A User-driven Adaptation Approach for Microservice-based IoT Applications

Martina De Sanctis
Gran Sasso Science Institute
L'Aquila, Italy
martina.desanctis@gssi.it

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

Karthik Vaidhyanathan
University of L'Aquila
L'Aquila, Italy
karthik.vaidhyanathan@univaq.it

ABSTRACT

Modern IoT-based applications are developed by using microservices implementing various functionalities. However, they still tend to be rigid from a user's perspective, i.e., the user typically adapts to how the software is designed. Conversely, we want the software and the IoT devices adapting to the user's goal and its dynamic nature, thus making the user as one of the key design element. For these reasons, we present MiLA4U, a multi-level self-adaptive approach that works at the three different user, microservices, and devices levels. Specifically, it *i*) makes use of a goal model defining run-time user goals that must be achieved without compromising the overall QoS, by adaptations towards the other levels. It therefore *ii*) continuously monitors the QoS of the microservices and IoT, and *iii*) leverages multiple algorithms for the QoS-aware dynamic selection, execution, and adaptation of microservices and IoT devices. MiLA4U is experimented on a real case study. Evaluation results show that it is able to satisfy the user goals while guaranteeing higher QoS on the microservices and IoT devices compared to standard baselines.

CCS CONCEPTS

• **Software and its engineering** → **Software architectures**; • **Networks** → *Cloud computing*; • **Applied computing** → **Service-oriented architectures**; • **Human-centered computing**;

KEYWORDS

User goals, microservices, IoT, multi-level adaptation

ACM Reference Format:
Martina De Sanctis, Henry Muccini, and Karthik Vaidhyanathan. 2021. A User-driven Adaptation Approach for Microservice-based IoT Applications. In *11th International Conference on the Internet of Things (IoT '21), November 8–12, 2021, St.Gallen, Switzerland.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3494322.3494329

1 INTRODUCTION

Modern digital ecosystems, built at the crossroads of the Internet of Things (IoT), the Internet of Services (IoS), and the Internet of People (IoP) [11, 12], bring new challenges. On one hand, *social challenges* are due to the changing user's preferences and needs. However, modern applications typically do not consider the dynamic nature of user's goals, thus affecting the user's engagement. According to Gil *et al.*, people cannot be excluded entirely from the adaptation loop, since the human attention is a critical factor for user participation and it may depend on different factors, e.g., user needs [19]. On the other hand, modern applications are built by combining a multitude of microservices and IoT devices running on edge, fog, and cloud computational infrastructures [17]. However, *technical challenges* arise when microservice-based solutions are applied to IoT systems. This is due to uncertainties faced by IoT devices in addition to those of microservices themselves (e.g., battery level, VMs/containers resource constraints) [22, 29].

In this context, where applications are built on top of various interchangeable services and IoT devices, the IoP asks for a shift towards a *user first* paradigm where users have the freedom to define their own personalized *goals*, which need to be accomplished by the applications with certain Quality-of-Service (QoS) guarantees. Being user needs possibly defined dynamically, *applications have to emerge at run-time*, so to satisfy users' goals and maximize the system's QoS.

Self-adaptation techniques can be exploited for managing run-time uncertainties [26, 31]. Specifically, a *chain of adaptations, both at the service and at the physical layer*, may be required to satisfy evolving users' goals, to deal with social challenges, while complying to system's requirements, to deal with technical challenges. In fact, although social and technical challenges do not coincide, they correlate to each other. For instance, in a trip planning scenario, the user may decide at run-time to use shared bike or bus based on weather conditions, seat availability, etc. Her decision may impact both the microservices to be called and certain physical devices to be invoked (e.g., IR counters in bus, RFID of smartbikes).

Different adaptation approaches try to cope with such challenges, such as [1, 15, 30], with the limitation of dealing with uncertainties at only one or a subset of the three levels we envisage, namely users, microservices and IoT devices. For this purpose, *we propose a user-driven multi-level self-adaptation approach*, namely MiLA4U. Given possibly thousands user goals dynamically derived by users' behaviors, a set of microservices (replicated to multiple instances) and IoT devices, MiLA4U selects the most appropriate microservice instances so to satisfy (or not) user goals while guaranteeing the best possible QoS of both microservices and IoT devices. In other words, we want application's workflow being dynamically defined, similarly to what the IFTTT app[1] does. It basically combines apps installed on the user device, driven by a user defined control-flow. Differently than IFTTT, we aim to combine available microservices

[1]https://ifttt.com

and devices, whose execution is then orchestrated by MiLA4U. The main design drivers that shape our approach are:

- A *multi-level self-adaptation process* relying on a corresponding architecture that extends existing microservice-based IoT reference architectures. Thus, MiLA4U can be built on top of existing architectures to bring self-adaptation.
- A *user goal model* supporting the system to transparently derive *user's goal* at run-time based on the user's selection of functionalities among those offered by the application. Then, the application's workflow is automatically derived by the user's goal to reflect the selected functionalities, their execution order, and QoS guarantees.
- *Self-adaptation algorithms* run by the architectural components and handling adaptation concerns arising from different entities through different levels, namely user, microservices and devices levels.

We evaluate our approach on a real microservice-based IoT application. To show the effectiveness of MiLA4U in performing multi-level adaptations, we compared it with three baseline approaches that are representative of a possible benchmark. The results of our experiments show that our approach exhibits remarkable improvement compared to baselines in achieving users goals while maintaining the overall QoS requirements of the system. Moreover, to the best of our knowledge there are no available self-adaptive exemplars/benchmarks that combine users, microservices and IoT. We believe that this work can serve as a potential benchmark for future self-adaptive research of this kind.

2 MOTIVATING CASE STUDY

We motivate our work with a microservice-based IoT system[2] developed for the street science fair of the city of L'Aquila. In this event, the research community and public are brought together to share a combination of entertainment and information. It takes place at multiple venues in the city center and witnesses around 25,000 participants every year. To better manage the crowd and for improving the quality of the visiting experience, we developed the NdR mobile app[2], providing different features such as real-time parking lot availability, venue booking, weather information, parking lot recommendation. Figure 1 shows the high level architecture of the NdR application. The information on crowd and car movement in the

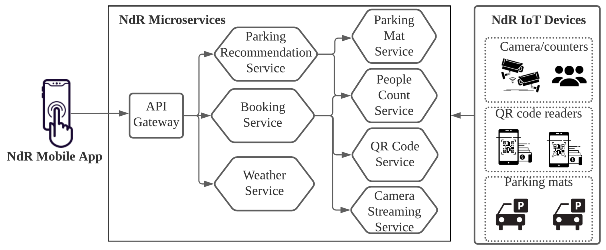


Figure 1: High-level architecture of the NdR Application

venues and parking lots are captured using different sensors such as people counters, parking mats, cameras and QR code readers. The data provided by the sensors are further exploited by various microservices (e.g., parking mat service handles the data coming from parking mats) to accomplish the different functionalities of the NdR mobile app. Eventually, to accomplish any functionality, NdR mobile app, like any modern applications, entail a predefined static application flow, such as check availability → book venues → get parking recommendations → check weather. However, in dynamic contexts, considering the heterogeneity and multitude of involved actors, such as *users*, *microservices* and *IoT devices*, different user's needs and preferences can arise and impact multiple levels. An example is given in the following:
i) **application:** To select between indoor/outdoor event, a user might want to first know the weather conditions and then, check for parking lots. Instead of forcing the user to follow a predefined flow, an adaptation can be to *dynamically* combine the weather and parking microservices to accomplish the *user's goal.*
ii) **microservice:** Further, the system should be able to combine the instances of weather and parking microservices that offer the least response time to ensure that the overall response time perceived by the user is minimized.
iii) **device:** In the above scenario, the parking microservice requires data only from the parking mats. Hence, the data transfer frequency of the other IoT devices can be reduced to save more power.

Hence, the system should have the ability to dynamically adapt based on the user's goals. Moreover, if we consider, for instance, *response time* and *devices energy* as quality attributes, then we derive the following objective: given i) eRT_{max}, the maximum expected response time for a given user goal, and ii) E_{max}, the maximum energy that can be consumed by the IoT devices for an execution period of time τ, at any point of time τ, the overall objective of the system is maximizing the following utility function, U_τ. It enables us to quantify the satisfaction of eRT_{max} and E_{max}:

$$U_\tau = x_u \cdot Q_\tau + x_e \cdot E_\tau, \text{ with } Q_\tau = \sum_{i=1}^{n} q_i$$

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

$$q_i = \begin{cases} eRT_{max} - rt(i) & \text{if } rt(i) < eRT_{max} \\ (eRT_{max} - rt(i)) \cdot p_{rt} & \text{otherwise} \end{cases}$$

where, $x_u, x_e \in \mathbb{R}^+$ are weights that capture the priority of user goal completion time and energy savings, respectively. Q_τ captures the sum of user goal response time gain (this also inherently captures the response time of the involved microservices) for n goals served within a period τ. E_τ and q_i are piece-wise functions that capture the energy savings and response time, respectively, where $p_{ev}, p_{rt} \in \mathbb{R}^+$ represent penalties for the violations of energy and response time thresholds, whereas e_τ represents the total energy consumed by the IoT devices for the τ-period, and $rt(i)$ represents response time of i^{th} goal.

3 MILA4U ADAPTATION APPROACH

The proposed user-driven multi-level adaptation approach relies on a corresponding architecture[3], which was outlined in a preliminary and abstract way in our previous work [14]. It is built on top

[2] http://streetscience.it

[3] The graphical representation of the architecture can be found in our online material at https://github.com/karthikv1392/MiLA4U/wiki/MiLA4U-Architecture

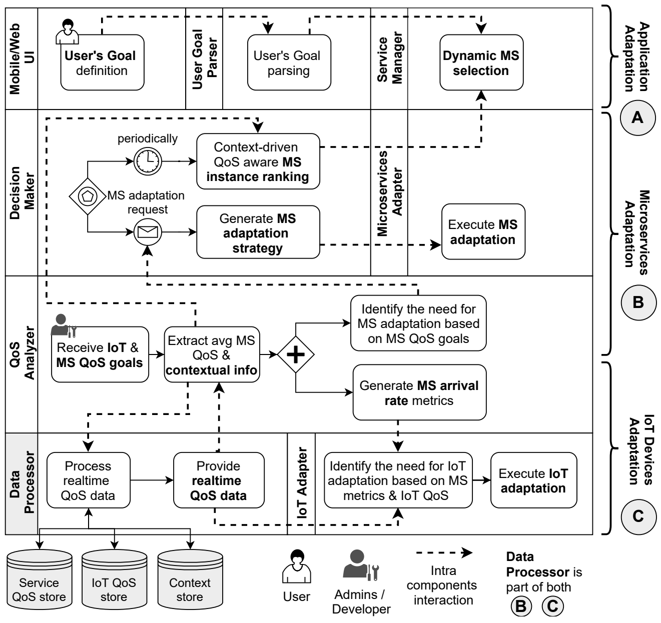

Mobile/Web UI
User Goal Parser
Service Manager
User's Goal definition
User's Goal parsing
Dynamic MS selection
Application Adaptation
A
Decision Maker
periodically
Context-driven QoS aware MS instance ranking
MS adaptation request
Generate MS adaptation strategy
Microservices Adapter
Execute MS adaptation
Microservices Adaptation
B
QoS Analyzer
Receive IoT & MS QoS goals
Extract avg MS QoS & contextual info
Identify the need for MS adaptation based on MS QoS goals
Generate MS arrival rate metrics
IoT Devices Adaptation
Data Processor
Process realtime QoS data
Provide realtime QoS data
IoT Adapter
Identify the need for IoT adaptation based on MS metrics & IoT QoS
Execute IoT adaptation
C
Service QoS store
IoT QoS store
Context store
User
Admins / Developer
Intra components interaction
Data Processor is part of both B C

Figure 2: Overall multi-level adaptation process of MiLA4U.

of traditional architecture for microservice based IoT system [17], with additional components we defined to support adaptation. This guarantees a fully decoupled extension and is done to promote the *generalizability* and *reusability* [25] of MiLA4U. In this work, we provide the adaptation methods behind MiLA4U used for effectively managing the adaptation concerns arising from applications, microservices and IoT devices. Fig. 2 reports the overall multi-level adaptation process realized by the newly defined components, and the interactions among them, enabling the adaptation to be performed in cooperation.

At the application level (Fig. 2 Ⓐ), the adaptation is triggered with the selection of desired functionalities by the user with the help of a Mobile/Web UI. This is further translated into a user goal (cf. Section 3.1) by a User Goal Parser which then identifies the set of microservices and tags the desired QoS required to accomplish the goal. This information is further processed by the Service Manager which identifies the optimal set of microservice instances, w.r.t. to QoS requirements, to accomplish the user goal (cf. Section 3.1). The identification of microservice instances at application level is supported by microservice level adaptation (Fig. 2 Ⓑ). It makes use of a QoS Analyzer, responsible for analyzing the QoS and context information (e.g., location) of the microservices and further identify the need for adaptation (e.g., QoS violation) based on the data obtained from the Data Processor. This information is then passed to a Decision Maker which periodically ranks the microservices based on QoS and context information to aid application level adaptation (cf. Section 3.2). Further, based on adaptation request from QoS Analyzer, it generates microservice adaptation strategy (e.g., add/remove microservices) and execute them by using Microservices Adapter. The adaptation at application and microservice level together support adaptation at device level (Fig. 2 Ⓒ) such that the QoS of the IoT is optimal. This is achieved with the help of an IoT Adapter which uses the real-time QoS data available from the Data Processor and the data from QoS Analyzer on the request arrival rate of microservices to adapt the IoT devices (e.g., reconfigure devices) (cf. Section 3.3).

In the following, driven by Fig. 2, we introduce the application level adaptation, then how it works in conjunction with the adaptation in microservice level. Lastly, we see how the former two can support the device level adaptation.

3.1 Application Level Adaptation

The application level adaptation is supported by i) a *goal model*, (implemented by the User Goal Parser), and ii) the *service management* algorithm.

Goal Model. The presented goal model supports applications

Table 1: Goal model syntax.

$\mathsf{G}_{\mathsf{User}}$	::=	F^+
F	::=	$\mathsf{f}_{?[\mathsf{QoS}]}$ $\vert$$[\mathsf{F}$ **and** $\mathsf{F}]_{?[\mathsf{QoS}]}$ $\vert$$[\mathsf{F}$ **or** $\mathsf{F}]_{?[\mathsf{QoS}]}$ $\vert$ **one_of** $[\mathsf{f}_1 \ldots \mathsf{f}_n]_{?[\mathsf{QoS}]}$ $\vert$**seq** $[\mathsf{f}_1, \ldots, \mathsf{f}_n]_{?[\mathsf{QoS}]}$ $\vert$ $\top$ $\vert$$\perp$
QoS	::=	'rt:' RT
RT	::=	$[\mathsf{Threshold}_{\mathsf{min}}$, $\mathsf{Threshold}_{\mathsf{max}}]$
Threshold	::=	$\mathsf{eRT}_{\mathsf{min}}$ $\vert$$\mathsf{eRT}_{\mathsf{avg}}$ $\vert$$\mathsf{eRT}_{\mathsf{max}}$
eRT	::=	$\mathsf{NUMBER}_{\mathsf{rt}}$ sec $\vert$$\mathsf{NUMBER}_{\mathsf{rt}}$ ms
$\mathsf{NUMBER}_{\mathsf{rt}}$	::=	$n, n \in \mathbb{N}$
f	::=	'ticket_availability' $\vert$ 'weather_checking' $\vert$ 'event_booking' $\vert$ 'parking_recommendation'

to derive *personal objectives at users level*. Specifically, we aim to allow application's workflow being dynamically defined, within the boundaries of the functionalities it exhibits. For this reason, the proposed goal model supports the definition of *procedural* goals representing an abstraction of the application's workflow directly coupled with the application functionalities.

Table 1 reports the syntax of our goal model. A goal, namely $\mathsf{G}_{\mathsf{User}}$, defines as objective a desirable (sub)set of functionalities, namely F^+, as per user preferences, selected from the overall set of currently available functionalities, namely f, offered by the application. Moreover, in order to derive F, our goal model foresees the use of *control-flow* constructs, namely **and**, **or**, **one_of** and **seq**, which can be specified to recursively combine functionalities, thus to create application workflows of different complexity. Goals of type **and** and **or** *succeed/fail* as defined by the logical operators. The **one_of** operator is equivalent to the **or** and specifies the user's desire of indistinctly accomplishing one of the listed functionalities. It can be used for long sequences of alternative functionalities, since it can be fastest parsed. An trivial example is given in (1):

$$\mathsf{G}_{\mathsf{User}} ::= \textbf{one\_of}\ [\text{'weather\_checking'}, \text{'parking\_recommendation'}] \quad (1)$$

It might be used to decide about the participation to an event of the street science fair. Indeed, bad weather conditions or parking unavailability may impact the user's decision. A goal of type **seq**, instead, specifies a set of functionalities to be executed in the *given order*, e.g., in (2).

$$\mathsf{G}_{\mathsf{User}} ::= \textbf{seq}\ [\text{'parking\_recommendation'}, \text{'event\_booking'}] \quad (2)$$

A hierarchical relation (or inhibition relation) among the available functionalities (see (3)) can be defined, to establish any precedence among them.

$$\text{‘ticket\_availability’} > ((\text{‘event\_booking’} \geq \text{‘parking\_recommendation’}) \parallel \text{‘weather\_checking’}) \quad (3)$$

Each functionality f can be optionally labeled with the desired QoS that the (dynamically selected) microservice implementing it is expected to show, via the ?[QoS] construct. Also combined goals might be optionally labeled with the desired QoS, i.e., the QoS expected by the execution of the dynamically defined application workflow. Currently, in our work, the QoS specification refers to the response time, RT, foreseen for the generated application workflow, and expressed as a range between two Thresholds values, to deal with run-time uncertainties. Thresholds values might be determined by admins/developers as the *expected* response time, eRT, from a user perspective. An example is given in (4):

$$G_{\text{User}} ::= [\text{‘ticket\_availability’ } \textbf{and}\text{‘event\_booking’}]_{\text{rt}:[3sec,5sec]} \quad (4)$$

Eventually, it is easy to notice that the presented goal model is open to extensions, as we plan for our future work, with further control-flow constructs and QoS parameters. Then, the more functionalities are offered by the application, the more diverse goals can be defined with the goal model.

Service Manager. Once the goal has been derived, it is parsed by the User Goal Parser, by means of regular expressions, and it results in a tree-like structure. The parsed goal is then passed to the Service Manager that is in charge of selecting the optimal set of microservices to accomplish the user's goal (cf. Fig. 2 Ⓐ). To this aim, it employs **Algorithm 1** that takes as input two parameters:

i) The *goal* that provides the main control-flow constructs. For instance, for a goal of type G_{User} ::= ‘parking_recommendation’ **and** ‘event_booking’, the construct **and** becomes the root while functionalities its child nodes.

ii) F_{map}, i.e., the mapping between the functionalities available in the system, (and in the goal model), and the corresponding microservices offering them. The algorithm first checks if the *goal* is a functionality. If true, it executes the functionality by invoking the corresponding microservice identified by the *Instance_Selector* function using F_{map} (lines **2-3**). If false, it checks the type of the control-flow construct to determine the execution flow. If the construct is **seq**, then the functionalities expressed are sequentially executed (lines **4-6**). In case of a construct of type **one_of**, the execution breaks as soon as one of the functionalities is successfully performed (lines **7-11**). For handling the **and** control-flow construct, the algorithm performs a recursive operation by taking as input either sides of the goal, ie., $goal.left$ and $goal.right$, in a parallel and asynchronous manner (lines **12-15**). The same happens for goals of type **or**, with the only difference that only one of the two goal's sub-trees is taken. To perform each functionalities, the Service Manager first identifies the best instance of a microservice to be invoked. This is achieved by sending the service name to the *Instance_Selector*. It makes use of a rank map that, for each microservice $m \in M$, contains the set of location zones L (e.g., US, Europe), where each location zone, $l \in L$, consists of a set of available microservices instances N ranked based on their QoS.

Algorithm 1 QoS-aware Service Management

```
1:  procedure SERVICE_MANAGER($goal$,$F_{map}$)
2:      if $goal$ is in $F_{map}$ then                          ▷ leaf node
3:          Execute(Instance_Selector($F_{map}[goal]$))
4:      else if $goal$ = seq then
5:          for $goal$ in $goal.child$ do
6:              Execute(Instance_Selector($F_{map}[goal]$))
7:      else if $goal$ = one_of then
8:          for $goal$ in $goal.child$ do
9:              s ← Execute(Instance_Selector($F_{map}[goal]$))
10:             if s = success then
11:                 break
12:     else if $goal$ = and then
13:         do in parallel
14:         SERVICE_MANAGER($goal.left$,$F_{map}$)
15:         SERVICE_MANAGER($goal.right$,$F_{map}$)
16:     else if $goal$ = or then
17:         SERVICE_MANAGER($goal.left$, $F_{map}$) OR
18:         SERVICE_MANAGER($goal.right$, $F_{map}$)
19:     return 1
```

This ranking is then leveraged by the *Instance_Selector* to select the optimal service instance $n \in N$, for a given service name m in a location l, based on the location of the Service Manager component itself. This is to ensure minimal latency of the network transfer/communication. The rank list is updated periodically by the Decision Maker (**Algorithm 2**).

We here highlight that each microservice typically implements a single functionality. The set of functionalities exposed by the application feed our goal model (f rule in Table 1). This allows MiLA4U to perform the *one-to-one type-to-instance adaptation*, iteratively refining the goal until leaf-level functionalities, thus promoting generalizability and reusability [25].

3.2 Microservice Level Adaptation

The core logic of the microservices adaptation is handled by the Decision Maker (cf. Fig. 2 Ⓑ), which uses the QoS and context data of microservices to dynamically rank them, by using **Algorithm 2**. The algorithm takes as input the set of available microservices, S, the average QoS of every microservice's instance in the system in the form of a QoS map, Q_{map}, and the context set C, consisting of the context information for all the microservices, i.e., the location. The algorithm then initializes a multi-dimensional rank map, R. Then, for each microservice s in the set of microservices, the algorithm determines the locations of the different instances

Algorithm 2 Context driven QoS-aware Microservice Ranking

```
1:  procedure RANK_GENERATOR($S$,$Q_{map}$,$C$)
2:      Initialize rank map, $R$
3:      for $s$ in $S$ do
4:          for $l$ in $C[s.locations]$ do
5:              $qos\_list \leftarrow []$
6:              for $i$ in $l.instances$ do
7:                  if $i$ is active then
8:                      $qos\_list$.add($Q_{map}[i]$)
9:              $R[s][l] \leftarrow qos\_list$.sort()
10:     return $R$
```

of s (lines **3-4**). For every microservice instance i close to a location l, the algorithm checks if the instance is active and if true, it identifies the average QoS of the instance from Q_{map} (lines **6-8**). This information is appended to a list, namely *qos_list*. This list is sorted at the level of service and location and added to the multi-dimensional rank map that provides the QoS-based sorted list of microservices instances, for each given couple of microservice and location. The generated rank map R is periodically sent to the Service Manager, which exploits it when executing Algorithm 1. Further, the Decision Maker also checks the average QoS offered by every microservice to automatically add/remove instances in order to maintain an acceptable QoS (response time, utilization, etc).

3.3 Device Level Adaptation

For handling the adaptation at the IoT devices level, the approach makes use of the IoT Adapter (cf. Fig. 2 Ⓒ). Such adaptation is determined by three aspects: i) the QoS of the IoT devices; ii) metrics related to those microservices that are associated either directly or indirectly with the IoT devices, in particular the microservices request arrival rate, and iii) the operational modes of the IoT devices, namely *normal mode* (sensors gather data at standard frequency rate) and *critical mode* (sensors gather data at a higher frequency rate e.g., in critical situation). The IoT Adapter periodically receives the information on the arrival rate metrics for the different microservices from the QoS Analyzer. It further uses this information with the IoT devices QoS and operational mode to generate an adaptation strategy. To this aim, it exploits **Algorithm 3**. The algorithm, in particular, enacts adaptation on the devices to reduce the overall energy consumption. It takes as input E_τ, the threshold limit for the total energy consumption in a given time interval, τ. This can be defined by the stakeholder, such as an IoT expert, based on the use case. In this work we make use of the adaptation strategy which dynamically controls device communication frequency as a mechanism to reduce energy consumption. Other adaptation strategies could be employed to satisfy different QoS constraints based on application domain. To this end, the algorithm takes as input E_C, the total energy consumed for the given interval. Sensors' reduction frequency, $R_{F_{map}}$, defines the maximum value of data transfer frequency that can be reduced for sensors in normal mode. These values can again be defined by an IoT expert. The algorithm then takes the arrival rate per microservices, $A_{R_{map}}$, containing information on the number of request arrivals per microservice for a given τ, while $A_{T_{map}}$ denotes the required arrival rate thresholds of the different microservices, below which service accuracy issues are managed. For instance, if the arrival rate threshold for the booking microservice is 100 requests, and the average arrival rate is above it, then the data transfer from sensors cannot be reduced, as it will affect the service accuracy. Eventually, the algorithm takes as input M_{map}, which contains the list of sensors that are involved directly or indirectly with every microservice. For instance, a *people_counter* might be sending real-time data and an *event booking* microservice might be leveraging this data to determine booking availability. This implies a dependency between *people_counter* and *event booking* microservice, which is stored in M_{map}. Based on these inputs, the algorithm first checks if E_C is above E_τ in which case it loops over $A_{R_{map}}$ to determine, for each microservice, if the arrival rate is less than the threshold (lines **2-4**). If true, it identifies the sensors corresponding to the microservice by using M_{map} and their frequencies are adjusted based on the information stored in $R_{F_{map}}$ (lines **5-7**). However, the frequency of any sensor is reduced only if they are not in critical mode (line **6**). Once the arrival rate is above threshold, it resets the sensors frequency to the original values (lines **8-11**).

Algorithm 3 Edge Device Adaptation

```
1: procedure IoT ADAPTER($E_\tau$, $E_C$, $R_{F_{map}}$, $A_{R_{map}}$, $A_{T_{map}}$, $M_{map}$)
2:     if $E_C > E_\tau$ then
3:         for $m$ in $A_{R_{map}}$ do                    ▷ $m$ stands for microservice
4:             if $A_{R_{map}}[m] < A_{T_{map}}[m]$ then
5:                 for $sensor$ in $M_{map}[m]$ do
6:                     if $sensor.mode != $ "$critical$" then
7:                         $sensor.freq \leftarrow sensor.freq + R_{F_{map}}[sensor]$
8:         else
9:             if $A_{R_{map}}[m] \geq A_{T_{map}}[m]$ then
10:                for $sensor$ in $M_{map}[m]$ do
11:                    reset $sensor.freq$
12:    return 1
```

4 EXPERIMENTS

The main objective of the experiment and evaluation is to compute the *effectiveness* and *efficiency* of the approach. We aim to answer the following research questions:

RQ1 How does the overall QoS of microservice-based IoT system using MiLA4Ucompare to standard baselines?

- **RQ1.1** How does the user goal satisfaction using MiLA4Ucompare to the baselines and what is the impact of user goal type on the response time?
- **RQ1.2** What is the impact of MiLA4Uon the energy consumed by IoT devices compared to the baselines?

RQ2 What is the computation overhead of MiLA4Ucompared to the baselines?

Experiment Setup. We consider the NdR case study (see Section 2) for evaluating our approach[4]. It consists of 7 microservices that provides different functionalities (e.g., venue booking, parking lot booking, etc.) based on the data gathered from 14 different sensors (e.g., parking mats, QR code reader, etc.). Each of the microservices were replicated to 4 instances making a system of 28 microservices. For implementing the IoT devices, we made use of CupCarbon [6], a state of the art smart city IoT simulator, especially for energy simulation [8]. The microservices were deployed with docker on two VM instances in Google Cloud, in two different geographical zones. This was done to capture the different context dimensions as well as QoS fluctuations that may arise from the type of CPU, geographical zone, memory, etc. We used Python for implementing the different algorithms. For our experiment, we used the goal model to generate a goal file consisting of 20K goals (considering different user goal combinations, including repetitions). Our experiments are based on real time execution of the system for 5 hours (peak load scenario). To simulate the user behavior, we developed a Python script which uses the *asyncio* library to emulate concurrent user sessions. The number of user requests per second were based on real world benchmark trace [3]. The experiment was evaluated based

[4] Details, source code and dataset: https://github.com/karthikv1392/MiLA4U

Table 2: 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
E_{max}	Maximum energy	1.45 joules
eRT_{max}	Maximum expected response time	4 secs
x_u	Weights on user goal satisfaction	5
x_e	Weights on energy savings	4

on the research questions on the evaluation candidates described in Table 3. To measure the effectiveness of the approach, we make use of the Utility Score (U), as defined in Section 2, using normalized values for Q_τ and e_τ for every τ-period. Parameters values are reported in Table 2. We assign a slightly higher weight to the x_u than x_e because, although saving energy is a priority, we do not want to do it at the expense of the user's perceived response time.

Results for RQ1. Fig. 3a shows the *cumulative utility score* (based on U) of all the candidate approaches. We can observe that the approaches with dynamic workflow (*DN* and *DA*) offer a much higher utility (close to 75%) compared to approaches with static workflow. This, in turn, validates the fact that the increased flexibility provided to the users together with effective algorithms can ensure higher QoS of the overall system. Moreover, *DA* (which also uses the algorithms 2 and 3) offers a much better utility (1538), about 34% higher than *DN* (1145). This clearly demonstrates the effectiveness of the adaptation algorithms used by *DA* to satisfy the different QoS goals. In Fig. 3a, it is also visible that, as time progresses, the gap between the utility offered by *DA* w.r.t. *DN* increases gradually. Moreover, as we can observe after 200 minutes, while the *SA* and *SN* follow a constant trend, *DA* and *DN* are able to offer higher utility. This is due to the increase in user goal arrival rate that the dynamic workflow based approaches, especially DA, are able to handle more effectively.

Results for RQ1.1. Fig. 3b shows the box plot of the goal utility (Q_τ in U_τ) per minute attained by each of the approaches. The results clearly demonstrate the effectiveness of dynamic workflow based approaches to satisfy user goals compared to that of their static counterparts. *DA* offers the best goal utility with an average of 0.67. Moreover, the average goal utility of *DA* is 31% higher than that offered by *DN* and about 70% higher than that offered by, *SN* and *SA*. In addition, we observed that the number of user goals that exceeded eRT_{max} were 50% less in *DA* compared to *DN*, thus providing higher user goal satisfaction while using *DA*. Considering the impact of user goal type on the response time, Fig. 3c, represents the bar plot of the average response time offered by the two dynamic approaches for different goal types[5]. As expected sequential goals (*seq*), on average has the biggest impact on the response time. Even in that case, *DA* is able to reduce response time by almost 15%. Goals of type *and* has the second biggest impact on the response time and, *DA* is able to reduce the response time by 20%. Similar, is the case with other goal types. These result clearly demonstrates the ability of *DA* in guaranteeing higher user goal satisfaction.

Results for RQ1.2. As we can observe in Fig. 3d, *DA* could save the maximum amount of energy compared to other approaches. The total energy savings of *DA* was around 20% higher than *SN*, *DN* and 9% higher than *SA*. Moreover, besides *SA* uses the same adaptation algorithm as *DA*, the energy saved is higher in the case of *DA*. This is because since *SA* requires the user to complete a predefined flow to achieve a specific functionality, the arrival rate on almost all microservices is equally distributed as opposed to the dynamic workflow-based approaches. This provides the IoT devices with more level of flexibility to regulate the data transfer frequency. Hence, the results depict that *DA* offers maximum energy savings for the IoT devices compared to the candidate approaches.

Results for RQ2. To evaluate the computation overhead, we clocked the time required to execute different algorithms. On average, the service selection process in the case of *DA* and *DN* (algorithm 1) took around 1.2 seconds to complete parsing, selection and execution of user's goal, while it took, on average, around 3 seconds in *SA* and *SN*. The ranking algorithm itself just took around 30 microseconds on average while the complete ranking process took around 1.7 seconds, on average. This involves querying the recent QoS values, generating the rank map and transferring the ranks to the Service Manager. However, this is performed as a batch process and not in real-time. The edge adaptation process (algorithm 3) took around 0.5 seconds to execute the adaptation process, in which 0.3 seconds were taken by the network transfer of adaptation decision.

The results of our evaluation in RQ1 indicate the effectiveness of our approach (*DA*) in improving the overall QoS of a microservice based IoT system. Further, RQ1.1 and RQ1.2 demonstrate the effectiveness of our approach in guaranteeing higher user goal satisfaction and in reducing energy consumption of the IoT devices respectively. Moreover, the results of RQ2 proves that our approach is able to guarantee effectiveness without compromising on the overall efficiency.

Threats to Validity. Threats to *construct validity* relates to the IoT simulation configurations. To this end, we used configurations based on the real IoT deployment of the NdR system. Threats to *external validity* concern the generalizability and scalability of our approach. Although our approach has been applied on a system with 28 microservices and 14 sensors, it uses techniques that can be generalized and extended to more complex systems, as long as there are methods to obtain QoS information of microservices and IoT devices.

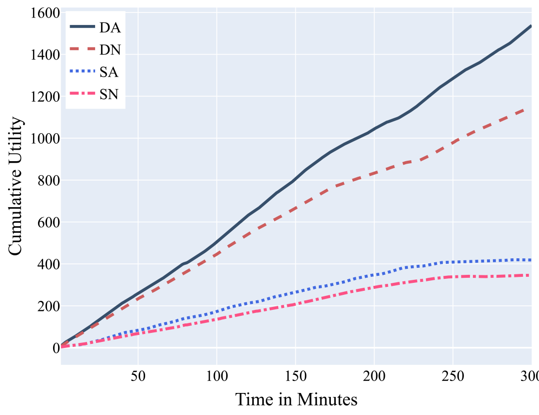
5 RELATED WORK

In self-adaptive systems, goals are used to express the desired runtime behaviour of systems execution (e.g., in [13]). However, in recent years, the pervasive impact that mobile applications are having on society, pushed forward the need for defining *personal objectives at users level*, as done in our work. To this aim, Qian *et al.* [28] proposed *MobiGoal*, a framework exploiting a runtime goal model supporting the adaptive scheduling of user's goals and execution of tasks. Similarly to *MobiGoal*, our goal model allows the definition and fulfillment of personal goals in a customizable and adaptive way. However, in *MobiGoal*, a goal is defined at design-time and it drives the design of the application, whose runtime execution supports the goal's fulfillment. Differently, in our approach, users goals emerge at runtime and users are not aware of the goal model, its syntax and semantic. They simply select functionalities and express preferences in their applications (e.g., by means check lists).

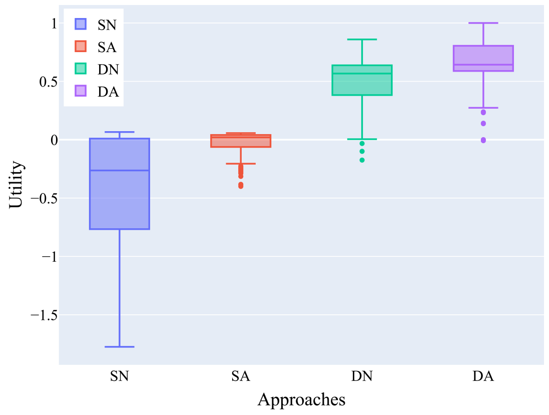
[5] *SA* and *SN* use a static workflow and do not differentiate among user goal types.

Table 3: Evaluation Candidates for the Experiment

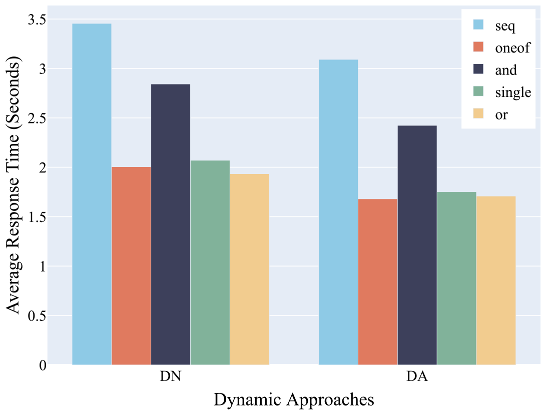
Name and Description
Static workflow-no-adaptation (SN): It emulates the behavior of NdR application without using goals. It uses a static application workflow, where each user request has to follow the sequence made by ticket_availability → event_booking → parking_recommendation → weather_checking. To ensure fairness among the evaluation candidates, the same goal file is used, however, to accomplish any goal, the user has to execute the flow until the goal is reached.
Static workflow-adaptation (SA): It is similar to the previous candidate but it uses the ranking and Edge adaptation algorithms (**algorithms 2** and **3**) for the QoS aware adaptation of the microservices (although there is no selection due to the absence of dynamic goals) and IoT devices.
Dynamic workflow and selection (DN): It emulates the behavior of NdR application allowing users to dynamically define goals. It implements the *User Goal Parser* and the service management algorithm (**algorithm 1**) but it does not consider the IoT devices and microservices level adaptation (It randomly selects a microservice instance instead of using algorithm 2).
MiLA4U (DA): It emulates the approach described in this paper which provides users with the ability to dynamically define goals and further uses the service management, ranking and edge adaptation (**algorithms 1, 2** and **3**) to address adaptation concerns of user, microservices and IoT devices.



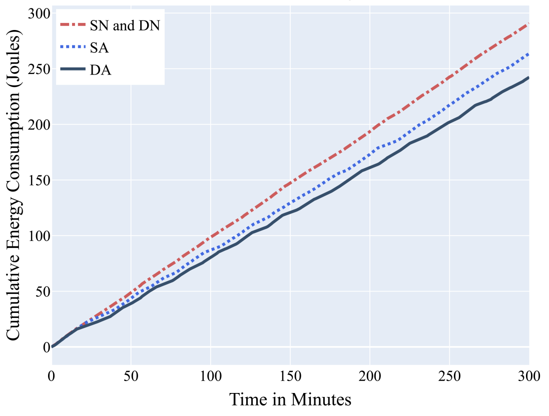
(a) Cumulative Utility



(b) Utility per user goal



(c) Response Time per User Goal



(d) Cumulative Energy Consumed

Figure 3: Results of the performed experiment

From this information the application's workflow is dynamically derived. Alkhabbas *et al.* also make use of goal-driven approaches for the realization of emergent configurations in the IoT [1] and their deployment [2]. However, a goal model and microservice-based adaptation are not provided. To show the wide range of application of different low-level goal models, Kalia *et al.* [23] exploit goals to transform people-driven processes to chatbot services. Noura *et al.* [27], instead, propose an approach for understanding end-users goals from voice inputs in smart homes. Lastly, our approach is comparable to services mashups, namely value-added services build from existing services. In [4] the authors propose an approach that combines the modeling power of software product line feature models with AI planning techniques to perform service mashups composition. A responsive decentralized composition of service mashups for the IoT, instead, is proposed in [10]. However, in [4] the IoT is neglected, whereas the work in [10] does not consider the runtime adaptation of the developed IoT applications.

Works providing *adaptation in microservice-based environments* have also been presented. Florio *et al.* [18] provided an agent-based approach for handling adaptation through a decentralized MAPE loop. A reference architecture for self-adaptation in microservices using an adaptation registry was proposed by Baylov *et al.* [5]. Khazaei *et al.* [24], instead, introduced the idea of using self-adaptation as a service for managing adaptations concerns in microservice-based architectures. QoS has become a major concern in service-based systems from a while [7]. However, only a few works addressing QoS-driven adaptation exist in the context of microservices-based systems (e.g., [20, 21]).

Different approaches for *self-adaptation in IoT* have been proposed [1, 31]. Model-Driven Engineering based approaches were presented, for instance in [9] where the adaptation is carried out using the concept of models@run.time and MAPE-K loop. An agent-based framework for performing self-adaptation for IoT applications was proposed in [16]. De Sanctis *et al.* [15] presented an approach for the dynamic user-driven QoS-based formation of IoT architectures, by leveraging AI planning to derive operational plans. However, while these works focus on the optimal selection of devices, microservices are neglected.

Lastly, although several multi-level (and multi-layer) adaptation approaches exist (e.g., [32]), they refer to different levels w.r.t. our approach, given the specific context of the work. To the best of our knowledge, a full-fledged self-adaptation approach that combines runtime user's defined goals, microservices and IoT, as MiLA4U, does not exist. MiLA4Ufurther considers adaptation challenges that emerge when IoT devices and microservices are used in tandem.

6 CONCLUSION AND FUTURE WORK

This paper shows how a user-driven multi-level approach can be used to effectively handle adaptation concerns arising from different levels in an microservice-based IoT system. The results of our evaluation on a real case study are promising. They also indicate how an adaptation centered around users can provide them with a greater flexibility and how this can support better management of QoS at multiple levels. With regard to the future work, i) we would like to focus on the scalability aspect of MiLA4U, by applying it to large scale systems with larger set of functionalities, IoT devices with multiple QoS dimensions, e.g., utilization, data traffic (IoT); ii) we plan to extend MiLA4U to support adaptation based on user and functionality types, i.e., users with high-priority or critical functionalities may require higher QoS; iii) we plan to leverage machine learning techniques to perform proactive adaptations, e.g., for service ranking and IoT based adaptation.

ACKNOWLEDGMENTS

This work was partially supported by the PON Cultural Heritage (AIM1880573), Project at GSSI, and the Italian National Funded project, VASARI PON (ARS01_00456 - co-funded by MIUR, with the support of EU) at the University of L'Aquila.

REFERENCES

[1] Fahed Alkhabbas, Martina De Sanctis, Romina Spalazzese, Antonio Bucchiarone, Paul Davidsson, and Annapaola Marconi. 2018. Enacting Emergent Configurations in the IoT Through Domain Objects. In *Service-Oriented Computing - 16th International Conference, ICSOC 2018, Proceedings*. Springer, 279–294.

[2] Fahed Alkhabbas, Ilir Murturi, Romina Spalazzese, Paul Davidsson, and Schahram Dustdar. 2020. A Goal-Driven Approach for Deploying Self-Adaptive IoT Systems. In *IEEE International Conference on Software Architecture, ICSA*. IEEE, 146–156.

[3] Martin Arlitt and Tai Jin. 2000. A workload characterization study of the 1998 world cup web site. *IEEE network* 14, 3 (2000), 30–37.

[4] Mahdi Bashari, Ebrahim Bagheri, and Weichang Du. 2016. Automated Composition of Service Mashups Through Software Product Line Engineering. In *Software Reuse: Bridging with Social-Awareness - 15th International Conference, ICSR*. 20–38.

[5] Krasimir Baylov and Aleksandar Dimov. 2018. *Reference Architecture for Self-adaptive Microservice Systems*. Springer International Publishing, Cham, 297–303.

[6] Ahcène Bounceur. 2016. CupCarbon: A New Platform for Designing and Simulating Smart-City and IoT Wireless Sensor Networks (SCI-WSN). In *Proceedings of the International Conference on Internet of Things and Cloud Computing (ICC '16)*. ACM, Article 1, 1 pages.

[7] Radu Calinescu, Lars Grunske, Marta Z. Kwiatkowska, Raffaela Mirandola, and Giordano Tamburrelli. 2011. Dynamic QoS Management and Optimization in Service-Based Systems. *IEEE Trans. Software Eng.* 37, 3 (2011), 387–409.

[8] Maxim Chernyshev, Zubair Baig, Oladayo Bello, and Sherali Zeadally. 2017. Internet of things (iot): Research, simulators, and testbeds. *IEEE Internet of Things Journal* 5, 3 (2017), 1637–1647.

[9] Federico Ciccozzi and Romina Spalazzese. 2016. MDE4IoT: supporting the internet of things with model-driven engineering. In *International Symposium on Intelligent and Distributed Computing*. Springer, 67–76.

[10] Andrei Ciortea, Olivier Boissier, Antoine Zimmermann, and Adina Magda Florea. 2016. Responsive Decentralized Composition of Service Mashups for the Internet of Things. In *Proceedings of the 6th International Conference on the Internet of Things, IOT*. 53–61.

[11] European Commission. 2019. Next Generation Internet initiative. https://ec.europa.eu/digital-single-market/en/policies/next-generation-internet.

[12] Marco Conti, Andrea Passarella, and Sajal K. Das. 2017. The Internet of People (IoP): A new wave in pervasive mobile computing. *Pervasive and Mobile Computing* 41 (2017), 1–27. https://doi.org/10.1016/j.pmcj.2017.07.009

[13] Martina De Sanctis, Antonio Bucchiarone, and Annapaola Marconi. 2020. Dynamic adaptation of service-based applications: a design for adaptation approach. *J. Internet Serv. Appl.* 11, 1 (2020), 2.

[14] Martina De Sanctis, Henry Muccini, and Karthik Vaidhyanathan. 2020. Data-driven Adaptation in Microservice-based IoT Architectures. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 59–62.

[15] Martina De Sanctis, Romina Spalazzese, and Catia Trubiani. 2019. QoS-Based Formation of Software Architectures in the Internet of Things. In *Software Architecture - 13th European Conference, ECSA 2019*. 178–194.

[16] Nathalia Moraes do Nascimento and Carlos José Pereira de Lucena. 2017. Fiot: An agent-based framework for self-adaptive and self-organizing applications based on the internet of things. *Information Sciences* 378 (2017), 161–176.

[17] Bob Familiar. 2015. *Microservices, IoT, and Azure*. Springer.

[18] Luca Florio and Elisabetta Di Nitto. 2016. Gru: An approach to introduce decentralized autonomic behavior in microservices architectures. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 357–362.

[19] Miriam Gil, Vicente Pelechano, Joan Fons, and Manoli Albert. 2016. Designing the Human in the Loop of Self-Adaptive Systems. In *Ubiquitous Computing and Ambient Intelligence*. Springer International Publishing, 437–449.

[20] Xiang He, Zhiying Tu, Xiaofei Xu, and Zhongjie Wang. 2019. Re-deploying Microservices in Edge and Cloud Environment for the Optimization of User-Perceived Service Quality. In *Service-Oriented Computing - 17th International Conference, ICSOC 2019, Proceedings*. Springer, 555–560.

[21] Z. Houmani, D. Balouek-Thomert, E. Caron, and M. Parashar. 2020. Enhancing microservices architectures using data-driven service discovery and QoS guarantees. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 290–299.

[22] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. 2018. Microservices: The journey so far and challenges ahead. *IEEE Software* 35, 3 (2018).

[23] Anup K. Kalia, Pankaj R. Telang, Jin Xiao, and Maja Vukovic. 2017. Quark: A Methodology to Transform People-Driven Processes to Chatbot Services. In *Service-Oriented Computing - 15th International Conference*. Springer, 53–61.

[24] Hamzeh Khazaei, Alireza Ghanbari, and Marin Litoiu. 2018. Adaptation as a service.

[25] Nabor C. Mendonça, David Garlan, Bradley Schmerl, and Javier Cámara. 2018. Generality vs. Reusability in Architecture-Based Self-Adaptation: The Case for Self-Adaptive Microservices. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings (ECSA '18)*. Association for Computing Machinery, Article 18, 6 pages. https://doi.org/10.1145/3241403.3241423

[26] Andreas Metzger and Elisabetta Di Nitto. 2013. Addressing highly dynamic changes in service-oriented systems: Towards agile evolution and adaptation. In *Agile and Lean Service-Oriented Development: Foundations, Theory, and Practice*. IGI Global, 33–46.

[27] Mahda Noura, Sebastian Heil, and Martin Gaedke. 2020. Natural language goal understanding for smart home environments. In *IoT '20: 10th International Conference on the Internet of Things*. 1:1–1:8.

[28] Wenyi Qian, Xin Peng, Huanhuan Wang, John Mylopoulos, Jiahuan Zheng, and Wenyun Zhao. 2018. MobiGoal: Flexible Achievement of Personal Goals for Mobile Users. *IEEE Trans. Services Computing* 11, 2 (2018), 384–398.

[29] Antero Taivalsaari and Tommi Mikkonen. 2017. A roadmap to the programmable world: software challenges in the IoT era. *IEEE Software* 34, 1 (2017), 72–80.

[30] Omid Tavallaie, Javid Taheri, and Albert Y. Zomaya. 2019. QCF: QoS-Aware Communication Framework for Real-Time IoT Services. In *Service-Oriented Computing - 17th International Conference, ICSOC 2019, Proceedings*. Springer, 353–368.

[31] Danny Weyns. 2020. *An Introduction to Self-adaptive Systems: A Contemporary Software Engineering Perspective*. John Wiley & Sons.

[32] Shuai Zhang, Mingjiang Zhang, Lin Ni, and Peini Liu. 2019. A multi-level self-adaptation approach for microservice systems. In *2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*. IEEE, 498–502.