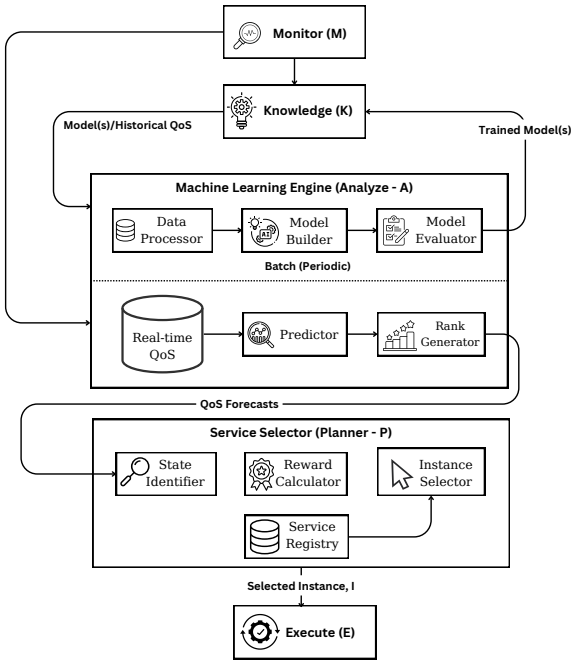

Figure 4: View of the Learning and Selection Process

collection of models, $\mathcal{M} = \{\mu_1, \mu_2, \ldots, \mu_n\}$ where n represents the number of QoS parameters.

Each model is trained to predict the expected QoS values of each microservice instance for a horizon H, given the previous l observations. Therefore, for creating a trained model predicting a QoS parameter o for a given microservice m, each model receives an input (in the form of a matrix) of shape $(|O|, |I^m|)$ where $O = \{o_1, o_2, \ldots, o_n\}$ is the set of previous observations of the given QoS parameter p, and I^m is the set of available instances for a microservice m. A value $v^m_{k,i} = (l_k, p_k)$ is the average monitored QoS value of a parameter p in the time interval k of the service instance i of a microservice m. This data is then trained on an LSTM network to build a trained model μ such that given an observation of o values of p, the output is a list of values $\{o_1, o_2, \ldots, o_h\}$ that represents the average predicted values of p for a horizon H.

Model Evaluator evaluates the set of models built by the *Model Builder* to verify the accuracy of each model prior to deployment. It performs this using the test data set obtained from the Data processor to forecast the QoS values and compare them with the actual values. In the case of lower accuracy, the model evaluator retrains the LSTM network by tuning the network parameters, such as modifying the number of hidden layers, training epochs, etc, supplemented by manual inputs. The final set of trained LSTM models is ingested to the *Knowledge*. Once the models are built, they are used for making predictions of the expected QoS in real time. This is accomplished by the three components of the MLE in the real-time phase (explained below).

Real-time QoS is responsible for collecting real-time QoS data from Knowledge and transforms the real-time QoS data by aggregating them such that there are $|O|$ data points of QoS parameter and

further scaling the data to the form required by the forecasting model(s). This transformed data is sent to the predictor.

Predictor makes use of the trained models available from the Knowledge to forecast the QoS values for the next n intervals as defined by H for each instance for each microservice m. This results in a forecast vector $QF_m = \{QF_{p1}, QF_{p2} \dots QF_{pn}\}$ for each service $m \in M$ representing the QoS forecasts for each parameter p_i. These are forwarded to the *Rank Generator*.

Rank Generator is a lightweight component responsible for ordering the instances according to their QoS values. For a given QoS parameter, p, the generated rank list will be, $QL_p = [i_1, i_2, \dots i_n]$ such that $p_{i_u} < p_{i_w}, \forall u < w$. This is further used by the *Service Selector* for selecting an optimal instance.

As introduced in Section 3, *Service Selector* performs the Plan activity. *Service Selector* directly interacts with the *Service Registry*, which keeps track of the list of instances that are alive for each microservice. The selection problem is modeled as an infinite horizon Markov Decision Process for each microservice $m \in M$. Indeed, each microservice m is associated with an agent, represented by the *Service Selector*, which continuously selects instances $\widehat{i^m}$ by leveraging Q-learning [23]. To this end, the *Service Selector* makes use of three components, namely *State Identifier*, *Rewards Calculator*, and *Instance Selector*.

State Identifier assumes a discrete set of QoS categories for each QoS parameter (e.g., response time, utilization, etc.). The first step of the selection is identifying the expected QoS state of the system EQ, i.e., the set of expected QoS categories for every QoS parameter. This is achieved by mapping QF to a set of categories C^P that identifies the QoS state of the given microservice m until the next service query. For a given QoS parameter p, the mapping between QF and C^p is obtained based on the values in QF and how they meet the thresholds stated in adaptation goals. For a set of QoS parameters, $p1$ and $p2$, the MDP state w is modeled as a triple $w = (c^{p_1}, c^{p_2}, \widehat{i^m})$, where $\widehat{i^m} \in I^m$ is the instance selected, $c^{p_1} \in C^{p_1}$ and $c^{p_2} \in C^{p_2}$ are the quality of service categories of $\widehat{i^m}$, that is the current selected instance of microservice m. Consequently, the total number of states for each service type is $|W^m| = |C^{p_1}| \times |C^{p_2}| \times |I^m|$. For instance, in the case of response time and energy as stated in Section 2, the categories used for response time, $C^{Res} = \{LowRes, MedRes, HighRes\}$ and for energy consumption, $C^E = \{LowE, MedE, HighE\}$. These categories are used to build the states of the Q-learning algorithm, which are then passed into the *Reward Calculator*.

Reward Calculator is responsible for mapping the state of the microservice instance into an integer value by using a reward function $r(C^P)$. At any point, the reward is calculated using a linear combination of the: $r_{t+1} = x_1 \cdot r(c^{p1}) + x_2 \cdot r(c^{p2}) + \dots + x_n \cdot r(c^{pn})$. For example, considering response time (R) and energy consumption (E) as the parameters, the immediate reward r_{t+1} for a selection will be the following linear combination: $r_{t+1} = x_1 \cdot r(c^R) + x_2 \cdot r(c^E)$ where $r(c^R)$ is a positive value for $\{LowRes, MedRes\}$ and a negative value for $HighRes$ and, $r(c^E)$ is a positive value for $\{LowE, MedE\}$ and a negative value for $HighE$. Adjusting the coefficients vector $[x, y]$, it is possible to create different balancing profiles, preferring performance or energy consumption. Once the reward is computed for a given state, w, the next step is to select one of the instances given the current state w.

Table 1: Evaluation metric parameters.

Parameter	Description	Value
τ	Time Period	60 secs
p_{ev}	Penalty for energy violations	0.8
p_{rt}	Penalty for response time violations	0.8
R_{max}	Maximum Response time	0.8 seconds
R_{min}	Minimum Response time	0.2 seconds
E_{max}	Maximum energy	1.45 joules
w_r	Weights on response time	0.4
w_e	Weights on energy savings	0.6

Instance Selector is responsible for the selection of an instance $\bar{i}$ given a state s. The selection of an instance is equivalent to the selection of an action a given a state s. Therefore, the set of possible actions of this MDP is $(A^m = I^m = \{i_1^m, i_2^m, \dots i_n^m\}$ where i_n^m is the action to select the instance i_n of the microservice m. To make optimal selections and to improve with every selection, Q-learning makes use of a simple lookup table, Q-table. The Q-table is initialized for every microservice m and contains all the possible (state, action) pairs $Q(s_t, a_t)$, called Q-values. Thus, each Q-table size is given by $|Q^m| = |S^m| \times |A^m|$. Each Q-table is initialized with zero values. Then, for each arriving request and, consequently, selection, the Q-table is updated according to the equation: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta \cdot (r_{t+1} + \gamma \cdot \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$ Starting from the current state s_t, the possible next states s_{t+1} are the ones generated with the forecasts computed by the predictor. Among them, Q-learning selects the state s_{t+1} with the highest Q-value. Therefore, the action a_t taken to arrive in state s_{t+1} is the service instance associated with the state s_{t+1}. After each selection, the Q-table is updated according to the previously cited formula, discounting the rewards with the discount factor γ and providing gradual learning with the learning factor η. Selection after selection, every entry of the Q-tables will fill up with different Q-values, and the Q-learning algorithm will learn how to behave in any situation to improve the instance selection strategy.

5 EXPERIMENTS AND EVALUATION

In this section, we elaborate on the evaluation of the proposed *ML-enabled Service Discovery* approach through SockShop, a widely used microservice exemplar developed for the e-commerce domain described in Section 2.

Experimentation Setup – We made use of the SockShop (ref. Section 2) installed in a machine with 16 GB of RAM and a 3.2 GHz processor. Each of the six microservices was replicated five times, making it a system with a total of 30 microservice instances. The *API Gateway*, *Service Discovery*, *Machine Learning Engine*, and *QoS Monitor* components were implemented in Python[2].

Evaluation Metrics – To measure the effectiveness of the approach, we make use of the Utility Score (U), as defined in Section 2, using normalized values for R_τ and E_τ for every τ-period (see values in Table 1). We assign a slightly higher weight to the w_e than w_r because, although reducing response time is our priority, we do not want to do it at the expense of sustainability.

[2]Implementation details are available at: https://github.com/karthikv1392/ML_SD_Tradeoff

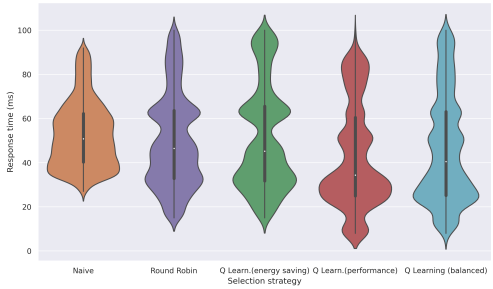

Figure 5: Distribution of response time values

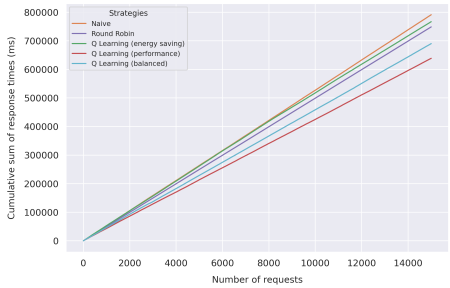

Figure 6: Cumulative response time

Evaluation Candidates – We evaluated the approach by deploying the system integrated with an ML-enabled service discovery mechanism that makes use of five different self-adaptation strategies for a period of 24 hours. These strategies represent the evaluation candidates: *Naive* always selects the same instance of a microservice for a given service discovery query; *Round-robin* follows a round-robin strategy in selecting instances for every service discovery query; *Q-learning-perf* uses Q-learning for service discovery and gives more priority to response time over energy consumption while making the selection (achieved by using reward function coefficients $[x_1 = 0.3, x_2 = 0.7]$); *Q-learning-energy* uses Q-learning for service discovery and gives more priority to energy consumption over response time while making the selection (achieved by using reward function coefficients $[x_1 = 0.8, x_2 = 0.2]$); *Q-learning-balanced* uses Q-learning for service discovery and balances energy consumption and response time while making the selection (achieved by using reward function coefficients $[x_1 = 0.45, x_2 = 0.55]$, slightly higher value is kept for x_2 to ensure balanced trade-off keeping energy as the main focus). For each of the three strategies based on Q-learning, the Q-values of each strategy are updated with the same learning rate $\eta = 0.08$ and discount factor $\lambda = 0.9$.

Data Setup – To collect the data for training the ML models, we deployed the SockShop system with a service discovery mechanism that selects the instances using a round-robin approach. Further, the system was executed for a week, and the workload to the different microservices was simulated based on a real-world benchmark trace of *FIFA 98 World Cup* site[3]. The uncertainty in this scenario is induced by the sudden variability in the number of requests based on the day time of the day, resulting in the high CPU of the host machine and, thereby, high response time. This system is executed for seven days. During this period, the response time and energy consumption of each of the instances were recorded. The response time data was used to develop LSTM models to response time for each of the six microservices with a forecast horizon (H) of 1 minute and lag value (l) of 10 minutes. On the other hand, the energy values were used to create the three categories.

5.1 Evaluation

As a first step, we evaluate the effectiveness of *ML-enabled Service Discovery* when using only response time as the key QoS parameter. This is represented by the strategy *Q-learning-perf*. To this end, we computed the response time offered by the selected instances while using each of the five evaluation candidates. Figure 5 shows the distribution of response time values for each selection strategy. The performance-oriented strategy (*Q-learning-perf*) has the lowest median value. This indicates that *Q-learning-perf* can select instances that are performing better, adapting to changing conditions. This is further demonstrated in Figure 6, where we can observe that with time, the distance between different approaches starts increasing, and *Q-learning-perf* can improve. However, we can notice from Figure 7 and Figure 8 that when it comes to energy consumption, the effectiveness of *Q-learning-perf* is not the best compared to other approaches.

As a second step, we evaluate the effectiveness of the *ML-enabled Service Discovery* approach when it uses only energy consumption as the key QoS attribute over response time. As depicted in Figure 7, *Q-learning-energy* performs marginally better than *Q-learning-balanced* in terms of energy consumption (almost 2% better) due to the selection of the less consuming instance at each request. This is further demonstrated in Figure 8, which shows the cumulative plot of energy consumption when using each approach. As in the case of response time, *Q-learning-energy* strategy continuously improved to guarantee higher energy savings. However, as we can observe, the *Q-learning-perf* strategy consumes the second maximum energy. This further emphasizes that the approach, which works well when it comes to optimizing response time, is not effective with respect to energy efficiency.

Finally, we want to evaluate the ability of our approach to perform a trade-off aware balanced service discovery considering both response time and energy as parameters. As can be observed in Figure 9, *Q-learning-balanced* is able to offer considerably low energy consumption while maintaining the response time. This is further emphasized through the results presented in Figure 10, which represents the cumulative utility accrued by the different approaches over the course of the execution of the system. The plot shows how the utility offered by Q-learning-based approaches starts diverging marginally during the initial stage compared to naive and round-robin approaches, but then the gap between Q-learning-based approaches and other approaches keeps on increasing. This further demonstrates the ability of Q-learning to improve continuously over time. It is also worth pointing out that while *Q-learning-balanced* has shallow utility compared to *Q-learning-energy*. As time progressed with more and more selection and feedback, *Q-learning-balanced* can improve and grow at a much higher rate compared to other approaches. Hence, *Q-learning-balanced* is able

[3]ftp://ita.ee.lbl.gov/html/contrib/WorldCup.html

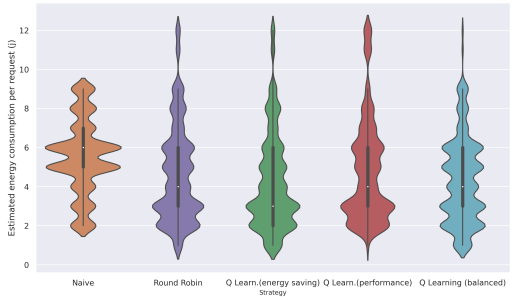

Figure 7: Distribution of energy consumption

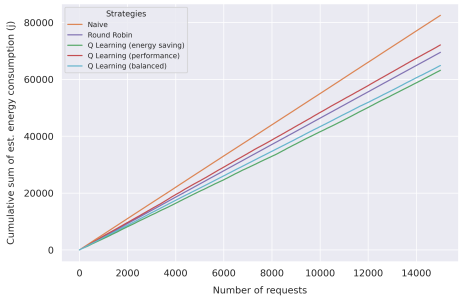

Figure 8: Cumulative energy consumption

to make optimal decisions by trading between energy consumption and response time.

The efficiency of the approach was evaluated by measuring the time taken by the end-to-end discovery process to return a selection when integrated with our ML-enabled approach using the combination of prediction and Q-learning. The results show that, on average, the approach takes 0.10 seconds to perform the whole process with a worst-case scenario of 0.20 seconds. This is just marginally higher than *Naive* and *Round-robin* approaches, which are mostly leveraged in many traditional service discovery mechanisms. The speed can be mainly attributed to the fact that Q-Learning, being a model-free technique, performs only a lookup operation in the Q-Table. The majority of the time is taken by the prediction process, as although it's a constant time process, the prediction needs to be done for each of the instances of a given microservice. Training of the LSTM networks does not impact real-time service discovery.

6 THREATS TO VALIDITY

Threats to *construct validity* are related to the use of a controlled experimental setup and incorrect selections. Even though we performed *real-time execution* of the system, we simulated the workload based on the real-world benchmark, namely the FIFA 98 World Cup site logs. Further, for the sake of simplicity and feasibility, energy consumption was considered by associating a static estimated energy value to each instance configuration. In this way, some instances will have a greater energy efficiency ratio and others less. Finally, as machine learning is notoriously expensive in terms of energy consumption, further experimentation in real settings is required to measure the overhead introduced by the *ML-enabled Service Discovery* on global energy consumption.

Threats to *internal validity* concern the use of a static number of microservice instances in the experimentation. Even though this is the case, as explained in Section 5 (Data setup), the variability in the request pattern induces the uncertainty required to demonstrate the capability of our approach. More work is required to enhance the approach to handle the uncertainty induced by the dynamic addition/removal of instances.

Threats to *external validity* concern the scalability of our approach. In fact, as machine learning is notoriously expensive in terms of computational resources, the *ML-enabled Service Discovery* might be hardly usable in large-scale settings, where the microservices applications might involve hundreds of service instances. To this end, we plan to run extensive experimentation in a real setting and investigate methods and techniques to optimize Q-Learning (e.g., using Deep Q-Learning [13]) to solve the effectiveness and efficiency issues that might arise from the increased state and action space.

7 RELATED WORK

While a large body of work exists about *Service Discovery*, we summarize hereafter only those approaches that consider QoS attributes and Self-adaptation.

QoS-aware Service Discovery has been first investigated in the context of SOA. For example, [2] leverages hidden Markov model for discovering Web services and predicting their response time, whereas [3] combines global and local optimization algorithms for discovering and composing services.

More recently, service discovery mechanisms have been largely investigated in the context of the Internet of Things (IoT) [7][6]. In this context, self-adaptation is used to deal with the inherent dynamics of IoT environments and with changes in service quality, network topology, or service usage.

Machine learning approaches have been largely investigated for developing recommendation systems for Web Services and demonstrated to be effective and efficient. For example, in [4], authors introduce a framework for optimizing service selection based on consumer experience (i.e., context), and preferences (i.e., utility). The framework maintains a set of predefined selection rules that are evolved at run time by means of reinforcement learning.

In the context of microservice architectures, machine learning has been exploited to address some important aspects: in [1], unsupervised learning is used to automatically decompose a monolithic application into a set of microservices; in [11] reinforcement learning has been used for considering QoS factors while assembly services; in [9], bayesian learning and LSTM are used to fingerprint and classify microservices; in [17], reinforcement learning is used to autoscale microservices applications, whereas in [19] random forest regression is used to implement an intelligent container scheduling strategy; in [15] authors describe a data-driven service discovery for context-aware microservices. Finally, [8] leverages machine learning to select micreservice instances fulfilling the QoS requirements. In particular, the work relies on a single forecast model for the entire system to deal with QoS variability.

On the other hand, this work presents a general *ML-enabled Service Discovery* approach that can be applied to any class of microservice system (built using docker stack), as demonstrated by

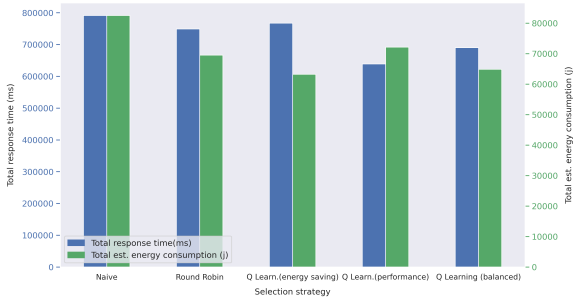

Figure 9: Energy consumption and response time

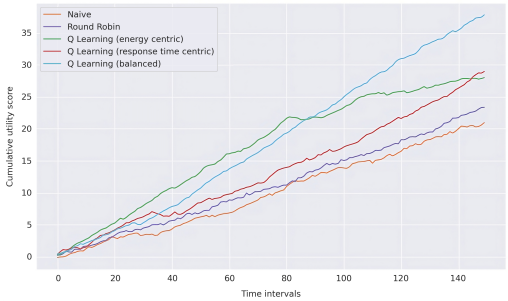

Figure 10: Utility accrued by the different selection strategies

applying it to a state-of-the-art exemplar. Moreover, departing from other approaches (e.g., [8]), in our approach, the models are created per service, thereby ensuring fine-grained control. Finally, this work combines two different techniques, namely ML and self-adaptation, to mitigate the uncertainties emerging from frequent changes in the system context and to trade off different QoS attributes.

8 CONCLUSIONS AND FUTURE WORK

The main challenge addressed in this work deals with the need for service discovery to adapt to dynamic changes in the QoS of the microservice instances as well as to select instances based on trade-offs between multiple QoS parameters.

To this end, we proposed a novel *ML-enabled Service Discovery* approach, which leverages machine learning and self-adaptation. In particular, deep learning models are trained to forecast the behavior of a containerized application, and a Q-learning-enabled selector performs the selection between the predicted states of requested instances by trading off between different QoS attributes.

The work demonstrated how to incorporate machine learning into microservice architectures, especially for service discovery, without introducing overheads. Moreover, as the approach architecture extends the well-known server-side discovery pattern, it can be easily implemented in many different application domains.

Finally, we believe that many other problems can be solved with the effective use of ML techniques. Therefore, future work includes but is not limited to: (*i*) extending and applying this trade-off aware approach to large-scale microservice-based systems. This may result in developing an advanced version of the Q-learning algorithm that we have employed in this work to handle challenges associated with larger state space and further handling challenges associated with the automatic removal of instances, the addition of instances, etc.; (*ii*) exploring the possibility of using transfer learning to reuse the knowledge obtained from one microservice-based system into another. This may contribute towards faster convergence, reducing too much exploration; (*iii*) integrating this approach with technologies like Netflix Eureka or Zookeeper to create novel tools for service discovery.

ACKNOWLEDGMENTS

This work was partially supported by the Italian PNRR MUR Centro Nazionale HPC, Big Data e Quantum Computing, Spoke9 - Digital Society & Smart Cities.

REFERENCES

[1] Muhammad Abdullah, Waheed Iqbal, and Abdelkarim Erradi. 2019. Unsupervised learning approach for web application auto-decomposition into microservices. *Journal of Systems and Software* 151 (2019), 243–257.
[2] Waseem Ahmed, Yongwei Wu, and Weimin Zheng. 2015. Response Time Based Optimal Web Service Selection. *IEEE Trans. Parallel Distrib. Syst.* 26, 2 (2015).
[3] Mohammad Alrifai, Thomas Risse, and Wolfgang Nejdl. 2012. A Hybrid Approach for Efficient Web Service Composition with End-to-End QoS Constraints. *ACM Trans. Web* 6, 2 (2012).
[4] Jesper Andersson, Andreas Heberle, Jens Kirchner, and Welf Lowe. 2011. Service Level Achievements – Distributed Knowledge for Optimal Service Selection. In *2011 IEEE Ninth European Conference on Web Services*.
[5] Richard Bellman. 1957. A Markovian Decision Process. *Journal of Mathematics and Mechanics* 6, 5 (1957), 679–684.
[6] Christian Cabrera and Siobhan Clarke. 2022. A Self-Adaptive Service Discovery Model for Smart Cities. *IEEE Transactions on Services Computing* 15, 1 (2022).
[7] Christian Cabrera, Andrei Palade, and Siobhán Clarke. 2017. An Evaluation of Service Discovery Protocols in the Internet of Things. In *Proceedings of the Symposium on Applied Computing*.
[8] Mauro Caporuscio, Marco De Toma, Henry Muccini, and Karthik Vaidhyanathan. 2021. A Machine Learning Approach to Service Discovery for Microservice Architectures. In *Proceedings of 15th European Conference in Software Architecture (Lecture Notes in Computer Science)*, Vol. 12857.
[9] Hyunseok Chang, Murali Kodialam, T.V. Lakshman, and Sarit Mukherjee. 2019. Microservice Fingerprinting and Classification using Machine Learning. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*. 1–11.
[10] Chris Chatfield. 2000. *Time-series forecasting*. Chapman and Hall/CRC.
[11] Mirko D'Angelo, Mauro Caporuscio, Vincenzo Grassi, and Raffaela Mirandola. 2020. Decentralized learning for self-adaptive QoS-aware service assembly. *Future Generation Computer Systems* 108 (2020), 210 – 227.
[12] Alex Graves. 2012. Supervised sequence labelling. In *Supervised sequence labelling with recurrent neural networks*. Springer, 5–13.
[13] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, et al. 2018. Deep Q-learning From Demonstrations. In *AAAI*.
[14] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[15] Zeina Houmani, Daniel Balouek-Thomert, Eddy Caron, and Manish Parashar. 2020. Enhancing microservices architectures using data-driven service discovery and QoS guarantees. In *20th International Symposium on Cluster, Cloud and Internet Computing*.
[16] J. O. Kephart and D. M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003).
[17] Abeer Abdel Khaleq and Ilkyeun Ra. 2021. Intelligent Autoscaling of Microservices in the Cloud for Real-Time Applications. *IEEE Access* 9 (2021).
[18] John F. Kolen and Stefan C. Kremer. 2001. *Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies*.
[19] Jingze Lv, Mingchang Wei, and Yang Yu. 2019. A Container Scheduling Strategy Based on Machine Learning in Microservice Architecture. In *2019 IEEE International Conference on Services Computing (SCC)*. 65–71.
[20] Joseph Muli. 2018. *Beginning DevOps with Docker*. Packt Publishing.
[21] Chris Richardson. 2018. *Microservices Patterns*. Manning Publications.
[22] Sima Siami-Namini, Neda Tavakoli, and Akbar Siami Namin. 2018. A comparison of ARIMA and LSTM in forecasting time series. In *Proc. of International Conference on Machine Learning and Applications*.
[23] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8, 3-4 (1992), 279–292.