A DevOps Perspective for QoS-aware Adaptive Applications

Martina De Sanctis[1], Antonio Bucchiarone[2], Catia Trubiani[1]

[1] Gran Sasso Science Institute, L'Aquila, Italy
{martina.desanctis, catia.trubiani}@gssi.it
[2] Fondazione Bruno Kessler, Trento, Italy
bucchiarone@fbk.eu

Abstract. This paper presents a vision on how to apply the DevOps paradigm in the context of QoS-aware adaptive applications. The goal is to raise awareness on the lack of quantitative approaches that support software designers in understanding the impact of design alternatives at the development and operational stages. To this end, in this paper we: (i) verify the compliance of a design for adaptation approach with the DevOps life-cycle; (ii) perform the runtime monitoring of dynamic IoT systems, through Quality-of-Service (QoS) evaluation of system parameters, to guide a QoS-based adaptation with the goal of fulfilling QoS-based requirements over time.

1 Introduction

DevOps is a novel trend that aims to bridge the gap between software development and operation teams, and round-trip engineering processes become essential in such a context [1]. When applied to adaptive applications, it brings new challenges since it is still unclear when and how it is possible to enable which adaptations, even more, when evaluating the Quality-of-Service (QoS) characteristics of systems (e.g., performance and reliability) [2]. Moreover, applications are required to face the increased *flexibility* and *dynamism* offered by modern pervasive environments. This firmly demand for *adaptive* applications that are able to adapt to their actual environment (i.e., the currently available resources) and to new situations (e.g., missing services, changes in the user requirements and needs). Adaptive applications are a reality, and a key challenge is to provide the capability of dealing with the *continuously changing and complex environments* in which applications operate.

In our recent work [3] we introduced the automated formation of *the most suitable* Emergent Configurations (ECs) [4] in the Internet-of-Things (IoT) domain. In particular, ECs consist of a set of things that connect and cooperate temporarily through their functionalities, to achieve a user goal. To derive optimal ECs in terms of QoS, we make use of a model-based approach that embeds the specification of QoS-related properties of IoT things, and further support the specification of a QoS-based optimization problem returning the most suitable EC. However, one of the main limitations of [3] is that the runtime monitoring of the formed ECs has been neglected. In fact, there are some QoS-related characteristics associated with things that may change over time, e.g., the battery level of devices decreases when they are in use or increases after charging. These aspects of runtime adaptation of things are not handled in [3], where the selection of devices is driven by some preliminary check on their current status only. The goal of this paper is to extend the approach in [3] to enable the runtime monitoring of system parameters and trigger a QoS-based adaptation (e.g., switching among actuators with similar QoS-based characteristics but different battery level) of ECs periodically.

The rest of the paper is organized as follows. Section 2 describes a motivating example in the IoT context. Section 3 provides background information. In Section 4 we discussed how to match an adaptation-based approach to the DevOps life-cycle. The QoS-based adaptation of ECs in the IoT provided in this work, together with the conducted experimentation, are illustrated in Section 5. Section 6 reports our afterthoughts, while related work and conclusion are presented in Section 7.

2 Motivating Example

In this section we present the IoT Smart Light (SL) scenario, where things cooperate to achieve a predefined light level in a lecture room. Consider, for instance, a university campus made by different buildings hosting diverse types of rooms, e.g., libraries, dormitories, classrooms, offices. Each room is equipped with several IoT things, i.e., light sensors, curtains, and lamps. The things, along with their functionalities, are configured to be controllable via a mobile application allowing authorized users to increase/decrease the light level while moving in different rooms, based on their needs. For instance, in a lecture room, the lecturer can decide to decrease the light level when giving a presentation through a projector or, to the contrary, to increase it when using the blackboard. A possible way to achieve such goals is to dynamically identify an EC made, for instance, by the user's smartphone, a light sensor, and available curtain(s) and lamp(s). The selected light sensor measures the current light level in the room, and subsequently the lamps are turned on/off, and the curtains can be opened or closed. In our previous work [3], the scenario has been extended by adding the possibility of fulfilling extra-functional requirements besides the functional goal of the application (e.g., adjusting the light level). For instance, *the committer may want to minimize the power consumption of all the devices installed in the campus, while guaranteeing users' satisfaction*.

However, besides the *static attributes* (e.g., power consumption, sensing accuracy) that are provided by vendors and defined once for each device, other *dynamic attributes* that change over time or are independent from the vendor, must be considered when looking for the proper devices. In fact, it is necessary to take into account the system context evolution and adapt parameters accordingly. Specifically: (i) the selection of a sensor or an actuator leads to a decrease of a certain amount its battery level and activating their charging whether it is no longer available for the application, e.g., the battery level is lower than 50%. The charging status may be interrupted or devices might be constrained to fill their battery up to a certain threshold value (e.g., larger than 90%) before they can be selected again; (ii) devices can be subject to failures, making them no longer available for participating in the EC of devices; (iii) the light level in a room is further affected by environmental characteristics, e.g., the ambient light decreases while the evening comes, thus it might be that the provided light level may no longer meet the user requirement.

3 Background

This work builds upon an existing *design for adaptation* approach [5, 6] for adaptive applications. The aim of the approach is twofold: (i) introduce mechanisms enabling adaptation in the life-cycle of applications, both in the *design* and in the *run-time* phases and, (ii) support the continuous development and evolution of adaptive applications operating in dynamic environments. The approach relies on the Domain Objects (DOs) model. More precisely, DOs allow developers to define independent and heterogeneous things/services as well as their dynamic interactions in a uniform way. This way, they do not need to deal with the heterogeneity of things and their communication protocols, but they can work at a more abstract level. At design time, to model things, developers wrap them as DOs. Each DO implements its own behavior, namely the *core process*, which models its capability (e.g., the light sensing capability of a light sensor). Then, for its intended execution, a DO may optionally require capabilities provided by other DOs (e.g., the light sensing and the lamp actuating capabilities are externally required by the SL application). In fact, each DO exposes one or more *fragments* (e.g., the 'sense light level' fragment) describing offered services that can be *dynamically* discovered and used by other DOs. Both core process and fragments are modeled as *dynamically customizable processes*, by means of the Adaptive Pervasive Flows Language (APFL) [6]. We furthermore highlight that the task of wrapping things as DOs is done only *una tantum*, i.e., when a new device type/brand is available.

Since the actual system can vary in different execution contexts, the system realization is performed closer to its execution, that is, when the actual environment is known. This guarantees as much as possible the success of the provided applications (e.g., in terms of their applicability, correct execution, coherent adaptation). The dynamic cooperation among DOs is performed by exploiting different *adaptation mechanisms and strategies* [7] (e.g., refinement mechanism) allowing applications to adapt to different situations (e.g., select the proper services, react to a context change) at runtime. At design time, APFL allows the partial specification of the expected behavior of a DO through *abstract activities*, i.e., activities that the DO does not implement itself; they are defined only in terms of a goal (e.g., sense the light) and they represent open points in DOs' processes and fragments. At runtime, the refinement mechanism allows abstract activities *to be refined* through the (composition of) fragments offered by other DOs, whose execution leads to achieve the abstract activity's goal. This enables a *chain of refinements*, supported by advanced techniques for the dynamic and incremental service composition and re-configuration, based on Artificial Intelligence (AI) planning [8]. To experiment with this approach please refer to [9].

When considering QoS-based characteristics of adaptive applications, further challenges arise, since it is indeed not trivial to evaluate such characteristics (e.g., system response time) of systems subject to run-time variability (such as workload fluctuations, services availability). In this case, adaptation needs have to be extended to target functional goals while meeting QoS-based requirements. To this aim, in our recent work

[3] we extended the approach by embedding the specification of QoS-related properties at the level of things. This allows the automated formation of the most suitable ECs relying on the selection of QoS-based optimal devices.

4 Compliance of the design for adaptation with the DevOps life-cycle

The DevOps paradigm recently emerged [10] to decrease the gap between the design of a software product and its operation. Such paradigm provides a process (i.e., Dev –*plan, code, build, test*–, and Ops –*release and deploy, operate, monitor*–), but it is not constrained to any specific modeling and analysis formalism, or tool. This implies that the life-cycle is based on a set of pillars (e.g., collaborative development and continuous integration [11]), but its implementation is fully decidable by the DevOps team.

Similarly, our design for adaptation approach [5, 6] has also been defined with the aim of reducing the distance between the design of adaptive applications and their run-time adaptation. Hence, we are interested to investigate at what extent the design for adaptation approach can be considered compliant with the DevOps paradigm.

To this end, we performed a mapping between the two life-cycles, as sketched in Figure 1. For each DevOps phase, we describe the corresponding activities performed when exploiting the design for adaptation approach.

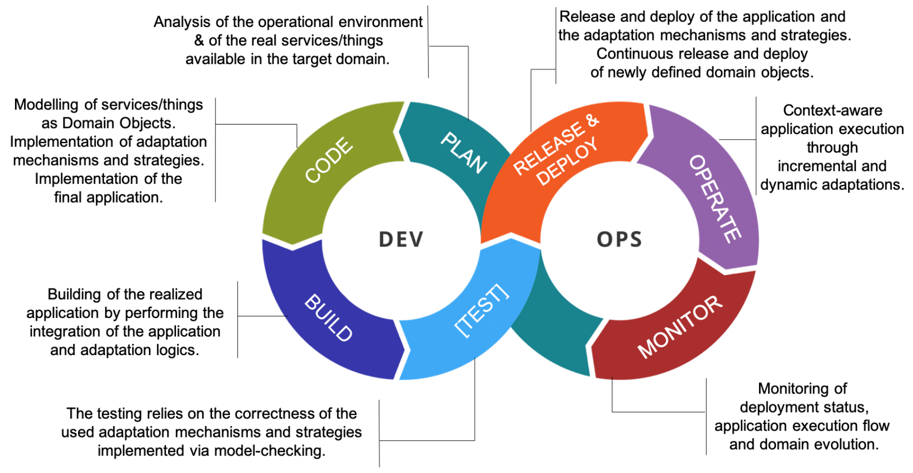

Analysis of the operational environment & of the real services/things available in the target domain.
Release and deploy of the application and the adaptation mechanisms and strategies. Continuous release and deploy of newly defined domain objects.
Modelling of services/things as Domain Objects. Implementation of adaptation mechanisms and strategies. Implementation of the final application.
Context-aware application execution through incremental and dynamic adaptations.
CODE
PLAN
RELEASE & DEPLOY
OPERATE
DEV
OPS
BUILD
[TEST]
MONITOR
Building of the realized application by performing the integration of the application and adaptation logics.
The testing relies on the correctness of the used adaptation mechanisms and strategies implemented via model-checking.
Monitoring of deployment status, application execution flow and domain evolution.

Fig. 1: Detailed mapping between the Design for Adaptation and DevOps life-cycles.

Plan. Our approach supports the planning of application development, which can be seen as the transition towards the code stage. This is done by defining a domain model that describes the specific operational environment in which the provider wants to instantiate the application to be developed (e.g., the SL application). More precisely, the domain is modeled as a set of domain properties describing specific concepts of the domain (e.g., light sensing, lamp actuating). At the same time, another important task to accomplish during this phase is an accurate analysis of the available devices/services that are part of the targeted domain (e.g., available devices, their functionalities and brands), which will be used in the next stages.

Code. In this phase, three main tasks are performed: (i) the identified devices and services are wrapped as DOs; (ii) the adaptation mechanisms and strategies provided by the approach are enabled (this task does not require any development activity to developers); (iii) the final application, such as SL, is designed as a DO as well. DOs are specified in the xml language while the modeling of DOs behaviors (i.e., core processes and fragments) is performed by using APFL [6]. No further programming languages are needed at this stage. In Figure 2 we report an example of a light sensor (i.e., the Sensmitter[3]) expressed as a .xml file representing the corresponding light sensor's DO. In particular, it shows the pointers to the DO's constituent parts, such as its domain knowledge (see lines **4-7**), its state (see lines **9-25**), its core process (see line **27**), and its fragment(s) (see line **28**). The DO's state also contains QoS-related attributes (see lines **16-24**), besides state variables. Specifically, regarding the SL scenario, the specification of light sensors is augmented with three metrics such as power consumption, sensing accuracy and battery level[4]. Differently from the other attributes, the battery level is dynamic, i.e., the state of the device' battery can be dynamically updated since it changes over time.

```
 1<?xml version="1.0" encoding="UTF-8"?>
 2<tns:domainObject name="SensmitterLightSensor" xmlns:tns="http://.../">
 3
 4    <tns:domainKnowledge>
 5        <tns:internalDomainProperty name="domainProperties/LightSensing">
 6    </tns:internalDomainProperty>
 7    </tns:domainKnowledge>
 8    <!-- List of state variables -->
 9    <tns:state>
10        <tns:stateVariable name="DeviceID" type="string">
11            <tns2:content type="anyType">Sensmitter_435</tns2:content>
12        </tns:stateVariable>
13        <!-- Other state variables here -->
14
15        <!-- QoS-related attributes -->
16        <tns:QoSAttribute name="PowerConsumption" type="integer">
17            <tns2:content type="anyType">2.5</tns2:content>
18        </tns:QoSAttribute>
19        <tns:QoSAttribute name="SensingAccuracy" type="integer">
20            <tns2:content type="anyType">8</tns2:content>
21        </tns:QoSAttribute>
22        <tns:QoSAttribute name="BatteryLevel" type="integer">
23            <tns2:content type="anyType">100</tns2:content>
24        </tns:QoSAttribute>
25    </tns:state>
26
27    <tns:process name="processes/PROC_SensmitterLightSensor"/>
28    <tns:fragment name="fragments/LS_senseLight"></tns:fragment>
29
30</tns:domainObject>
```

Fig. 2: Domain object model for the Sensmitter light sensor [3].

[3] https://www.senssolutions.se/ [4] Note that metrics can be expressed in different units for sensors and actuators of different brands, however such units can be converted to a common reference unit in the DO model, thus to avoid misleading comparison.

Build. In this phase the application is built by performing the integration between the application and adaptation logic. Indeed, the former represents *what* the application is designed to do (e.g., adjust the light level) and it is specified at design time. The latter specifies *how* to do it (e.g., by combining available light sensors and actuators), and it can vary in different execution contexts, since the availability of things can change dynamically. To deal with this dinamicity, the application logic includes open points where the adaptation can take place (e.g., dynamic devices selection).

Test. This phase deals with the testing of the final application when the adaptation mechanisms and strategies are in their operational stage. Currently, our approach does not provide a testing environment able to simulate the adaptive applications. However, as demonstrated in [8], the services composition (i.e., a plan) returned by the adaptation planner (implementing the adaptation mechanisms and strategies via model-checking) is correct by construction. In other word, if a plan is found, it is guaranteed that its execution allows the application to reach a situation in which the goal of the adaptation problem is satisfied.

Release & Deploy. After the testing, both the SL application and the adaptation mechanisms and strategies are released and deployed, being ready for the execution. Obviously, the application domain continuously evolves due to new available services and devices (e.g., a new light sensor of a different brand has been installed in the campus). This triggers the application evolution resulting in continuously releasing and deploying newly defined DOs.

Operate. At operational stage, our approach provides enablers for the automatic formation and adaptation of ECs. After the approach extensions made in [3], the devices selection and composition is also driven by QoS-related requirements (e.g., the sensing accuracy of light sensors has to be larger than a threshold). As introduced in Section 3, the run-time operation of applications relies on different adaptation mechanisms and strategies [7] that handle the dynamicity of the environment in which applications operate. These mechanisms are process-based and, in particular, they rely on the use of APFL and its constructs (i.e., abstract activities, annotations). Figure 3 provides an ex-

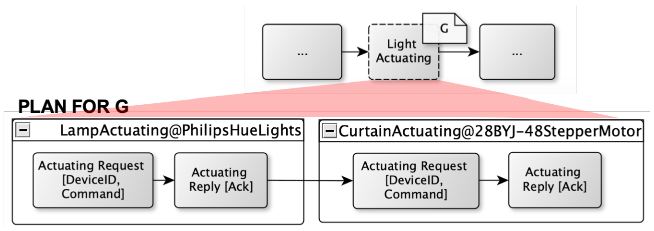

...
Light Actuating
G
...
PLAN FOR G
LampActuating@PhilipsHueLights
Actuating Request [DeviceID, Command]
Actuating Reply [Ack]
CurtainActuating@28BYJ-48StepperMotor
Actuating Request [DeviceID, Command]
Actuating Reply [Ack]

Fig. 3: Excerpt of the Smart Light execution example [3].

ample of the *abstract activity refinement mechanism* to better understand the adaptation functioning. It represents an excerpt of the SL execution, i.e., the refinement of the *Light Actuating* (goal G in Figure 3) abstract activity, depicted with a dotted line[5]. The fragments composition returned for this refinement is made by two fragments provided by those actuators in the room whose QoS-related characteristics are compliant with the

[5] The complete overview of the SL execution can be found in [3].

QoS-based requirements stated in the SL application (e.g., minimize power consumption). Specifically, the fragments *Lamp* and *Curtain Actuating*, respectively provided by the *Philips Hue Lights*[6] and the *Stepper Motor*[7] DOs are selected, composed and injected in place of the abstract activity they refine.

Monitor. The connection point between the Dev and the Ops cycles is a monitoring step that is in charge of enacting changes, triggered by both evolution and adaptation needs. Indeed, in this phase, the deployment status, the application execution flow and the domain evolution are constantly observed. Furthermore, there are some QoS-related characteristics associated to things that may change over time, e.g., the battery level. These aspects of runtime evolution of things have not been handled by our approach in [3] that instead computes some preliminary check on the current status of devices only. In this work, instead, we enable a runtime monitoring to update these changing values and trigger a QoS-based adaptation (e.g., switching among actuators showing similar QoS-based characteristics but with different battery level) of ECs, periodically. In particular, performing a QoS-based adaptation does not lead to a new execution of the whole DevOps life-cycle. As opposite, the detection of application's evolution needs (e.g., availability of new device types in the operational environment) leads to a new execution of the DevOps life-cycle.

Summarizing, the stages described above represent the DevOps-based contributions of the design for adaptation approach. As expected, due to its aim of reducing the distance between the design of adaptive applications and their runtime adaptations, the life-cycle of our approach easily maps with DevOps. Moreover, the approach itself is extended in this paper to deal with the monitoring of QoS-based characteristics of adaptive applications subject to runtime variabilities, thus to further contribute to the DevOps paradigm.

5 QoS-based evaluation of adaptive by design applications

In this section we provide an overview of the functioning of QoS-based adaptation of previously formed and enacted ECs, and we report on a concrete example (see Section 2) some experimental results demonstrating the usefulness of the approach.

5.1 Overview

The automated formation of ECs based only on purely functional requirements shows the drawback that an optimal usage of resources is not guaranteed, possibly leading to end-users unsatisfaction. To deal with these issues, in [3] we extended our modeling formalism to include the specification of QoS-based properties, and enable a QoS-aware formation of ECs. The specification of QoS-related characteristics, indeed, is performed at the level of DOs, as shown in Figure 2. In particular, each thing is associated to an arbitrary number of metrics inherited from its producer. Thus, we enhanced the specification of DOs (e.g., those representing real world things in the environment) by adding QoS-related attributes. The default setting of extra-functional requirements (i.e., min, max, threshold value) is enabled by developers in the setting of the SL application, but

[6] https://www2.meethue.com/en-us [7] https://bit.ly/2VmRegr

no assurance can be given. End-users may have different preferences while using the available things, hence they can modify such requirements. This is later translated into the QoS-based optimization problem that guides the formation of the most suitable ECs.

In this paper we are interested to study the usefulness of our approach in terms of QoS-based resilience to changes and their impact on the DevOps life-cycle. To this aim, we enabled the QoS-based adaptation of ECs, by implementing a runtime monitoring, i.e., periodically monitor the dynamic attributes of the devices involved in a running EC. Specifically, instead of considering static attributes only, in the following we focus on dynamic attributes that require adaptation while the system is up and running. The approach we propose in this paper contributes to the DevOps domain since it jointly considers development and operational properties of software systems; more in details it tackles the following three main characteristics:

- **updates of inner system parameters at operational stage**, i.e., there are some system characteristics that change overtime and it is necessary to update their value, e.g., the battery level of mobile devices is consumed when they are in operation and such a parameter requires to be updated accordingly.
- **system failures**, i.e., there are some software and hardware components that do not properly work at operational stage and it is necessary to substitute them, or to foresee recovery techniques that allow their recovery. This last point opens an interesting line of research that we leave as part of our near future research.
- **environment**, i.e., there are some environmental characteristics that may affect the users' perception and contribute to the selection of different design alternatives, e.g., the light of the day may be low, medium, and high, and this contributes to calculating differently the required leftover. For example, in our motivating example (see Section 2), if the required light level is equal to ten, but e.g., the environment provides eight already, then lamp actuators are required to fill the remaining two units of lighting.

5.2 Experimentation

Experimental settings. Table 1 reports the QoS-related characteristics of employed devices. Specifically, our scenario includes light sensors and lamps acting as actuators. For sensors we have five different instances of a different brand ($LS_1, \dots, LS_5$), and their sensing accuracy is specified in the first row of the table. For lamps, we also have five different instances of a different brand ($LA_1, \dots, LA_5$), and their light level is specified in the second row of the table. All devices show a battery decrease unit that indicates how much their battery is decreased when they are in operation. Such values are reported in the last row of Table 1. All these values represent an estimation of QoS-related characteristics for arbitrary things, however, further numbers can be considered as well when other specification of things is available.

In the following we discuss three main experiments that have been performed to evaluate different *dynamic* aspects of our motivating example.

$Experiment_1$ - runtime availability of devices conditioned to their battery level. In this experiment we are interested to show that the selection of devices is currently driven by the battery level that is updated and changes over time. This means that when

Table 1: QoS-related characteristics.

	Light Sensors					**Lamps**				
	LS_1	LS_2	LS_3	LS_4	LS_5	LA_1	LA_2	LA_3	LA_4	LA_5
sensing accuracy	4	7	2	10	8	-	-	-	-	-
light level	-	-	-	-	-	4	8	6	1	3
battery decrease	1	2	3	4	5	2	4	6	8	10

selected for use, the battery of devices is decreased and, when lower than a certain threshold (e.g., 70% in our experimentation), the devices are set to a charging state and not available up to when they are usable again since their battery level goes over a predefined threshold (e.g., 80% in our experimentation).

$Experiment_2$ - runtime availability of devices conditioned to their battery level and failures. In this experiment we aim to demonstrate that software and hardware failures randomly occur and they affect the operational stage of the system. The application needs to take such failures into account. This analysis is also helpful when considering the number of sensors and actuators to put in place, in fact it is obvious that a restricted number of available devices may lead to no alternative options whether all sensors and/or actuators are in a failing state condition.

$Experiment_3$ - runtime availability of devices conditioned to their battery level, failures and conditions of the environment. As mentioned above, there might be some environmental aspects that affect the users' perception. For example, the light is conditioned to the time of the day and available actuators are required to fill the gap between the perceived environment and the application requirements. This means that depending on the time of the day, some actuators may be not valid for the fulfillment of stated system requirements. We considered three main light environment levels (i.e., low, medium, and high) and we investigated the availability of lamps moving among such three settings.

In the remaining of the section we argue on collected experimental results while considering multiple observation runs (i.e., up to 100 in our experimentation) denoting several intervals of time of the same duration when the application is up and running.

Experimental results. In the following we describe the conducted experiments and collected results, related to the three experiments reported above.

Figure 4 shows the runtime availability of devices conditioned to their battery level. On the x-axis we report the number of system runs, i.e., up to 100 time frames. On the y-axis we show the total number of discarded devices, including both sensors and actuators. As expected, initially all devices are up and running, at the 16-th run one of the devices shows a low battery, later on two devices, and so on. These discarded devices are recharged in the subsequent runs, in fact at the 23-th run no devices are discarded, their battery is considered enough to accomplish the planned tasks. As expected, we can notice that the availability of devices is fluctuating, and the number of discarded devices varies between one and six, but in average across all runs it turns out to be around three. All these numbers are indicators of the devices availability over time due to their battery characteristics, and these evaluations are helpful to understand the evolution of the application under analysis.

Figure 5 depicts the runtime availability of devices conditioned to their battery level and random failures. For lack of space and readability reasons, we show 1-25 observa-

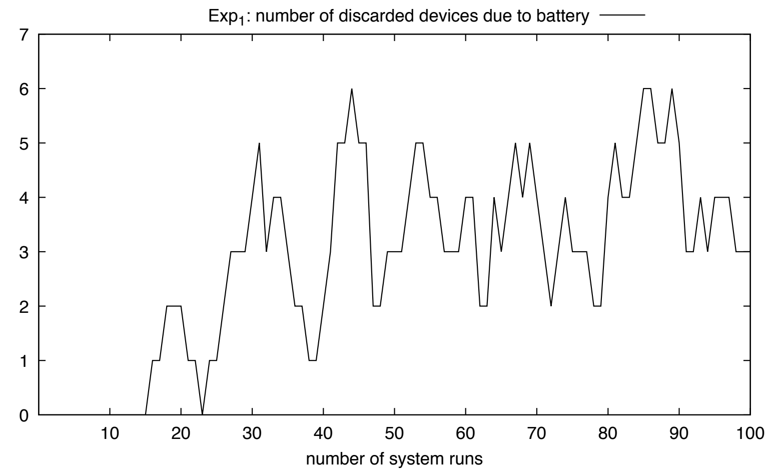
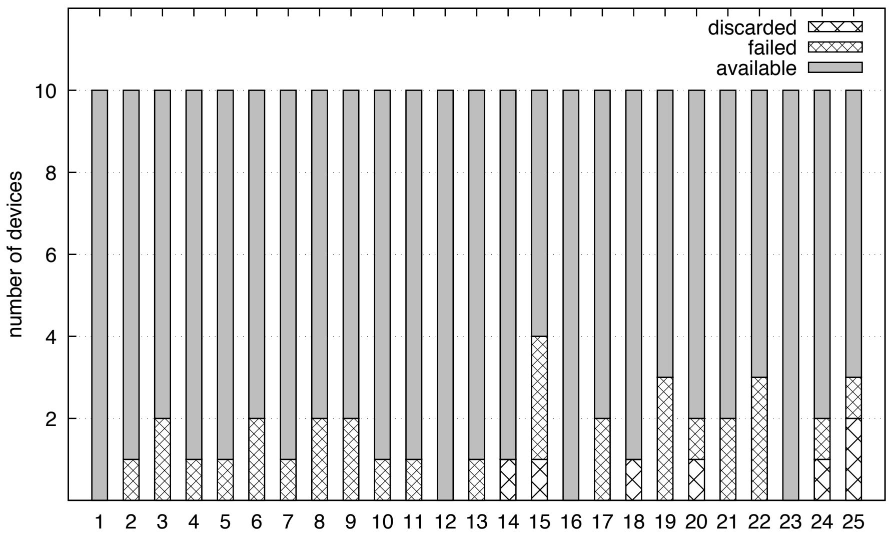


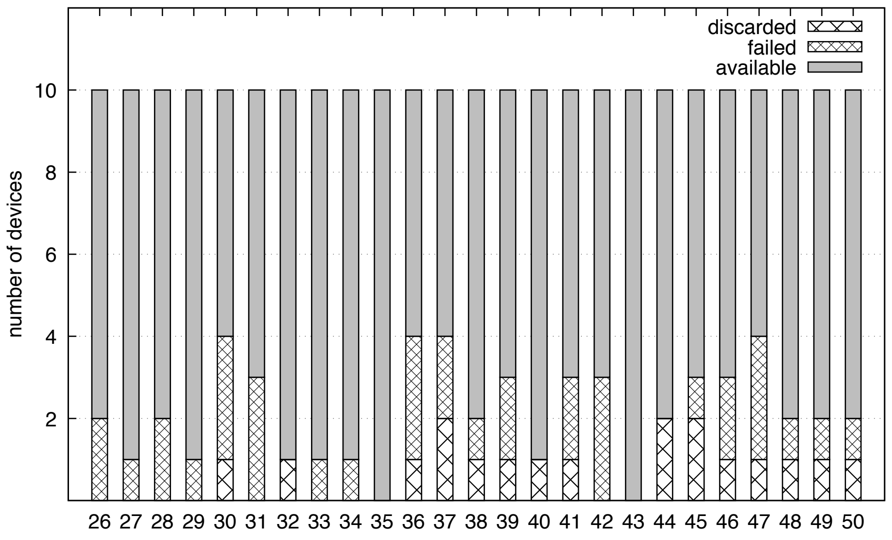
Fig. 4: Exp_1 - runtime availability of devices conditioned to their battery level.

tion runs in Figure 5(a), and 26-50 observation runs in Figure 5(b); the remaining runs are omitted but show a similar trend. At the first run all devices are available, but starting from the second run and up to the 11-th run we can notice up to two devices (per run) showing failures. At the 12-th run all the devices are available again, and discarded devices (due to low battery) start to appear at the 14-th run. This is because the battery decrease requires some time to become critical, whereas failures are random and can happen anytime, even in the first run itself. Some runs encounter a mix of failed and discarded devices, e.g., at the 15-th run we have one discarded device and three failed ones. Later on, we can see that few runs show the availability of all devices, but most of them report failures and/or devices with low battery. In average, across all the 100 runs, we found 0.64 discarded devices due to low battery and 1.29 failed devices. Similarly to the previous experiment, all these numbers represent indicators of the devices availability over time due to battery decrease and failures, and our application resulted to be exposed to few unavailable devices on average.

Figure 6 shows the runtime availability of devices conditioned to their battery, random failures, and different levels of environment light. Figure 6(a) focuses on a low environment light level (e.g., early morning), i.e., the sensing reports a light level set from the environment that is estimated to be equal to 2. To achieve the stated requirement (light level larger than 5 units), all the devices contributing with a light level larger than 3 units are sufficient to fulfill the end users need. When moving to the medium scenario (e.g., late morning), see Figure 6(b), we considered 4 units as the environment light level; for the high scenario (e.g., mid day), see Figure 6(c), we stick on 5 units that basically indicate the suitability of all devices, no matter of their power since the environment itself is sufficient. As expected, in Figure 6 we can notice that the number of discarded devices due to the environment progressively decreases while moving across the observation runs, and this is due to an increasing environment light level. In average, across the 60 runs, we found an average of 0.4 discarded devices (due to battery), 1.05 failed devices, and 0.9 discarded devices (due to the environmental conditions). Similarly to previous experiments, all these numbers indicate that our application is marginally affected by battery, failures, and environmental changes.

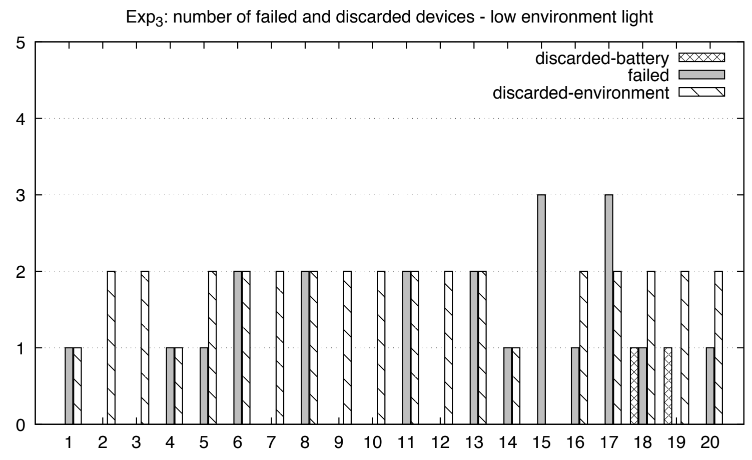


(a) 1-25 observation runs.

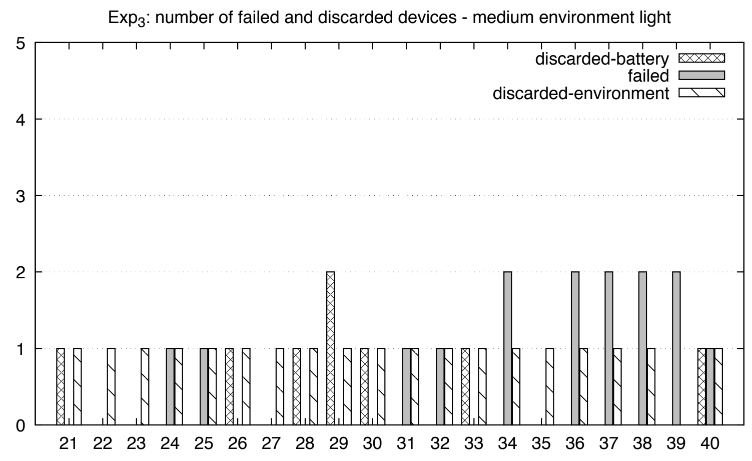


(b) 26-50 observation runs.

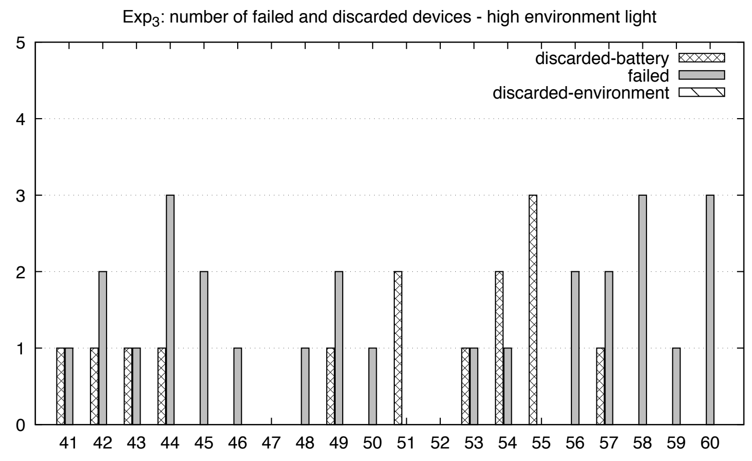
Fig. 5: Exp_2 - runtime availability of devices conditioned to their battery level and random failures (up to 50 observation runs).



(a) low light: 1-20 observation runs.



(b) medium light: 21-40 observation runs.



(c) high light: 41-60 observation runs.

Fig. 6: Exp_3 - runtime availability of devices conditioned to their battery, random failures, and different levels of environment light.

Summarizing, the presented three experiments aim to demonstrate that QoS-based adaptation is feasible, and our approach allows the quantification of how the design settings (i.e., the intrinsic characteristics of available devices) contribute to some properties of the running application, thus to get further knowledge on its operational stage. Note that the execution time of running 100 runs varies in a narrow interval, and its average is less than 2 seconds, thus to assess the efficiency of the approach.

6 Discussion

Our approach includes a set of limitations that we discuss in the following.

Flexible guarantees. It may happen that, when looking for an EC, the application is not able to provide a solution that strictly meets the stated requirements. This problem is exacerbated when QoS-based requirements are also considered, since they further restrict the devices selection space. As future work we plan to introduce techniques that identify the "closest" EC, i.e., the one slightly deviating from requirements, but probably still satisfying the end users. This can be performed by moderately modifying the stated requirements and executing the process as it is.

Recovery policies. As demonstrated in our experimentation, devices are subject to failures and this implies a lower number of suitable ECs. To address this issue, we plan to extend our approach by minimizing the mean time to repair, so that failed devices are fixed more efficiently and they soon become available again. To this end, a component should be added to trigger recovery policies that automatically provide strategies (e.g., restart, reset to default settings) for some or all sensors/actuators.

Adaptation as a DevOps-cycle itself. In this paper we proposed a mapping between the life-cycle of a design for adaptation of adaptive applications on the classical DevOps life-cycle. This means adaptation is an integrated activity in all the DevOps phases within the considered application. As opposite, we can envision a DevOps-cycle focusing on adaptation mechanisms only, so that design, build, test, deliver, operation and evolution are dedicated phases for adaptation concerns. In other words, applications and adaptation mechanisms should be correlated but, at the same time, able to evolve independently of each other, also considering that they might be realized by diverse professionals with different skills and roles.

Centralized vs. distributed. Our approach is currently centralized, in fact it leverages on an *adaptation engine* and a *process engine*, both operating in a centralized manner. As future work, we plan to decentralize these two engines, so that the execution of applications can run in distributed environments. As consequence, the QoS-based evaluation of adaptive applications must evolve accordingly, in order to manage those QoS characteristics particularly affected by distributed executions, e.g., response time.

Adaptation and application testing. As discussed in Section 4, currently our approach does not provide a testing environment and, for the correct execution of applications, it relies on the used adaptation mechanisms and strategies that exploit model-checking [8]. As future work, we plan to implement our own testing environment able to verify the built applications (w.r.t. requirements) through the evaluation of (i) applications executions, (ii) services composition, and (iii) (QoS-based) adaptations.

7 Related work and Conclusion

Evaluating QoS-based characteristics of adaptive systems and applications is not trivial, in fact they are subject to run-time variability (such as workload fluctuations, services availability). As we said, in this case, adaptation needs have to be extended to target functional goals while meeting QoS-based requirements. Our previous work in this direction focused on performance-related issues: in [12] we make use of performance models to analyze the dynamics of performance indices; in [13] performance models are guided by model predictive control techniques to achieve performance guarantees; in [14] we identified the sources of uncertainties (e.g., deployment infrastructure) affecting performance in the DevOps life-cycle.

In [15] the authors introduce ENACT aimed to enable DevOps in trustworthy smart IoT systems. To this end, they propose to evolve DevOps methods and tools to address specific IoT related challenges, as for instance the continuous quality assurance. In our approach, the adaptation takes place at behavioral level, by exploiting a domain-independent approach. In contrast, ENACT, whose focus is mainly on the trustworthiness, supports architectural adaptations at operational time and with a strong focus in the IoT domain. Cito et al. [16] advocate the need to capture feedback from operations data and mapping them on software development life-cycle phases, in order to drive informed decisions. This becomes particularly relevant in the DevOps context that aims on promoting synergies between the development and execution of software systems. As a first attempt in this direction, we enabled the monitoring of dynamic QoS-based characteristics whose changes trigger a new execution of the SL application leading to a new EC of sensors and actuators. As discussed in section 6, other steps are required to enable a feedback-loop between the Dev and Ops cycles. The work in [17] shares with our work the idea that IoT-based applications must be able to automatically adapt to changes in the QoS of their component services. To this aim, the authors exploit a collaborative QoS prediction of candidate services that enables a goal-driven service composition which, in turn, allows a QoS-based adaptation to be performed. In [18] a DevOps environment for design-time modeling and optimization, and runtime control is proposed. Here, the goal is to minimize the execution cost of cloud applications providing QoS guarantees by design. Summarizing, all the discussed approaches represent valid competitors with our DevOps-based design for adaptation, and we plan to further investigate the comparisons in the near future.

In this paper we presented a DevOps perspective for QoS-aware adaptive applications. In particular, we provided a mapping between an approach for adaptive by design applications and the DevOps life-cycle, along with QoS-based adaptation. A motivating example illustrates the feasibility of the approach, and calls for future research. Besides all the directions mentioned in Section 6, we are interested to industrial case studies to further investigate the soundness of the proposed methodology.

References

[1] M. A. Jiménez, L. Castaneda, N. M. Villegas, G. Tamura, H. A. Müller, J. Wigglesworth, Devops round-trip engineering: Traceability from dev to ops and back again, in: International Workshop DEVOPS, 2018, pp. 73–88.

[2] M. U. Iftikhar, D. Weyns, Activforms: A runtime environment for architecture-based adaptation with guarantees, in: International Conference on Software Architecture - Workshops, 2017, pp. 278–281.
[3] M. De Sanctis, R. Spalazzese, C. Trubiani, Qos-based formation of software architectures in the internet of things, in: European Conference on Software Architecture, 2019, pp. 178–194.
[4] F. Alkhabbas, R. Spalazzese, P. Davidsson, Architecting emergent configurations in the internet of things, in: International Conference on Software Architecture, 2017, pp. 221–224.
[5] A. Bucchiarone, M. De Sanctis, A. Marconi, M. Pistore, P. Traverso, Design for Adaptation of Distributed Service-Based Systems, in: International Conference on Service-Oriented Computing, 2015, pp. 383–393.
[6] A. Bucchiarone, M. De Sanctis, A. Marconi, M. Pistore, P. Traverso, Incremental Composition for Adaptive by-Design Service Based Systems, in: International Conference on Web Services, 2016.
[7] A. Bucchiarone, A. Marconi, M. Pistore, H. Raik, A context-aware framework for dynamic composition of process fragments in the internet of services, Journal of Internet Services and Applications 8 (1) (2017) 6.
[8] P. Bertoli, M. Pistore, P. Traverso, Automated composition of web services via planning in asynchronous domains, Artif. Intell. 174 (3-4) (2010) 316–361.
[9] M. De Sanctis, A. Bucchiarone, A. Marconi, ATLAS: a new way to exploit world-wide mobility services, Software Impacts 1 (2019) 100005.
URL http://www.sciencedirect.com/science/article/pii/S2665963819300053
[10] L. Bass, I. Weber, L. Zhu, DevOps: A Software Architect's Perspective, Addison-Wesley Professional, 2015.
[11] J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, 1st Edition, Addison-Wesley Professional, 2010.
[12] E. Incerto, M. Tribastone, C. Trubiani, A proactive approach for runtime self-adaptation based on queueing network fluid analysis, in: International Workshop on Quality-Aware DevOps, 2015, pp. 19–24.
[13] E. Incerto, M. Tribastone, C. Trubiani, Software performance self-adaptation through efficient model predictive control, in: International Conference on Automated Software Engineering, 2017, pp. 485–496.
[14] C. Trubiani, P. Jamshidi, J. Cito, W. Shang, Z. M. Jiang, M. Borg, Performance Issues? Hey DevOps, Mind the Uncertainty, IEEE Software 36 (2) (2019) 110–117.
[15] N. Ferry, A. Solberg, H. Song, S. Lavirotte, J. Tigli, T. Winter, V. Muntés-Mulero, A. Metzger, E. R. Velasco, A. C. Aguirre, ENACT: development, operation, and quality assurance of trustworthy smart iot systems, in: International Workshop DEVOPS, 2018, pp. 112–127.
[16] J. Cito, J. Wettinger, L. E. Lwakatare, M. Borg, F. Li, Feedback from operations to software development—a devops perspective on runtime metrics and logs, in: International Workshop DEVOPS, 2019, pp. 184–195.
[17] G. White, A. Palade, S. Clarke, Qos prediction for reliable service composition in iot, in: International Conference on Service-Oriented Computing - Workshops, 2017, pp. 149–160.
[18] M. Guerriero, M. Ciavotta, G. P. Gibilisco, D. Ardagna, A Model-Driven DevOps Framework for QoS-Aware Cloud Applications, in: International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2015, pp. 345–351.