A Machine Learning Approach to Service Discovery for Microservice Architectures

Mauro Caporuscio[1(✉)][0000−0001−6981−0966], Marco De Toma[2], Henry Muccini[2][0000−0001−6365−6515], and Karthik Vaidhyanathan[2][0000−0003−2317−6175]

[1] Linnaeus University, Sweden
mauro.caporuscio@lnu.se
[2] University of L'Aquila, Italy
{marco.detoma, henry.muccini, karthik.vaidhyanathan}@univaq.it

Abstract. Service discovery mechanisms have continuously evolved during the last years to support the effective and efficient service composition in large-scale microservice applications. Still, the dynamic nature of services (and of their contexts) are being rarely taken into account for maximizing the desired quality of service. This paper proposes using machine learning techniques, as part of the service discovery process, to select microservice instances in a given context, maximize QoS, and take into account the continuous changes in the execution environment. Both deep neural networks and reinforcement learning techniques are used. Experimental results show how the proposed approach outperforms traditional service discovery mechanisms.

Keywords: Service Discovery · Machine Learning · Microservices Architecture

1 Introduction

Microservices have become enormously popular since traditional monolithic architectures no longer meet the needs of scalability and rapid development cycle. The success of large companies (Netflix among them) in building and deploying services is also a strong motivation for other companies to consider making the change. The loosely coupled property of microservices allow the independence between each service thus enabling the rapid, frequent and reliable delivery of large, complex applications [7].

Like most transformational trends, implementing microservices poses its own *challenges:* Hundreds of microservices may be composed to form a complex architecture; tens of instances of the same microservice can run on different servers; the number or locations of running instances could change very frequently. Moreover, the set of service instances changes dynamically because of autoscaling, failures, and upgrades. Therefore, one of the challenges in a microservice architecture concerns how services *discover*, *connect*, and *interact* with each other. Consequently, elaborated *service discovery mechanisms* are required [19].

Service discovery mechanisms has continuously evolved during the last years (e.g., Consul, Etcd,Synapse, Zookeeper, etc). A huge effort has been reported to make the service discovery effective and efficient by improving the functional matching capability, i.e., – discovering all the available instances of a specific microservice very quickly –, while delegating QoS concerns to external load-balancing components (e.g., AWS Elastic Load Balancing).

These solutions do not take explicitly into account the *context* and *quality of services*, that are transient and continuously change over time because of several different reasons – e.g., a service consumer/provider can change its context because of mobility/elasticity, a service provider can change its QoS profile according to day time, etc. In this settings, our approach envisages a new service discovery mechanism able to deal with uncertainty and potential adverse effects attributed to frequent variability of the context and QoS profile of services.

This paper proposes the use of Machine Learning (ML) techniques as part of the service discovery process, so to perform context-driven, QoS-aware service discovery. During the process of service discovery, our approach *i*) uses deep neural networks to predict the QoS of microservice instances; *ii*) extracts context information from service consumer such as location, time of request, etc.; *iii*) keeps track of the execution context of each microservice instances; *iv*) leverages the context and QoS data obtained using Reinforcement Learning (RL) technique to select the instance that is expected to guarantee the optimal QoS.

The *contribution* of this work is as follows: i) it proposes a machine learning approach to be used for service discovery; ii) it defines a microservices architecture framework that integrates ML in service discovery and service selection; iii) it provides experimental results comparing QoS offered by traditional service discovery mechanism with that of our ML approach.

In the *rest of the paper*, Section 2 presents related work on (ML for) service discovery in (micro)service applications. Section 3 introduces the terminology and notation used in the paper. Section 4 presents our ML-based service discovery proposal. Section 5 evaluates the benefits of our approach through an exemplar application. Conclusions are provided in Section 6.

2 Related Work

Work related to the general concept of *discovery* is manifold and ranges from architecture (e.g., centralized, decentralized) to matching mechanisms (e.g., QoS-aware, context-aware, and semantic-aware) and selection criterion (e.g., single objective, multi-objectives).

While a large body of work exists in the context of SOA for each of these categories, we summarize hereafter only those approaches which consider QoS in conjunction with context.

Contextualization refers to the ability to discover, understanding and selecting services of interest deployed within the environment. To this end, a key role is played by ontologies, which have been employed to make service discovery context-aware [23]. The SAPERE framework implements a service discovery

that accounts for contextual and QoS factors [21] in pervasive networking environments. The EASY framework focuses on pervasive services and specifically targets extra-functional properties by rating services according to user preferences on extra-functional properties [16]. Finally, in [3] service discovery considers contextual factors while discovering and assembling services of interest satisfying QoS constraints.

ML have been already employed for developing recommendation systems for Web Services, and demonstrated to be effective and efficient. In [18], the authors propose a number of data mining methods to improve service discovery and facilitate the use of Web services. In [2], 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 a reinforcement learning strategy.

More recently, various ML techniques have been exploited for addressing important aspects related to microservice architectures. In [1], unsupervised learning is used to automatically decompose a monolithic application into a set of microservices. In [6] reinforcement learning has been used for considering QoS factors while assembly services. In [4], bayesian learning and LSTM are used to fingerprint and classify microservices. In [13], reinforcement learning is used to autoscale microservices applications, whereas in [15] random forest regression is used to implement intelligent container scheduling strategy. Finally, in [12] authors describe a data-driven service discovery for context-aware microservices.

In this work, we make use of ML techniques for selecting services of interest that fulfill the QoS requirements in a given context. In particular, we rely on ML to mitigate the uncertainty and variability emerging from frequent changes in services context and QoS profiles. To this end, next sections will present the approach formalization, and evaluation.

3 System Model

In this section we introduce the terminology and notation used in the rest of the paper, define the model of the system we are considering, and formally define the performance indexes we will use to measure the effectiveness of our approach.

Figure 1 shows the structure of the *server-side discovery* pattern [19], where the clients – i.e., the API Gateway or another services – make requests via the *Router*, which is deployed at a known location. The *Router* queries a *Service Registry*, selects the proper service instance, and forwards the request to it. *Service Registry* usually is a database of services, their instances and their locations. Service instances are registered with the *Service Registry* on startup and unregistered on shutdown.

The model of the system under consideration builds on such a general definition of the Server-side Service Discovery Pattern. In particular, let $\mathcal{S}$ be a set of services, hosted by different nodes in a networked system (e.g., edge, fog, or cloud architecture). A Service $s \in \mathcal{S}$ is defined as a tuple (i, c, p, e), where: $s.i \in \mathcal{I}$ denotes the *interface* provided by the service (i.e., the functionality provided by

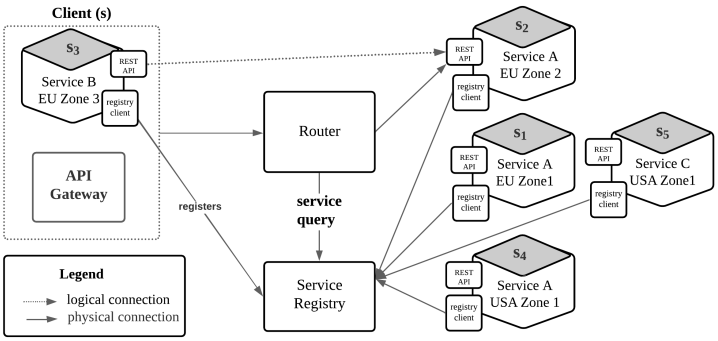


Fig. 1: Server-side Service Discovery Pattern

s[3]), $s.c \in \mathcal{C}$ denotes the *context* of the service s, $s.p \in \mathcal{P}$ denotes the *quality profile* of the service s, and $s.e \in \mathcal{E}$ denotes the *endpoint* of the service s.

Further, a selection criteria $q \in \mathcal{Q}$ is a couple (c, p), where $q.c \in \mathcal{C}$ denotes the context of interest, and $q.p \in \mathcal{P}$ denotes the quality profile of interest.

Given a selection criteria $q \in \mathcal{Q}$, and a service set $S \in 2^{\mathcal{S}}$, the *Service Discovery* mechanism is defined according to two different functions, namely *match* and *select*. On the one and $match : \mathcal{I} \times 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}$ is a function that given a interface $i \in \mathcal{I}$ and a set of services $S \subseteq \mathcal{S}$, returns a set of services $\overline{S} \subseteq S$ such that $\overline{S} = \{s | s.i = i\}$. On the other hand, $select : \mathcal{Q} \times 2^{\mathcal{S}} \rightarrow \mathcal{S}$ is a function that given a selection criteria $q \in \mathcal{Q}$ and a set of services $S \subseteq \mathcal{S}$ returns a service $\hat{s} \in S$ such that $(\hat{s}.c \cong q.c) \wedge (\forall s \neq \hat{s} : \hat{s}.p \succeq s.p)$, where $\cong$ and $\succeq$ are defined according to some suitable methods [3][6].

Therefore, the Service Discovery mechanism can be defined as a function that, given an interface of interest i, a selection criteria q and a set of registered services $RS \subseteq \mathcal{S}$, returns a service instance $\hat{s}$, such that:

$$\hat{s} \leftarrow ServiceDiscovery(i) \equiv select(q, match(i, RS))$$

Example 1. Let $RS \subseteq \mathcal{S}$ be the set of services stored within the Service Registry depicted in Figure 1:

$s_1 = (\text{“}\texttt{Service A}\text{”}, (\texttt{loc}, \texttt{EU}), ((\texttt{responseTime}, \texttt{0.5s}), (\texttt{throughput}, \texttt{100r/s})), \texttt{IP}_1)$
$s_2 = (\text{“}\texttt{Service A}\text{”}, (\texttt{loc}, \texttt{EU}), ((\texttt{responseTime}, \texttt{0.3s}), (\texttt{throughput}, \texttt{300r/s})), \texttt{IP}_2)$
$s_3 = (\text{“}\texttt{Service B}\text{”}, (\texttt{loc}, \texttt{EU}), ((\texttt{responseTime}, \texttt{0.2s}), (\texttt{throughput}, \texttt{500r/s})), \texttt{IP}_3)$
$s_4 = (\text{“}\texttt{Service A}\text{”}, (\texttt{loc}, \texttt{USA}), ((\texttt{responseTime}, \texttt{0.1s}), (\texttt{throughput}, \texttt{100r/s})), \texttt{IP}_4)$
$s_5 = (\text{“}\texttt{Service C}\text{”}, (\texttt{loc}, \texttt{USA}), ((\texttt{responseTime}, \texttt{0.6s}), (\texttt{throughput}, \texttt{200r/s})), \texttt{IP}_5)$

Let suppose s_3 (of type `Service B`) is interested in sending a request to a service of type $i =$ “`Service A`”. The *Router* invokes the Service Discovery mechanism implemented by the *Service Registry* by providing i as input parameter. Applying the *match* function we obtain $\overline{S} = \{s_1, s_2, s_4\} \leftarrow match(i, RS)$.

[3] We use interchangeably the terms *interface* and *type* to denote the functionality of a service s.

Once retrieved the set of instances $\overline{S}$ implementing `Service A`, the *Router* makes use of the *select* function by providing a selection criteria q opportunely defined. For example, let suppose s_3 (which is located in EU) is interested in a `Service A` providing best response time. Therefore, defining $q \in \mathcal{Q} = ((\texttt{loc}, \texttt{EU}), \texttt{responseTime})$ and applying the function *select*, we obtain $\hat{s} = s_2 \leftarrow select(q, \overline{S})$ where, $\cong$ is defined according to geographical proximity (i.e., "close to"), and $\succeq$ is defined according to a less-is-better relationship (i.e., "$\leq$"). Then, s_2 is the `Service A` instance providing the "best response time" among those instances located "close" to s_3 (i.e., s_1, and s_2).

In order to evaluate our approach, we define a performance index measuring the *QoS* delivered by all services. To this end, let $SB_t \subseteq \mathcal{S}$ be the set of services bound at a given time t. To measure the overall system performance, we define the *Average QoS* delivered by all services in SB_t:

$$\overline{QoS}(SB_t) = \frac{1}{|SB_t|} \sum_{s \in SB_t} s.p \tag{1}$$

4 ML Based Service Discovery

In this section, we describe how our approach uses a combination of ML techniques to perform context-aware service discovery.

For each query, $q \in \mathcal{Q}$ received by the *Service Registry* from a service consumer s, the overall goal of the approach is to select the matching service provider $\hat{s} \in RS$ so as to maximize the QoS perceived by s with respect to the context of the service consumer, s as well as with that of $\hat{s}$.

Figure 2 shows the overall flow of the approach. On receiving a service query, the *Service Registry* component first identifies a matching set, $\overline{S}$ of instances based on the strategy defined in Section 3. It then uses a combination of ML techniques to select the best instance $\hat{s}$ from $\overline{S}$.

The first part of the *select* function is to estimate the expected QoS of every instance in $\overline{S}$. This is due to the fact that, based on the change in contexts such as time of request, execution memory of instance, etc, there can be variations in the QoS offered by instances in $\overline{S}$ and neglecting this can lead to sub-optimal selections. Towards this, our approach uses deep neural networks which considers the historical QoS data offered by the instance along with their context data to *predict* the expected QoS for every instance in $\overline{S}$. The QoS forecasts alone are not sufficient for selection as the perceived QoS of $\hat{s}$ changes based on the context of s. In order to accommodate this, our approach further uses RL technique to select the best instance $\hat{s}$ from $\overline{S}$.

The overall ML process of the approach primarily consists of two phases (as represented in the figure 2) namely the *Batch Phase* and the *Real-Time Phase*. Training and building ML models for performing forecasts is a time consuming process. For this reason our approach involves periodic execution of *Batch Phase* where the historical *Quality Profile* information of each service $s_i \in RS$ as well as their context information are used for training and building models for

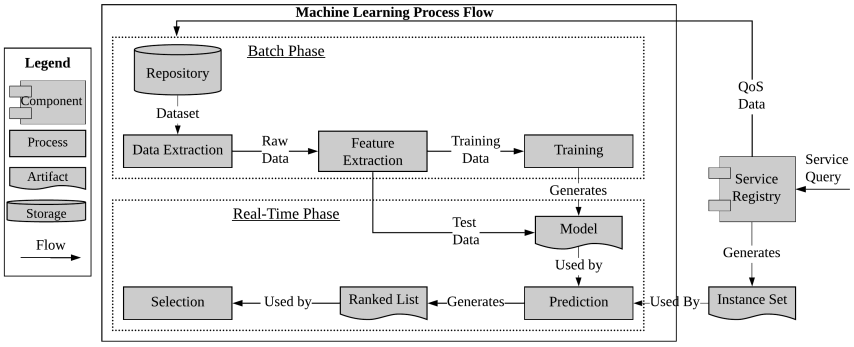


Fig. 2: Machine Learning Process Flow

forecasting the QoS. The *Real-Time Phase* on the other hand, consist of the instance selection process that happens in real-time. In order, to achieve this, it uses the latest ML models available from the batch phase to continuously forecast the expected *Quality Profile* of each service $s_i \in \overline{S}$. It further processes this forecast along with context of s using RL technique to *select*, $\hat{s}$ from $\overline{S}$.

4.1 Data Extraction

The Quality Profiles (i.e., $s.p$) of all the service instances (e.g., response time, throughput, etc) as well as the context information (i.e., $s.c$) which includes details such as instance memory, geographical location, etc. are continuously monitored and stored in a *Repository* component in the *Service Registry*. This forms the raw QoS/Context data. During the batch phase, the *Data Extraction process* retrieves the QoS/Context data from the *Repository*. This raw monitoring data contains the information of different QoS/Context attributes of every instance for different intervals of time. This data is sent to *Feature Extraction* process for further processing.

4.2 Feature Extraction

This process converts the raw monitoring data extracted during the data extraction process into structured set of features as required by the ML technique.

This data has a temporal nature and we use this to convert the problem of predicting QoS into a *time-series forecasting* problem. The QoS data with respect to time forms a *continuous time-series* [5]. For the ease of analysis, we first convert this into a *discrete time-series* by aggregating the data into uniform time intervals.

Let us assume to have m different service instances $s_1, \ldots, s_m$ providing a given interface $i \in \mathcal{I}$, which have been running for n units of time. Each of these m instances has an associated d dimensional *Context Feature-set* $C_f \subseteq \mathcal{C}$ describing the context of each running instance s_j.

Definition 1 (Context Feature-set). *We define context feature as a set* $C_f \subseteq \mathcal{C} = \{c_1, c_2, .., c_n\}$ *where* c_j *represents a pair* (l, v) *such that* l *is a unique label identifier for a context attribute, and* v *denotes the value of the context attribute.*

For example, $C_f = \{(\texttt{loc}, \texttt{IT}), (\texttt{day}, \texttt{Mon}), (\texttt{hour}, \texttt{10}), (\texttt{min}, \texttt{30})\}$ denotes that a given service instance $s \in S$ is located in `Italy`, day is `Monday` and the current time of invocation is `10:30`.

Then for each instance, the observation at any instant of time t can be represented by a matrix $O \in V^{d \times n}$ where V denotes the domain of the observed features. The process of generating time-series results in the formation of a sequence of the form $O_1, O_2, O_3 .... O_t$. The problem of forecasting QoS values is then reduced to predicting the most likely k-length sequence in future given the previous j observations that include the current one:

$$O_{t+1}, ..., O_{t+k} = \underset{O_{t+1},..O_{t+k}}{argmax} \; P(O_{t+1}, ..., O_{t+k} | O_{t-j+1}, O_{t-j+2}, ..., O_t)$$

where P denotes the conditional probability of observing $O_{t+1}, ..., O_{t+k}$ given $O_{t-j+1}, O_{t-j+2}, ..., O_t$. Since we also consider the context data for forecasting the QoS level of each instance, the observation matrix, O can be considered as a multivariate time-series dataset and as the forecasting needs to be done for the next k steps, the problem of time series forecasting becomes a multivariate multi-step forecasting problem [5].

Each column in O represents a feature vector, v. The process of feature scaling is applied to each column v such that $v_i \mapsto [0, 1] \; \forall v \in O$. O is then divided into two data sets, training set, O_{train} and testing set, O_{test} in the ratio 7:3 respectively. The training set obtained is further sent to the *Training* Process.

4.3 Training

The training process uses the training set, O_{train} to create ML models for forecasting the expected QoS of every service instance, $s_j \in RS$ for a given time period known as forecast horizon, H.

Definition 2 (Forecast Horizon). *We define a forecast horizon,* H *as a couple* (h, u) *where* $h \in \mathbb{R}$ *represents the time horizon value, and* u *is the unit identifier for the time horizon value specified.*

The approach makes use of Long Short Term Memory (LSTM) network [11], a class of Recurrent Neural Network (RNN) [8] for building the forecast models. LSTM's have shown to be very effective in time-series forecasting [20] as they have the ability to handle the problem of long-term dependency better known in the literature as the *Vanishing Gradient Problem* [10], as compared to traditional RNN's. In our previous work [17], we have shown how LSTM networks can be used for forecasting QoS of the sensor components and why they are more effective when compared to traditional models like ARIMA. In this work, we use the same approach to train the LSTM network for forecasting the QoS of service instances. The training process results in the creation of a *Model* which is further

tested for accuracy using the test set, O_{test}. In the event of a low accuracy, the approach performs a retraining by tuning the neural network parameters. The tested models are further used by the *Prediction* process.

Training is executed as batch process in regular intervals to update the models so as to avoid the problems of concept drift [22].

4.4 Prediction

Prediction is a real-time process responsible for forecasting QoS of the matching instances. For every service discovery request received by the service registry, the prediction process uses the trained LSTM models to generate the QoS forecasts for each $s_j \in \overline{S} \leftarrow match(i, SR)$.

Definition 3 (QoS Forecasts). *We define the QoS forecasts as a set,* $F = \{p_1, p_2, .., p_n\}$ *where* p_j *represents the forecasted quality profile. Note that,* $p_j = (a, v)$ *where* a *identifies the quality attribute and* $v \in \mathbb{R}$ *denotes the forecasted quality value over the duration of the time horizon* h*.*

For example, $H = (\texttt{10}, \texttt{sec})$ and $F = \{(\texttt{responseTime}, \texttt{0.2}), (\texttt{throughput}, \texttt{200})\}$ for a given instance s denotes that s is expected to have an average response time of 0.2 seconds and a throughput of 200 requests/second in the next 10 seconds.

These QoS forecasts for each of the service instances $s_j \in \overline{S}$ are then sent to the *Selection* process for further processing.

4.5 Selection

The role of selection process is to select the best instance $\hat{s}$, such that, $\hat{s} \leftarrow select(q, \overline{S})$, based on the context of the service consumer as well as the forecasts of the expected QoS of each instance $s_j \in \overline{S}$. In order to achieve this our approach uses a RL technique in particular Q-learning [24]. Q-learning is a widely used method for decision-making scenarios due to their ability to come up with optimal decisions through a model-free learning approach. The key part of using Q-learning is to divide the problem into set of states, W, actions, A and rewards, R. The state space of our Q-learning approach is determined by two important attributes: (i) *QoS Categories*, and (ii) *Context Feature-set.*

Definition 4. *(QoS Categories) We define QoS category, QC as a discrete set* $\{qc_1, qc_2, ...qc_n\}$ *where* qc_j *represents the expected category for the QoS metrics* j*, obtained by mapping the values of the QoS forecasts* f_j *to a unique label,* $l \in \mathcal{L}$*.*

For instance, let $F = \{(\texttt{responseTime}, \texttt{0.3}), (\texttt{throughput}, \texttt{100})\}$ represent the forecast vector and $L = \{\texttt{lowRt}, \texttt{highRt}, \texttt{lowTh}, \texttt{highTh}\}$ denote a set of labels. Then we can define a simple mapping function for the QoS attribute, *responsetime* such that $[0, 0.2] \mapsto \texttt{lowRt}, [0.2, \infty] \mapsto \texttt{highRt}$. Similarly, we can define another mapping function for the QoS attribute, *throughput* such that $[0, 40] \mapsto \texttt{lowTh}, [40, \infty] \mapsto \texttt{highTh}$. We can then combine this to generate a QoS category set, QC for the given F as $QC = \{highRt, highTh\}$.

Algorithm 1 Instance Selection Algorithm

Require: :
1: States $W = \{w_1, w_2, w_3, ..., w_n\}$
2: Actions $A = \{0, 1, 2....m\}$ ▷ represents the selection of each m instance
3: Labels $L = \{l_1, l_2, l_3, ...l_p\}$ ▷ Threshold categories
4: Forecasts $F = \{f_1, f_2, ..f_n\}$ ▷ Forecasts of QoS attributes
5: Rewards $R = \{r_1, r_2, ..., r_n\}$ ▷ Reward for each of the state
6: **procedure** DECISION-MAKER(W,A,L,F,R,$\mathcal{C}$,α,γ) ▷ Find the state of the system from the forecasts and context
7: $QC \leftarrow identify\_category(F, L)$ ▷ Get Category from Forecasts
8: $w \leftarrow (QC, c)$ ▷ combine category and context to form the state
9: $r \leftarrow R[w]$ ▷ Reward for attaining the state, w
10: $(w', a') \leftarrow argmax_a Q(w, a)$
11: $Q'(w, a) = (1 - \alpha) * Q(w, a) + \alpha * (r + \gamma * max(Q(w', a)))$
12: $a \leftarrow a'$ ▷ The action to reach that state
13: **return** a

Based on this, the state space, $W \subseteq QC \times \mathcal{C}$ is defined over the set of possible QoS categories and the set of all context features in $\mathcal{C}$.

Action space, A, consists of a set of actions $\{a_1, a_2, a_3, .., a_m\}$ such that $a_j \in A$ denotes the selection of instance s_j in $\overline{S}$. Rewards, R on the other hand is a set $\{r_1, r_2, ..r_n\}$ where $r_j \in \mathbb{Z}$ denotes the reward value for attaining a state, $w_j \in W$.

W and A together form a $n \times m$ matrix where n denotes the number of states and m denotes the number of actions. Q-Table forms the heart of the Q-learning approach where each value corresponds to a (w, a) pair and it's value denotes the relevance of selecting an action, a from a state w. In other words, it denotes the benefit for selecting an instance s_j given the context of the service consumer and the expected QoS category of the instance. This property is leveraged by the selection process to select the best instance during the process of service discovery. The complete instance selection algorithm based on Q-learning is presented in algorithm 1. It uses two key parameters, α and γ where $0 < \alpha \leq 1$ denotes the learning rate which represents the importance given to the learned observation at each step, and $0 < \gamma \leq 1$ denotes the discount factor, which can be considered as the weight given to the next action (instance selection).

The algorithm first maps the forecasts received from the prediction process, F into a set of QoS Categories, QC based on the labels, L. The QC identified along with the context of the service consumer, $\mathcal{C}$ is combined to identify the current state, w inside the Q-Table (lines 7-8). The algorithm then assigns a reward for attaining the state, w. Following this, the maximum value of state-action pair, (w', a') that can be reached from the current state is identified (lines 9-10). The q-table is then updated using the q-function (line 11) and the action (instance, $\hat{s}$) is returned (lines 12-13).

In this manner for every query, q, our algorithm selects the best instance $\hat{s}$ by selecting the action that maximizes the reward. By assigning negative rewards (penalties) to selection of instances that offers sub-optimal QoS, the approach ensures that any incorrect selection is penalized in the form of a high negative reward and this means as the time progresses, given a q, the algorithm continuously improves the selection process to guarantee optimal QoS by ignoring instances that can lead to high penalties.

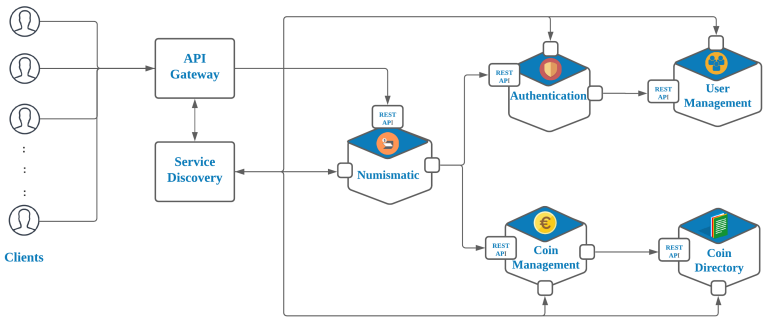


Fig. 3: High-level Architecture of the Prototype Application

5 Evaluation

In this section, we elaborate on the evaluation of the proposed approach through a prototype microservice-based application developed for managing coin collections of the users. Due to space limitations, we omit here the implementation details. The interested reader can find the complete implementation, along with source code and dataset, at the project repository[4]. Rather, we describe the microservice architecture, the experimental setup, and the data and metrics used for the evaluation of our approach. We used *response time* as the key QoS parameter for the evaluation of our approach. Hence, for each $i \in \mathcal{I}$ requested by the service consumer, s, the objective will be to select the instance, $\hat{s}$ that is expected to provide the best response time.

5.1 Proof-of-concept: the Coin Collection application

The *Coin Collection* application provides different functionalities to users who want to collect coins. In particular, users can: i) register to the application by providing details such as name, location, etc; ii) add information on the coins in hand (this information includes details such as name of the coin, country of use, etc; iii) query information about the different types of coins available in the system; iv) retrieve information on the different coins of a specific user or of the nearby users of a given location. *Coin Collection* consists of five key microservices (see Figure 3):

1. *User Management* microservice is responsible for handling user management related operations such as adding new users, deleting users and managing user profiles. It uses a database to store the user profile related information.
2. *Authentication* microservice provides functionalities to ensure that only authenticated users have permission to view/manage user-profiles and other user-related information. It has a simple database to store user credentials.

[4] https://github.com/karthikv1392/ML-SD

3. *Coin Management* microservice provides functionalities to manage coins such as adding/removing coins to/from the collection. It consists of a database for storing coin related information.

4. *Coin Directory* microservice is a simple directory management service that supports the coin management service in fetching additional information related to coin from external APIs.

5. *Numismatic* microservice accomplishes the key functionality of the application. It provides features such as retrieving user information, querying a specific user or nearby users' coin collection, and adding or removing coins from a user collection. It achieves these features by interacting with other microservices, as shown in Figure 3.

External clients interact with the system through the API Gateway, which further checks with the *Service Discovery* to fetch an instance of the desired microservice. Additionally, every microservice queries the service discovery before interacting with other microservices. For example, Numismatic service interacts with coin management microservice. This implies that every time Numismatic service has to interact with coin management microservice, it first needs to identify the instance that needs to be invoked using Service Discovery. The same holds true for other microservices.

5.2 Controlled experiments

Experiment Specification. For experimentation and evaluation, we used the prototype application described in Section 5.1 consisting of 25 microservices. The microservices were implemented using Java Spring Boot. The service discovery module was primarily implemented in Java. The module also supports integration with technologies like Zookeeper, Netflix Eureka, etc. The LSTM part was implemented using Python (Keras framework with Tensorflow backend). Mean Squared Error (MSE) loss function and the efficient Adam version of stochastic gradient descent was used for optimization of the LSTM models [14]. Algorithm 1 was implemented using Java with parameters $\alpha = 0.1$ and $\gamma = 0.1$.

Experimental Setup. Our system was deployed on two VM instances in Google Cloud. The first one was run on a N1-Standard-4 CPU Intel Haswell Processor comprising 4 vCPU and 17 GB RAM with US-Central-a as the geographical zone. The second one ran on a N2-Standard-4 Intel Skylake processor comprising 2 vCPU and 28GB RAM with Europe-West-a as the geographical zone. The microservice instances were distributed between the two VM instances. This was done to capture the different context dimensions that may arise from the type of CPU, number of cores, geographical zones, etc.

Evaluation Metrics. The objective of our evaluation is to assess the effectiveness and efficiency of the approach. The effectiveness of the approach is evaluated by i) measuring the *accuracy* of the response time forecasts produced; ii) calculating the *average response time* delivered by all service in SR, $\overline{QoS}(SB_t)$ as defined in Section 3; iii) computing the response time offered by the instance, $\hat{s}$ for every request made. Efficiency on the other hand is measured based on the average time taken for executing the process of matching and selection.

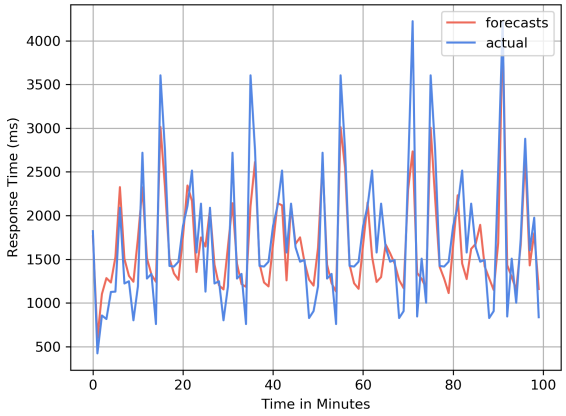


Fig. 4: LSTM Response Time Forecasts

Data Setup. We deployed the system (using standard service discovery mechanism) for a period of one week and we developed 10 different clients to send requests to various microservices based on a Poisson distribution with different mean values based on the given day of the week. To simulate a different workload between instances of the same microservice, a delay has been added at each request that depends on the type of instance and current time. This was done to emulate the real scenario where the incoming request rate can vary depending on the day and time. The response time of all the instances were continuously monitored and recorded. This was then used to create the LSTM based forecast model with a forecast horizon, $H = < 5, minutes >$.

Evaluation Candidates. We evaluated the approach by deploying the system integrated with each of the five different strategies for a period of 72 hours. These strategies form the evaluation candidates:

1. static-greedy (sta_gre): Instances are ranked based on the static response time registered by the instances during service registration. Selection is performed using a greedy strategy (select instance with the least response time). This strategy is the one used by standard service discovery mechanisms like Netflix Eureka, Apache Zookeeper, etc.
2. linereg-greedy (lin_gre): Prediction of instance response time is performed using linear regression and selection is performed using a greedy strategy. We use linear regression as it is a standard regression baseline. It was implemented using Python scikit-learn package.
3. timeseries-greedy (tim_gre): Prediction of instance response time is performed using time-series and selection using greedy.
4. linereg-Q-Learn (lin_Qle): Prediction of instance response time is performed using linear regression and selection using Q-learning.
5. timeseries-Q-Learn (tim_Qle): Our approach, which performs prediction of response time using time-series and selection using Q-learning.

5.3 Approach Effectiveness

Forecast Accuracy. The first part of evaluating the approach effectiveness was to measure the accuracy of the response-time forecasts produced by the LSTM models. The dataset was divided into training and testing set consisting of 8903 and 2206 samples respectively. We build the LSTM model with single hidden layer consisting of 60 neurons. This number was selected through experimentation. The model was fit in 250 epochs. We used Root Mean Square Error (RMSE) for evaluating the accuracy of LSTM models on the testing set, where RMSE for a dataset, O_{test} with n samples is given by the formula:

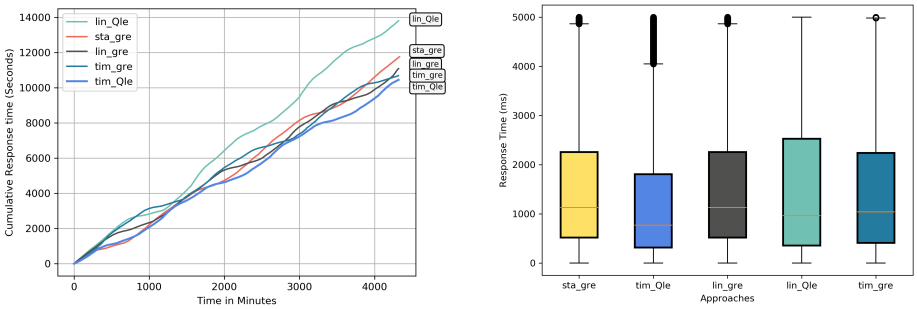
$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\Big(p_i - y_i\Big)^2} \tag{2}$$

where p_i represents the predicted value and y_i represents the actual value.

The calculation resulted in a value of 406.73 ms which means on average there is an error of 406.73 ms in prediction. Figure 4 shows the plot of actual versus forecasted values. We can see that the prediction made by the LSTM model is almost able to follow the curve of actual response time. This is due to the fact that LSTM, being a deep neural network possesses ability to identify any non-linear dependency that might exist between the different features such as the context attributes to generate accurate forecasts.

Average QoS Per Minute The second part of measuring the effectiveness was to calculate the metric, $\overline{QoS}(SB_t)$ as defined in Section 3. In order to accomplish this, we deployed our experiment by integrating the service discovery mechanism using each of the evaluation candidates for a period of 72 hours. The batch training phase was executed every 12 hours. The value of $\overline{QoS}(SB_t)$ was calculated for every minute and Figure 5a shows the plot of the cumulative $\overline{QoS}(SB_t)$ while using each of the approaches. We can clearly that, the cumulative $\overline{QoS}(SB_t)$ offered by the different approaches starts diverging marginally during the initial stages but as time progresses, the gap between *tim_Qle* and other approaches slowly starts increasing. In particular we can see that, the gap between *tim_Qle* and the traditional *sta_gre* keeps increasing steadily after 1900 minutes thereby resulting in the improvement of *tim_Qle* (10449 seconds) over *sta_gre* (11762 seconds) by 12% at the end of 72 hours. *lin_gre, tim_gre* on the other hand does perform better than *sta_gre* with *tim_gre* performing better than *lin_gre*. However, we can observe that the learning rate of these approaches is still less than *tim_Qle* and they are often inconsistent. This is more visible especially after 1900 minutes. This is due to the ability of Q-Learning to continuously improve with the help of feedbacks obtained for every selection performed. For Q-learning, the feedback for decision at time, t obtained via forecast at time, $t + 1$. Hence poor forecast accuracy implies that Q-Learning favors selection of wrong instances and due to this reason, *lin_Qle* performs the worst.

Average QoS Per Request In order to further evaluate the approach effectiveness in terms of the QoS offered by the instance, $\hat{s}$, we measured the



(a) Cumulative $\overline{QoS}(SB_t)$ per minute - lower is better

(b) Response time of $\hat{s}$ per request

Fig. 5: Effectiveness

response-time of $\hat{s}$ for every request made to $\hat{s}$. Figure 5b shows the box plot of the *response time* offered by $\hat{s}$ per request. The *tim_Qle* approach offers the least average response time of about 1236.23 ms which is $20\%, 19\%, 21\%$ and 16% better than the one offered by *sta_gre*, *lin_gre*, *lin_Qle* and *tim_gre* respectively. Also we can observe that most of the values fall between the range of 500 ms and 1800 ms which is much less compared to other approaches. Although, there are more outliers in the case of *tim_Qle* due to the initial phase of Q-learning where wrong selections are made, on average the response time perceived by s when using by *tim_Qle* is significantly lower compared to the others.

5.4 Approach Efficiency

The efficiency of the approach was evaluated by measuring the time taken by the whole service discovery process when integrated with our approach. The results show that on average the approach takes 0.10 seconds for performing the whole process. 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 service instances in each of the 5 services. LSTM training which happens every 12 hours takes around 125 minutes to complete but this does not impact the real-time process as only the trained models are used for service discovery.

5.5 Threats to Validity

Threats to *construct validity* is 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 context and quality change in order to emulate real-world scenarios as close as possible. In particular, the context setup such as

the server locations, request simulations, dynamic delays were all incorporated to achieve this goal. In order to improve the selection accuracy, the build phase is executed at regular intervals and by keeping the reward for incorrect selections to a high negative value, wrong decisions can be penalized.

Threats to *external validity* concern the generalizability and scalability of our approach. Although our approach has been applied on a system with 25 services, it uses techniques that can be generalized to more complex systems. Moreover, our approach can be gathered to large scale systems by optimizing Q-Learning (for instance by using Deep Q-Learning [9]) to solve the effectiveness and efficiency issue that might arise from the increased state, action space.

6 Conclusion

This paper has proposed a novel service discovery mechanism that takes into account the frequent variability of the consumers and providers contexts and QoS profile of services. Therefore, the overall goal of the approach is to select the matching service provider $\hat{s}$ so as to maximize the QoS perceived by the service consumer s with respect to its context, as well as with that of $\hat{s}$.

In this work, we have demonstrated how we can incorporate a combination of supervised and reinforcement learning into microservice architectures, especially for service discovery, without creating additional overheads as opposed to traditional ML integration (thanks to the use of build and real time phases). Moreover, our approach implementation also supports integration with existing server-side discovery patterns. Further, we believe that with the increasing usage of microservice-based architectural styles, this kind of approach, which effectively combines different ML techniques, can aid architects in guaranteeing the satisfaction of different non-functional requirements. We believe that this is just the beginning, and a lot more problems can be solved with the effective use of ML techniques.

To this extent, future work includes, but is not limited to: (i) extending the approach to large scale systems by considering multiple QoS parameters and the trade-offs among them; (ii) exploring the possibility of using transfer learning to make the approach more robust across different classes of systems.

References

1. Abdullah, M., Iqbal, W., Erradi, A.: Unsupervised learning approach for web application auto-decomposition into microservices. Journal of Systems and Software **151**, 243–257 (2019)
2. Andersson, J., Heberle, A., Kirchner, J., Lowe, W.: Service level achievements – distributed knowledge for optimal service selection. In: 2011 IEEE Ninth European Conference on Web Services. pp. 125–132 (2011)
3. Caporuscio, M., Grassi, V., Marzolla, M., Mirandola, R.: GoPrime: a fully decentralized middleware for utility-aware service assembly. IEEE Transactions on Software Engineering **42**(2), 136–152 (2016)

4. Chang, H., Kodialam, M., Lakshman, T., Mukherjee, S.: Microservice fingerprinting and classification using machine learning. In: 2019 IEEE 27th International Conference on Network Protocols (ICNP). pp. 1–11 (2019)
5. Chatfield, C.: Time-series forecasting. Chapman and Hall/CRC (2000)
6. D'Angelo, M., Caporuscio, M., Grassi, V., Mirandola, R.: Decentralized learning for self-adaptive qos-aware service assembly. Future Generation Computer Systems **108**, 210 – 227 (2020)
7. Di Francesco, P., Malavolta, I., Lago, P.: Research on architecting microservices: Trends, focus, and potential for industrial adoption. In: 2017 IEEE International Conference on Software Architecture (ICSA). pp. 21–30 (2017)
8. Graves, A.: Supervised sequence labelling. In: Supervised sequence labelling with recurrent neural networks, pp. 5–13. Springer (2012)
9. Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Horgan, D., Quan, J., Sendonaris, A., Osband, I., et al.: Deep q-learning from demonstrations. In: AAAI (2018)
10. Hochreiter, S., Bengio, Y., Frasconi, P., Schmidhuber, J., et al.: Gradient flow in recurrent nets: the difficulty of learning long-term dependencies (2001)
11. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation **9**(8), 1735–1780 (1997)
12. Houmani, Z., Balouek-Thomert, D., Caron, E., Parashar, M.: Enhancing microservices architectures using data-driven service discovery and QoS guarantees. In: 20th International Symposium on Cluster, Cloud and Internet Computing (2020)
13. Khaleq, A.A., Ra, I.: Intelligent autoscaling of microservices in the cloud for real-time applications. IEEE Access **9**, 35464–35476 (2021)
14. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
15. Lv, J., Wei, M., Yu, Y.: A container scheduling strategy based on machine learning in microservice architecture. In: 2019 IEEE International Conference on Services Computing (SCC). pp. 65–71 (2019)
16. Mokhtar, S.B., Preuveneers, D., Georgantas, N., Issarny, V., Berbers, Y.: EASY: Efficient semantic service discovery in pervasive computing environments with QoS and context support. J. Syst. Softw. **81**, 785–808 (2008)
17. Muccini, H., Vaidhyanathan, K.: PIE-ML: A Machine learning-driven Proactive Approach for Architecting Self-adaptive Energy Efficient IoT Systems. Tech. rep., University of L'Aquila, Italy (2020), https://tinyurl.com/y98weaat
18. Nayak, R., Tong, C.: Applications of data mining in web services. In: Web Information Systems – WISE 2004. pp. 199–205 (2004)
19. Richardson, C.: Microservices Patterns. Manning (2018)
20. Siami-Namini, S., Tavakoli, N., Namin, A.S.: A comparison of arima and lstm in forecasting time series. In: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA). pp. 1394–1401. IEEE (2018)
21. Stevenson, G., Ye, J., Dobson, S., Pianini, D., Montagna, S., Viroli, M.: Combining self-organisation, context-awareness and semantic reasoning: the case of resource discovery in opportunistic networks. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing. pp. 1369–1376. SAC '13 (2013)
22. Tsymbal, A.: The problem of concept drift: definitions and related work. Computer Science Department, Trinity College Dublin **106**(2), 58 (2004)
23. W3C: OWL-S: Semantic Markup for Web Services (2004)
24. Watkins, C.J., Dayan, P.: Q-learning. Machine learning **8**(3-4), 279–292 (1992)