Truong, Linh; Klein, Peter

DevOps Contract for Assuring Execution of IoT Microservices in the Edge

Published in:
Internet of Things

DOI:
10.1016/j.iot.2019.100150

Published: 01/03/2020

Document Version
Publisher's PDF, also known as Version of record

Internet of Things 9 (2020) 100150

Contents lists available at ScienceDirect

Internet of Things

journal homepage: www.elsevier.com/locate/iot

DevOps Contract for Assuring Execution of IoT Microservices in the Edge

Hong-Linh Truong [a,*], Peter Klein [b]

[a] Department of Computer Science, Aalto University, Finland
[b] Independent, Austria

ARTICLE INFO

Article history:
Received 5 May 2019
Revised 30 November 2019
Accepted 1 December 2019
Available online 04 December 2019

Keywords:
IoT
Service contract
Edge computing
Execution management

ABSTRACT

The increasing availability of edge and IoT infrastructure-as-a-service allows us to develop lightweight IoT components and deploy them into edge/IoT infrastructures, enabling edge analytics and controls. This paper introduces the development of service contracts for IoT microservices from DevOps perspectives. We analyze stakeholders and present our methods to support stakeholders to program IoT service contracts. We address the diversity of service contracts by using common languages for IoT data and programming. We integrate the development and operation lifecycle of IoT contracts with IoT software components and with supporting DevOps services. To illustrate our approach, we use a real-world Base Transceiver Station maintenance application with Raspberry Pi, Java, JavaScript, JSON and other microservices.

1. Introduction

The rise of IoT device-as-a-service, IoT infrastructure-as-a-service and edge infrastructure-as-a-service [1] has created several challenges for software development. We have observed two main development streams in IoT service offerings: (i) either IoT infrastructures provide data and services that one can consume through the subscription of the data and services or (ii) one can deploy his/her own software components, such as a microservice or a serverless function, into IoT infrastructures to perform certain functions for his/her services. Regardless of which models, the contract/agreement between the consumer and the IoT/edge infrastructure provider is crucial. However, while contracts have been reasonably supported for the case in which the consumer accesses data and services through remote (REST/MQTT) APIs, the contract support for the second case to allow the customer to deploy and run his/her services has not been well researched. The first case is popular partially because we can utilize many existing contract and access control models, such as data-as-a-service contracts, web services contracts, data subscription controls and remote access monitoring [2–4]. In the second case, it is quite tricky because, similar to the cloud infrastructure that one can buy virtual machines/containers and run one's microservices, the consumer can run his/her code in IoT gateways and edge servers in the IoT/edge infrastructures. The consumer might also utilize virtualized components like Docker to run his/her code [5,6]. The current focus on edge computing and fog computing has introduced various frameworks for deployment and execution of components in edge/IoT infrastructures on-demand,

* Corresponding author.
E-mail addresses: linh.truong@aalto.fi (H.-L. Truong), peterklein2308@gmail.com (P. Klein).

https://doi.org/10.1016/j.iot.2019.100150

such as [5,7–9]. However, little effort has been spent for examining service contracts along existing phases of the IoT software development in the context of IoT-as-a-service, where IoT and edge infrastructures can be provided under on-demand, pay-per-use principles. We believe that, given the increasing development and execution of software components in the IoT/edge infrastructures, we need to dedicate more effort on improving the service contract in IoT software development.

Few works have started to address the above-mentioned problems, but quite pre-mature. Traditional approaches, like using pre-defined policy languages, are too static, as the contract terms and constraint specifications are defined only at the time we deploy the code. Other models like IoT access control also use quite static policies [10]. Furthermore, policy syntaxes are predefined to allow only instantiating variables/metrics, thus it is difficult to cope with the diversity of IoT devices and tasks. In our work, we consider the case in which one will develop IoT software service – called *IoT (micro)services* – from various components – called *IoT units* – and deploy such services in IoT infrastructure-as-a-service. The key question is how to support the development of contracts, and testing and deployment of them as early as possible, like in DevOps and Security DevOps phenomenons and in Site Reliability Engineering [11]. The challenges for contracts in IoT are due to diverse underlying APIs for data and control that need to be integrated well with the development and operation of IoT units/services. While we can leverage quite common structures and categories of contracts from the related work, it is not enough to start the integration and development of contract only when deploying IoT components to the infrastructure. In this paper we examine the following key question: if a developer follows DevOps for IoT units/microservices, what would be steps/methods to do w.r.t services contracts for such IoT software?

In this work, we call our approach *DevOps Contract for IoT microservices*. We focus on execution management and monitoring aspects in service contracts (e.g., we do not focus on penalty or compensation). We present a set of microservices, templates and techniques for building IoT contracts from contract models and assigning them to IoT units. We present building blocks for creation of contracts as well as for enforcement using common languages for data and programming, like JSON and Javascript. Our runtime services for DevOps Contract include registry, deployment, enforcement and monitoring features. We also integrate our work with blockchain technologies to support verifiable assets and violation records in IoT contracts. To illustrate our work, we will use the case of infrastructure maintenance using IoT software for Base Transceiver Stations.

The rest of this paper is as follow. Section 2 presents motivation and approach. We present our DevOps Contract approach in Section 3. We present building blocks for contracts in Section 4 and for enforcement in Section 5. Section 6 presents our prototype and runtime evaluation. Related work is discussed in Section 7. We conclude the paper and outline future work in Section 8.

2. Motivation and Approach

Similar to DevOps in the Cloud, a typical DevOps process for components in edge/IoT infrastructure-as-a-service usually includes many tasks: writing, testing and deploying components [12] and these tasks can be carried out through CI/CD tools. However, due to the complexity of underlying IoT infrastructures and devices, IoT requires many collaboration among various stakeholders to deliver the right software [13]. In a typical development environment, the code of IoT components is developed and managed in the cloud, being tested and pushed to the IoT/edge. At runtime, to apply key principles of Site Reliability Engineering [11] and Service Level Agreements (SLAs) [14] to IoT components being executed in IoT infrastructures, both infrastructures and providers of IoT must support and guarantee various Service Level Indicators (SLIs) and Service Level Objectives (SLOs) for their high-level SLAs. Such SLIs and SLOs are the key elements of our service contracts for IoT. A great challenge relevant to the service contract is due to the diversity of the underlying edge/IoT infrastructure. For example, writing a component accessing a power generator status might require different API sets for different power generators and IoT infrastructures. Furthermore, in IoT we access many types of data, which require service contracts to deal with data access rights and data quality constraints. Service contracts must include common performance and execution control constraints in most IoT infrastructures.

Consider an IoT infrastructure-as-a-service for operation and maintenance of Base Transceiver Stations (BTSs) equipment, such as HVAC (Heating, Ventilation and Air Conditioning), electricity power generators, and electricity backup systems. Shown in Figure 1, in each BTS we have an IoT gateway connecting hardware sensors and actuators, which interface such BTS equipment. The maintenance of BTS equipment is outsourced to third-party companies that deploy and run their own IoT software for maintaining equipment based on IoT data analytics and controls at the edge. Such IoT software is built from IoT units and services: a unit is a basic component of which an instance is executed in a single place (e.g. a gateway or an edge server) where as a service is built from one or many instances of units. Such IoT units and services are running in the IoT infrastructures in BTSs.

Consider HVAC which is a critical component for operation of a BTS. The third-party IoT units have to operate properly but may not affect other IoT units (e.g. monitoring other components of the BTS) running on the same IoT gateway. Such IoT units and their compositions are technically IoT (micro)services. From the point of view of the maintenance company, various contractual terms have to be supported:

- access to sensors and actuators required for operation of the HVAC maintenance analytics and control service
- required quality of data delivered by the sensors (e.g., accuracy)
- required availability and responsiveness of the actuators
- required processing power and storage capacity for IoT units running on the IoT gateway

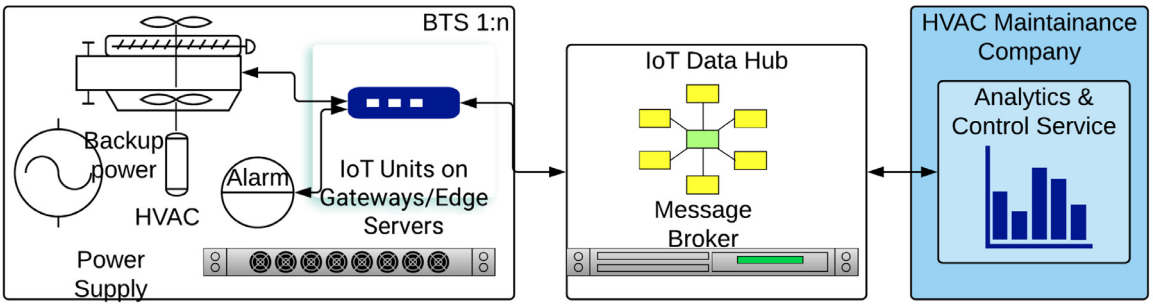

Fig. 1. Monitoring and control equipment in BTS

For the telco operator (the IoT infrastructure provider), the following terms should be defined in the contract:

- restriction of access to sensors and actuators required by the HVAC IoT units
- guaranteed processing power and storage capacity for the HVAC IoT units running on the IoT gateway
- QoS criteria for the HVAC maintenance analytics and control service
- availability of the HVAC maintenance analytics and control service
- price model for running the IoT maintenance units on the IoT gateway

For running such IoT units, major contract requirements are access control, data quality of sensors, availability of sensors, resource management on the IoT gateway and pricing. They are common but it is very challenging to support them through the lifecycle of development and operation of IoT units due to the diversity of IoT. Our main research questions are:

- what are possible activities for engineering contracts and how they are integrated with typical DevOps activities for IoT?
- how to describe and specify contract templates, contract models and contract enforcement for dealing with complexity and diversity of underlying IoT infrastructures?
- what are runtime services and tools for supporting contract development and operation?
- how to manage various artifacts of units and their contracts for IoT to foster secured, verifiable executions in IoT-as-a-service models?

In this work we present our initial contribution towards an integrated approach for engineering contracts for IoT.

3. IoT services and DevOps requirements

3.1. DevOps Contract Stakeholders

For DevOps of IoT microservices in IoT as-an-infrastructure, similar to that for cloud development, we also have a rich set of stakeholders:

- *IoT Service Users:* use services from service providers either for their own needs or to provide composite services, e.g. access to the sensors and actuators of the HVAC equipment. For IoT contracts, the major concerns for IoT service users are *availability, data quality*, and *trust relationships*.
- *IoT Service Providers:* provide IoT services, e.g., HVAC monitoring and maintenance service. In terms of IoT contracts, some major concerns for an IoT service provider are *usage control, authorization* and *payment*.
- *IoT Unit Providers:* provide IoT units with a set of defined functions, e.g. reducing HVAC operation when the outdoor temperature is low. Major IoT contract concerns are *usage control, authorization*, and *payment*.
- *IoT Gateway/Platform and Edge Platform Providers:* allow to run IoT units/services on IoT infrastructures, such as edge servers or IoT gateways. Major contract concerns for platform providers are *resource usage control, resource access control*, and *payment*.
- *IoT Developers*: program contracts and enforcement techniques and perform IoT software development tasks.

All of these stakeholders have many contractual concerns centered around the IoT units, services and infrastructural resources that they require a faster, dynamic interaction and integration w.r.t the development, deployment, operation and testing of IoT contracts in IoT microservices development and operation. However, to date, mainly we have contracts for IoT units specified only at the provisioning time or not integrated into the software development process.

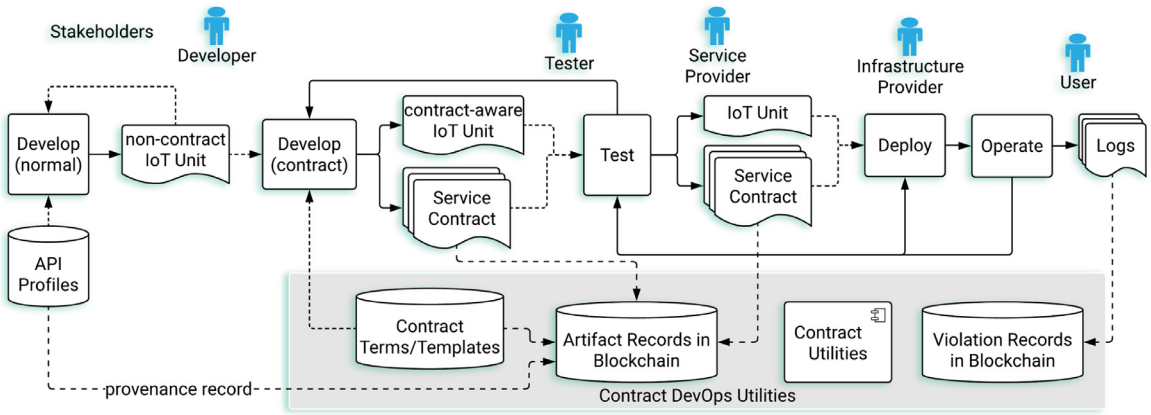

Fig. 2. Stakeholders, activities and artifacts in DevOps for IoT

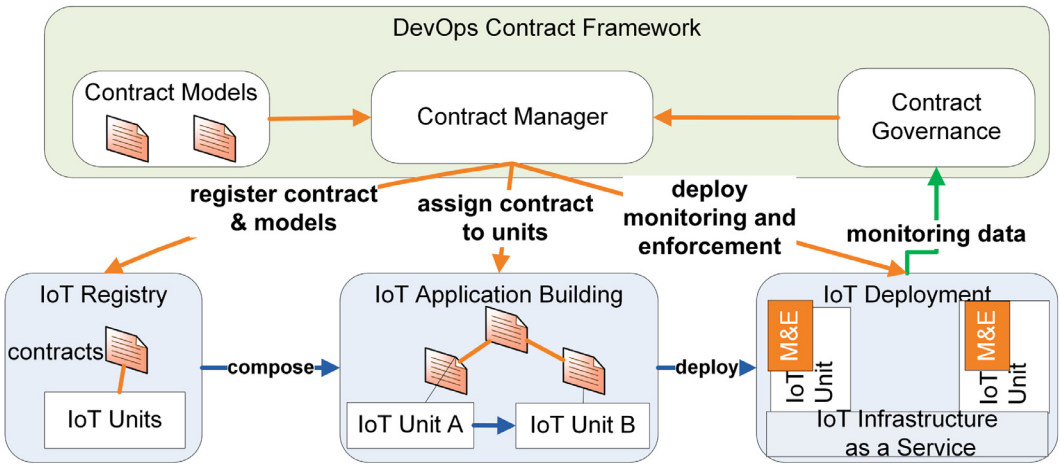

Fig. 3. Overview of IoT DevOps Contract interactions and services

3.2. DevOps Contract Approach

The benefit of DevOps for software assurance has been discussed widely [15]. Similar to DevOps and related Security DevOps[1] or DevSecOps[2], the key principle of our work is to enable the integration of contracts for service quality assurance at the start of the IoT services development, allowing us to test contracts and to change them as quickly as possible when needed, and to automate the deployment of IoT contracts together with IoT services. To this end, we should enable a different way to program and test contracts, instead of sticking to specific language-specific policies at the provisioning time.

Figure 2 presents stakeholders and activities w.r.t. the DevOps contracts in our approach. Following typical activities, the Developer, Service Provider and Infrastructure Provider will share API profiles of IoT infrastructures, develop, test and deploy IoT Units. However, instead of waiting until the operation phase to incorporate service contracts, we foster the development of service contract at an early stage: the Developer can use `Contract Templates` to develop `Service Contract`, which are associated with IoT Unit (`contract-aware IoT Unit`). The service contract for IoT unit can be tested and confirmed by Service Providers and Infrastructure Providers. All of these artifacts are linked and information related to contract can be recorded into, e.g., a blockchain, as evidence artifacts. At runtime the Service, Service Provider and Infrastructure Provider work on specific conditions for contracts and monitor and enforce the contract. Figure 3 presents our key microservices and tools for realizing the DevOps Contract approach.

- development: we provide tools to specific contract elements, such as building blocks and use programming languages to enforce contracts based on low-level languages, runtime, and APIs

[1] http://bit.ly/2ghdbJq
[2] http://bit.ly/2HbyUjg

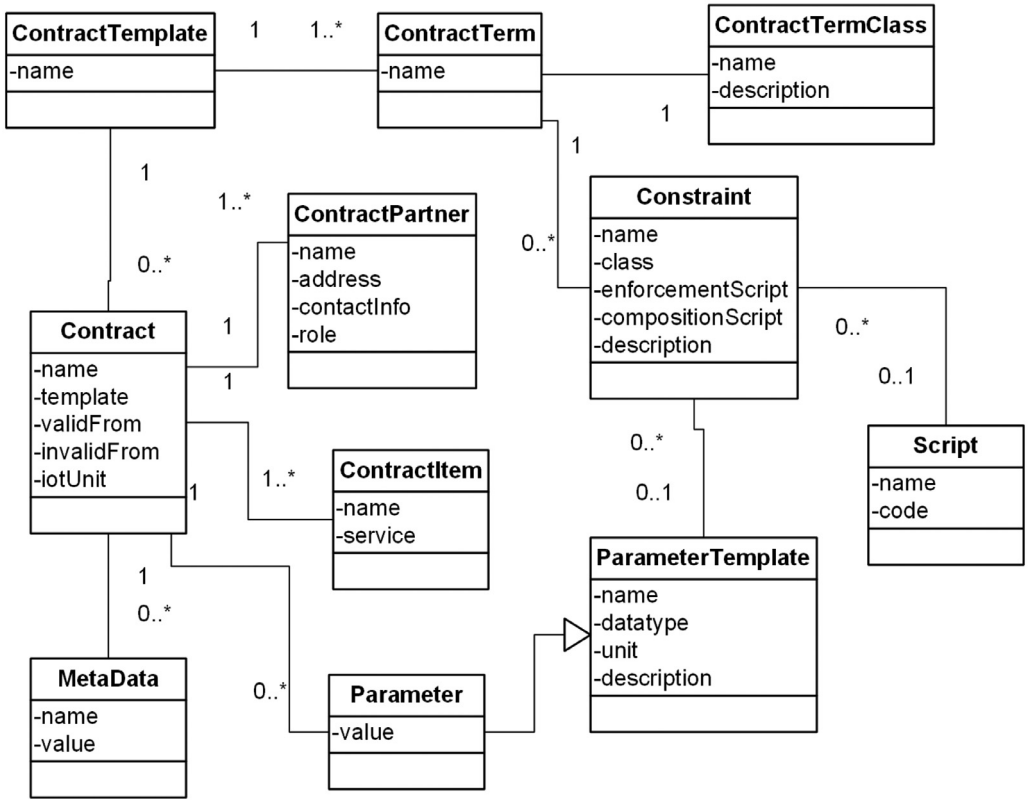

Fig. 4. Main building blocks for IoT contracts

- deployment: we use configuration tools to inject high-level scripts and contracts into IoT services.
- operation: we monitor contracts and log violations as immutable records.
- testing: we allow contract templates, enforcement scripts, etc. available for testing with different IoT devices and infrastructures.

In this view, writing a contract and its enforcement is part of the DevOps process for IoT services. The languages used for the contract development are close to the ones agreed by various stakeholders in the development, testing and operation. Another aspect is that the whole system should alleviate the deployment and engagement of stakeholders in the sense of pay-per-use. This means we need to support no central trust authority and tamper proofs of contracts and their usages. Hence blockchain technologies are useful. To glue teams in Contract DevOps for IoT services, we could use templates for contracts and building blocks of contract elements to enable generic ways of writing contracts. However, for operation, we need enforcement to work with concrete IoT services. Thus enforcement should be easy to change and adapt as well as be tested with various IoT devices and edge servers.

4. Building Blocks for DevOps of Contracts

4.1. Contract Elements for IoT

Based on extensive work on service contracts, such as [16,17], we focus on determining common ways to describe contracts through the DevOps. From previous works on typical contracts, main key building blocks are (i) *Access Rights*, including *Data Access* and *Control Access*, and (ii) *Quality*, including both service quality and data quality, and *Payment*. Therefore, we enable the development of contract terms that cannot be determined in advance by introducing basic building blocks for contract terms. Our approach allows the construction of such building blocks by using common languages in IoT programming. To deal with the complexity of IoT units, stakeholders can write and examine contracts and enforcement scripts during the development and operation. This is different from the use of policies where the enforcement is in place with pre-defined policies at runtime.

Our key building blocks for contracts, shown in Figure 4, include: *Contracts* are built from *Contract Templates* where each contract template holds a set of *Contract Terms* that specify the actual conditions defined for the contract. Contract terms are linked to *Scripts* which contain executable logic defined for runtime environment:

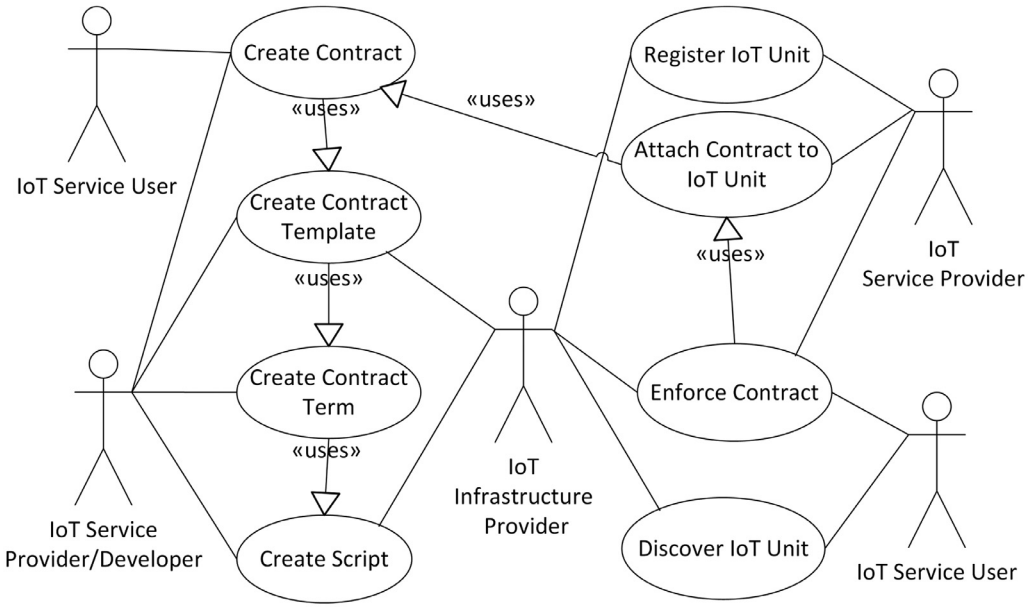

Fig. 5. Interactions in creating contracts and enforcement scripts

- A *generic model* defines the entities required to build contract templates and instances. It holds contracts terms, constraints attached to the terms and parameters required to instantiate the terms and constraints.
- A *contract template* is built using the generic model. It defines common terms of a contract, e.g., that access rights are used in the contract or that a throughput limit is applied.
- A *contract* is built on top of the contract template defining concrete values of the contract, e.g. throughput max. 100 KB / hour, as well as contract partners and contract items which bind the contract to a specific set of IoT units and their services.
- A *Script* is used to enforce contract terms. Instead of using pre-defined libraries to enforce contract terms, we use code scripts. This allows us to solve the dependency of enforcement logic on the actual implementation of IoT services. This can be considered as a consequence of the diversity of IoT but also as a rationale for enabling flexible contract terms. Scripts are provided as a template during DevOps process that stakeholders can see and test at the early stage of the DevOps. In a script, we have variables referring to constraints in contract terms that will be replaced with the value defined in the contract when we deploy contracts IoT unit. Scripts are injected to IoT units and executed on the unit to enforce contract terms. Figure 5 presents interactions in creating contracts and enforcement scripts.

4.2. Contract Template and Contract Implementation

We use JSON (Javascript Object Notation) to represent contract terms and contracts, and store them in a contract registry. A sample contract for the temperature sensing IoT unit in JSON is shown in Listing 1. Using this way, we build various contract templates and contracts. Such templates and contracts can be examined and tested for suitable IoT units.

In Listing 1, contractual constraints are characterized by variables, such as `NrOfReads`. Such a variable will be instantiated during testing or deployment time, allowing us to enforce the contract. For example, for a particular test/deployment we could set `NrOfReads=100`. Given the contract and the parameterized constraints at testing/deployment time, corresponding enforcement scripts (see Section 4.3) will perform the enforcement.

For an IoT microservice built from various IoT units, one can compose different terms and contracts for individual units to create a suitable contract for the mircoservice. For example, a simple composite service for controlling the HVAC can have a contract listed in Listing 2. The main difference from a contract for a single unit is that we can compose different contract terms for individual units.

As we use JSON to specify contract terms and contracts, which are available in the registry, one can consider such terms/-contracts as artifacts associated with IoT services, e.g., like configuration information for software components. Therefore, the developer and relevant stakeholders can bundle, modify and test such contracts together with IoT units/services during DevOps phases. Furthermore, using JSON based contract templates we can develop inspection utilities that can be integrated with continuous integration and continuous delivery (CI/CD) pipelines.

```
{ "name" : "TemperatureSensingContract",
 "ContractItem" : "TemperatureSensingService",
 "ContractPartners" :{"Provider" : "telco, "User" : "HVAC"},
 "ContractTerms" : [
    "AccessRights" :
      {"name" : "TempSensor",
       "constraint" : { "name" : "NrOfReads", "description"
           : "< 1/min" }
      },
    "ServiceQuality" :
      {"name" : "Availability",
        "constraint" : { "name" : "Average", "description" :
            "99.9 %" }
      },
    "Payment" :
      {"name" : "UsageFee",
       "constraint" : { "name" : "Price", "description" : "1
           c/read" }
      }
  ]}
```

Listing 1. Temperature Sensing IoT Contract

4.3. Development of contract enforcement

4.3.1. Script language for enforcement and testing

Contract enforcement logic is dependent on the type of constraints in contracts and on the concrete implementation of the corresponding functionality in the IoT microservices and underlying infrastructures. For example, to check access rights in an IoT unit we have to know how accesses in the IoT unit are handled and which access capabilities are provided by the infrastructure. In terms of DevOps Contract, we have considered following alternatives:

- Direct implementation of enforcement logic in the IoT unit. This would imply that a framework is only applicable to IoT units containing the enforcement logic and it is restricted to the type of constraints handled by the predefined logic.
- Provide a library of enforcement logic in the contract framework. This would also imply that the type of constraints that can be used in the contract is limited and it is not possible to add contract terms with arbitrary constraints.
- Develop a generic language for definition of enforcement logic. This implies that each IoT unit has to implement execution of the logic written in the own language and the runtime has to work with various underlying infrastructures.
- Use a well-known language for definition of enforcement logic and provide an interpreter for execution of the logic on the IoT unit.

To stay with DevOps of IoT microservices, we use programmable scripts to execute enforcement logic. This means one does not have to learn new language and can exploit existing features of underlying systems to enforce the contracts. On the other hand, it means that we need to understand such features and use tools to combine contracts with IoT units/services. To demonstrate our approach, we use Javascript to define the enforcement logic and use the open source Rhino Javascript interpreter to execute them on the IoT unit. Enforcement code is injected into IoT units automatically using Aspect-oriented Programming during the deployment and provisioning.

4.3.2. Enforcement script development

The development of enforcement of contracts will be under the responsibility of the developer and the provider, while the IoT infrastructure provider will clarify information about underlying APIs that can be used to obtain values

```
{ "name" : "HVACControllerContract",
  "ContractItem" : "HVACService",
  "ContractPartners":{"Provider" : "telco, "User" : "HVAC"},
  "ContractTerms" : [
    "ServiceQuality" : {
      "name" : "Availability",
      "constraint" : {
        "name" : "Average",
       "description" : "@ComposedBy (
           TemperatureSensingContract,
           AirConditionControllerContract)"
      }
   },
  "Payment" :
    {"name" : "UsageFee",
     "constraint" : { "name" : "Price", "description" : "5
         EUR/month" }
    }
  ]}
```

Listing 2. Example of a contract for an HVAC Controller

for the enforcement. Using DevOps methodologies, these stakeholders could collaborate together in defining, testing and deploying enforcement scripts. Similar to the development of contracts, enforcement code will be provided as templates and enforcement scripts can be reused, modified and combined. For example, considering the contract term for `Access Right` for sensor data shown in Listing 3, an enforcement script `AccessRightCheckWithTime` can be as follows:

```
var sensorID = @SensorID;
var from = @From;
var to = @To;
var ts = Date.now();
if ((!Boolean(dataPoint.getName() == SensorID)) ||
    (ts < from) ||
    (ts > to)) {
    _reason='ABORT';
    _log='access not allowed';
}
```

The values starting with `@`, e.g., `@To`, can be replaced with concrete values of the constraint parameters when the enforcement script is loaded to the IoT unit at the testing or production time. The variables starting with `_`, e.g., `_log`, capture information sent to the governance controller when a contract violation is detected.

```
{
   "name": "BTSServerRoomTemperatureAccess",
   "type": "AccessRight",
    "constraints": [ {
        "name": "SensorAccessAtTime",
        "enforcementScript": "AccessRightCheckWithTime",
        "description": "check access to the sensor at time",
        "parameters": [ {
            "name": "SensorID",
            "datatype": "String"
        }, {
            "name": "From",
            "datatype": "Date"
        }, {
            "name": "To",
            "datatype": "Date"
        }]
    }]
}
```

Listing 3. BTS Contract Term for Time based Access Rights

Another example is about the enforcement of `timelineness` for data. The timelineness of data can be determined from event timestamps received in the data and the result is then compared to the contract terms as shown in the following:

```
var requiredTimeliness = @RequiredTimeliness;
var ts = Date.now();
if ((ts - datapoint.getTimestamp()) > requiredTimeliness){
  _reason='NOTIFY';
  _log='timeliness violated'; }
```

for determining *accuracy* of data which is determined using meta-data and to compare with the contract terms, the following simple code can be used:

```
var requiredAccuracy = @RequiredAccuracy;
var ts = Date.now();
if (datapoint.getAccuracy() < requiredAccuracy){
_reason='NOTIFY';
_log='accuracy violated at '+ts; }
```

`@RequiredAccuracy` will be replaced by the real value in the contract at runtime. In the example, the underlying IoT infrastructure provides `datapoint.getAccuracy()` as an API to determine the accuracy of the data obtained from a `datapoint`, whereas `datapoint` is a data stream accessing data from the infrastructure.

Using the above-mentioned approach, we have contract terms and constraints are independent of a concrete IoT unit. The enforcement scripts are generic but they need to be instantiated with concrete values for concrete IoT units through the

instrumentation. Everything is described through common languages for configuration and programming so that through the DevOps phases, stakeholders can easily test and correct mistakes, and adapt terms and scripts.

Following the above-mentioned examples and methods, stakeholders can develop several building blocks for enforcing other terms, such as for data completeness, data volume, and payment.

4.4. Recording information about contracts

In the context of DevOps, contracts, APIs and IoT units can be considered assets that can be created and transferred. Information about various artifacts in Figure 2, contracts, and enforcement scripts can be recorded into blockchain systems as assets. Such assets can be verified and transferred by the Developer, Service Provider and Infrastructure Provider. The reason is to ensure that information about contracts have been validated in different DevOps phases in open IoT infrastructure and software ecosystems.

We define an asset of the DevOps Contract to be recorded as *devopscontract asset – devopsca*. A *devopsca* includes metadata about (i) IoT Infrastructure-as-a-service and its APIs, (ii) IoT units, (iii) contract templates, (iv) contracts, and (iv) enforcement scripts. Note that we only put the metadata as immutable records for verification purpose. It is possible to store contracts and scripts as assets into a blockchain system as well but we currently do not, due to the tight association between contracts and IoT units development. Listing 4 shows the structure of *devopsca*.

A *devopsca* can be initially created by the Developer and can be verified and transferred between the Developer, Service Provider and Infrastructure Provider. Various other operators can also be applied. In order to implement this, an asset-based blockchain would be suitable. To this end, we use BigChainDB[3] in our implementation as it is designed for registering and transfers assets. The following code excerpt shows an example of using BigChainDB to store a *devopsca*:

```
const assetdata = JSON.parse(fs.readFileSync(inputfile, '
    utf8'));
const metadata = {'type': 'devopscontract'}
// the developer adds a new asset about contract information
const txCreateDeveloperSimple = driver.Transaction.
    makeCreateTransaction(
    //information about a devopscontract asset
    assetdata,
    metadata,
    //the condition for transfer is that the developer must
        verify
    [ driver.Transaction.makeOutput(
      driver.Transaction.makeEd25519Condition(developer.
          publicKey))
    ],
    developer.publicKey
)
//the developer signs and sends the devopscontract asset
const txCreateDeveloperSimpleSigned = driver.Transaction.
    signTransaction(txCreateDeveloperSimple, developer.
    privateKey)
// Send the transaction to the blockchain
const conn = new driver.Connection(API_PATH)
conn.postTransactionCommit(txCreateDeveloperSimpleSigned)
```

[3] https://www.bigchaindb.com/

```
{"devopscontract_asset": {
  "iot_iaas": {
    "description": "information about infrastructures",
    "api_profile": "link to information about API profile"
  },
  "iot_units": [
    { "id":"id of the unit",
      "iot_unit_name":"name",
      "iot_unit_hash":"hash of the  unit",
      "iot_unit_code":"link to the code of IoT unit in the
          repository"
    } ],
  "contract_templates": [
    {
      "id":"template id",
      "name":"name of the template",
      "base_template":"the base template",
      "content":{
          "template_hash":"hash of the template",
          "template_uri" :"link to source of the template"
        }
   } ],
  "contract": {
    "id":"contract id",
    "name":"name of the contract",
    "content":{
        "contract_hash":"hash of the template",
        "contract_uri" :"link to source of the contract"
      }
 } ,
  "enforcementscript": {
    "id":"enforcement script id",
    "name":"short name",
    "content":{
      "script_hash":"hash of the script",
      "script_uri" :"link to source of the script"
    }}
}
```


Listing 4. Structure of a DevOps contract asset

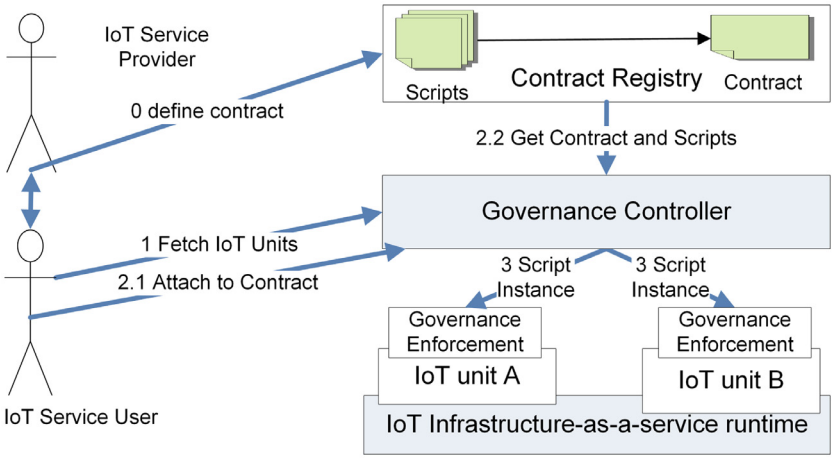

Fig. 6. Services for contract governance

5. DevOps Contract Enforcement

5.1. Contract governance implementation

DevOps Contract approach must also include a set of governance services to enable fast integration, deployment and testing of contracts. Shown in Figure 6, *Contract Registry* holding contracts, contract terms, contract templates and scripts; essentially, it is similar to a registry for containers. *Governance Controller* retrieves contract information and manages attachment of contracts to IoT units. It enables the automation of deployment of contracts and IoT services. *Governance Enforcement* which retrieves enforcement scripts from the governance controller and executes them locally on the IoT unit. Figure 7 shows sequences of their interactions.

5.1.1. Governance Controller

To enable runtime flexible deployment and governance of IoT contracts, our governance controller retrieves enforcement scripts available and execute scripts within IoT units, store contract violations and execute payments. When a contract is attached, contract enforcement will be loaded from the IoT contract registry and placeholders of parameters in the script code are replaced with actual values from the contract according to the pseudo-code in Figure 8.

5.1.2. Integration contract violations with blockchain

A key of DevOps Contract is to leverage existing technologies to record contract violations at runtime in a way that records are immutable and verifiable by stakeholders. This is similar to record contract assets but for violations at the operation phase. We leverage blockchain to store a fingerprint of the IoT contract violation information. When a contract violation is reported, a transaction capturing the violation information is performed on the blockchain and all stakeholders can inspect the blockchain and verify that the transaction occurred. This integration is suitable for managing IoT contracts of large-scale IoT infrastructure-as-a-service and fits into scenarios of using (private) blockchain for auditing IoT data sharing.

We prototype our work with Ethereum[4]. Shown in Figure 9, first, when the contract is attached to the IoT unit by the governance controller, contract data and scripts are fetched from the contract repository and a blockchain's "smart contract"[5] is created. It contains the account id's of the partners and a fingerprint of the contract and its enforcement scripts. When a contract violation is observed the governance controller sends a message containing a fingerprint of the log to the smart contract. Any partner can then verify the logs recorded by the governance controller using the fingerprint. Any partner can then verify the contract in the repository using the fingerprint as well as to trace back DevOps contract assets (discussed in Section 4.4).

6. Implementation and Evaluation

6.1. Prototype

The contract data model is integrated into a framework for dynamic configuration of IoT cloud services. Entities are stored by Neo4J database and using REST API we can access to IoT contract artifacts[6]. We provide web services interface to attach a contract to an IoT unit, build the concrete enforcement scripts based on the contract and make them available for download

[4] https://www.ethereum.org
[5] One should not be confused the "service contract" with "smart contract" in blockchain as they are two different types of contract.
[6] https://github.com/rdsea/IoTContract

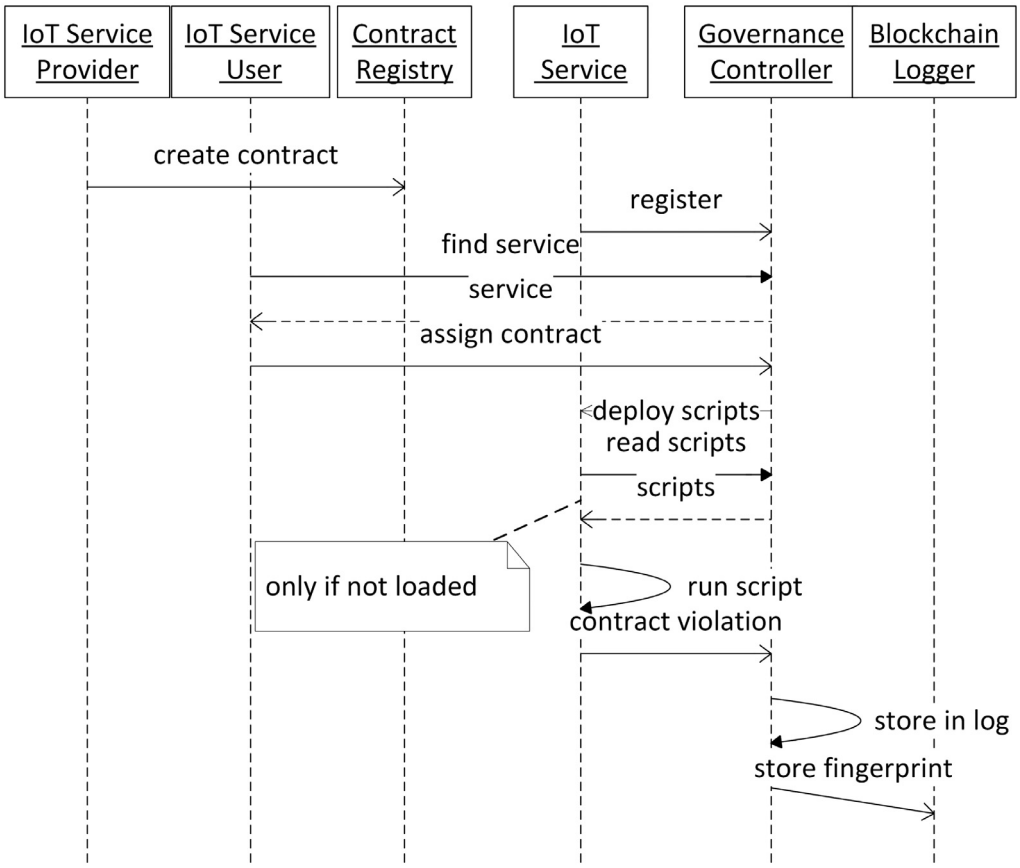

Fig. 7. Lifecycle interactions

```
ServiceTemplate = fetchServiceTemplate(Unit.ServiceTemplate)
ContractTemplate = fetchContractTemplate(ServiceTemplate)
FOR EACH ContractTerm IN ContractTemplate
  ContractTerm = fetchContractTerm(ContractTerm)
  FOR EACH Constraint IN ContractTerm
    Script = fetchScript(ContractTerm.Script)
    FOR EACH Parameter IN Constraint.Parameters
      replace placeholder in Script with actual contract value
    END FOR
  END FOR
END FOR
```

Fig. 8. Replace placeholders in contract templates

by the IoT unit, serve as a registry for IoT units and handle contract violation messaging and logging. In the prototype IoT units are treated as a white box and they are assumed to be written in the Java programming language. Instrumentation is based on AspectJ for injecting monitoring and enforcement into IoT units. Enforcement scripts are written in JavaScript and executed by the Rhino execution engine.

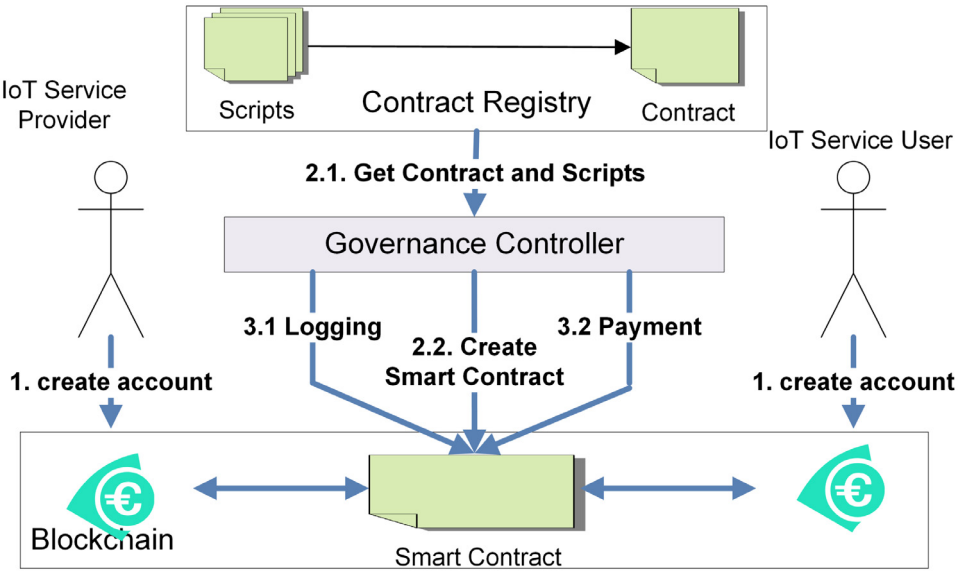

Fig. 9. Logging and payment via Blockchain

Table 1
Different number of events

nr. events	script loading time (s)	execution time (s)	events/s
10	1.38	0.79	12.66
100	1.38	4.13	24.21
1000	1.35	30.32	32.98
2000	1.38	59.16	33.81

6.2. Test System Setup

Our work is mainly about the DevOps contract approach which includes the activities and tools for contract. Therefore, the performance runtime evaluation is just a part of the approach to illustrate tools implementing the approach. In our experiments, we mainly measure script loading time, execution time and memory footprints. We actually do not have a baseline for such criteria, e.g., to which levels they are acceptable for the user. The purpose of testing and using these conventional criteria are typically for tesing the development framework in actions. In our future work such criteria could be established, e.g., based on the acceptance of the users.

For runtime evaluation we took a real-world sample dataset from the monitoring of BTS. The sample data covers alarms and status messages for a set of BTSs, each covering a set of data points[7]. We use the data of one BTS as input for an IoT emulator as csv file in the same format. Data contained in the sample is BTS `id`, `data point id`, `time-stamp` and `value`. We also use a humidity sensor being connected to a sensor IoT unit running on a small low power IoT device. The device uses wireless communication to a Raspberry Pi that works as an IoT gateway translating from the CoAP protocol on the wireless side to REST web services sent to a cloud based IoT platform. Figure 10 shows our test system. Since our goal is to examine the software developments and tools, we have not established a powerful testbed of large scale of IoT microservices.

6.3. Runtime Evaluation

6.3.1. Influence of different number of events processed

In the first set of experiments we evaluate the influence of different number of events processed. Table 1 shows that throughput (number of events per second) stays the same from 1000 events on so that any effects of startup processing can be neglected. We use 1000 events to process for the other experiments.

6.3.2. Influence of number of running units

We run IoT units on 4 different Raspberry Pi machines, first 1 unit on one machine, then 4 units spread over 4 different machines, and then 8 and 12 units spread over 4 machines. Table 2 shows that throughput increases nearly linear with

[7] https://github.com/rdsea/IoTCloudSamples/tree/master/data/bts

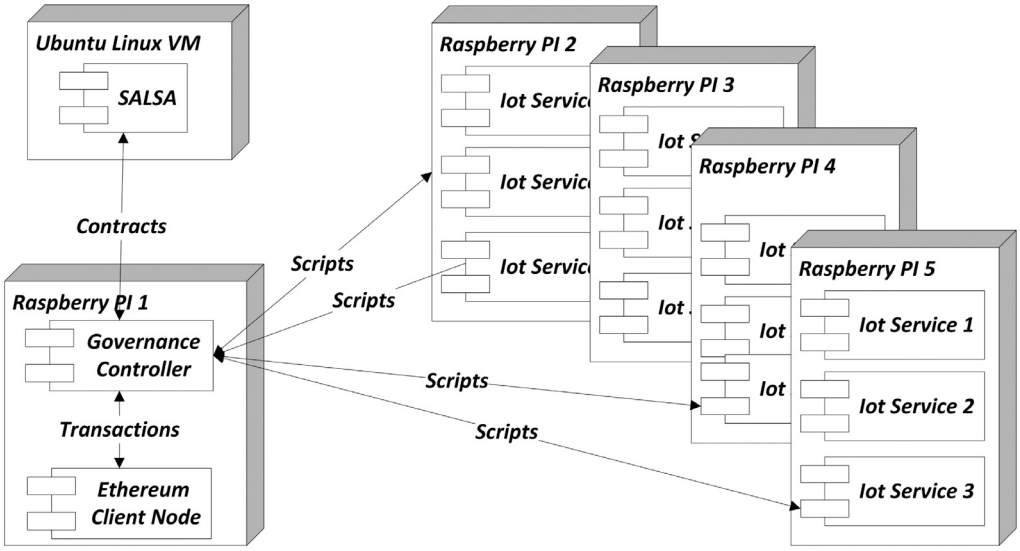

Fig. 10. Deployment for evaluation with IoT simulator

Table 2
Different number of units

nr. of units	script loading time (s)	execution time (s)	events/s
1	1.35	30.32	33.81
4	1.65	29.92	133.69
8	2.55	33.76	236.97
12	3.26	39.02	307.53

Table 3
Different types of contracts

nr. constraints	nr. contract terms	nr. customers	script loading time (s)	execution time (s)	events/s
1	1	1	1.35	30.32	32.98
10	1	1	1.51	91.15	10.97
1	10	1	1.51	88.47	11.3
1	1	10	1.53	90.16	11.09

the parallel execution on IoT units. Processing of enforcement is executed in parallel in different IoT units. Processing of messages on the governance controller also makes use of parallel execution depending on available processing capacity. Increased script loading time (measured as from starting of script loading on the first unit of the first machine to finishing script loading on the last unit of the last machine) increases because not all units are started on the Raspberry Pi at the same time, especially if more units are run on the same machine. The same applies to execution time (also measured from starting event processing on the first unit of the first machine to finishing on the last unit of the last machine) slightly increasing although throughput on one machine stays the same. In general, the system scales well (approximately tenfold increase in throughput from 1 - 12 units, even when the governance controller is running on a Raspberry Pi) and from architectural point of view it should be able to handle a large number of units.

6.3.3. Influence of different contracts

We also tested with a different number of constraints, of terms per contract and of contracts per IoT unit. Table 3 shows that execution time increases linear with the number of constraints, contract terms and contracts per unit. For a tenfold increase in the number of scripts from 1 to 10, we see an approximately 3 times reduction in throughput. We see that, with a large number of constraints, we still achieve reasonable performance.

6.3.4. Influence of enforcement technologies

Table 4 shows that enforcement with blockchain adds considerable performance penalties. Clearly, only important contract enforcement messages should be logged to the blockchain to avoid performance problems. Table 5 analyzes the code size for the monitoring part added to the IoT simulator and the gateway and analyzing the code size for the access right monitoring script. The size of the monitoring and enforcement component and of the enforcement scripts are rather small and have no substantial influence on the size of the contract-aware IoT unit.

Table 4
Enforcement performance via Blockchain

nr. of events	blockchain	script loading time (sec)	execution time (sec)	events per sec
100	No	1.38	4.13	24.21
1000	No	1.35	30.32	32.98
100	Yes	1.45	15,47	6.46
1000	Yes	1.37	112.77	8.48

Table 5
Monitoring and enforcement code size

	size (characters)	file size (KB)
Monitor IoT simulator	7716	16
Monitor gateway	7818	15
Access Right Script	122	n/a
quality of data check script	218	n/a

7. Related Work

Service contract terms and constraints are known in literature [11,14,18,19], however, the development, testing and deployment of such terms have not been seen throughout the entire of DevOps of IoT units and IoT services. On the other hand, existing works, like [20], concentrate on safe, secure features of IoT within programming languages. There are gaps and inflexibility with the current approaches for IoT service contracts. As we discussed in the introduction, we can see the related work through two directions. In the first direction, various works present different contracts models. In [21] a model for the definition of security policies for MQTT [22] is presented. It includes entities such as data, time, identity, role, behavior, trust and risk, rule templates and rules. They are more from the perspective of access authorization in using services through remote APIs.

The other approach is to control the execution of IoT units/components in gateways. In [10] access control is based on device functions. Other works on IoT assume the IoT units developed atop software defined machines – a kind of APIs abstracting low-level, diverse types of APIs. Still, the policy specifying service contracts is not developed during the development of IoT units but at the deployment time and the logic of enforcement is fixed by the system. This leads to many problems when dealing with different underlying machines and new types of contract terms. In [23] a capability based approach to access control is presented. Adinolfi et al. in [24] describe the QoS-MONaaS for monitoring of QoS in IoT and cloud applications. In [25] a privacy enforcement module is proposed for fog node. However, it is not associated with IoT units individually; it is applied for system-wide. In principle, such a module can be integrated into DevOps processes. Guth, et al. in [16] described a contract and rights management framework using machine learning based modeling, but not investigated the software engineering of contracts and IoT.

In general, we have not seen the development of contracts in parallel with IoT units. Current approaches do not support well changes due to new types of contract terms, new devices and new infrastructures. Our approach is to work on IoT units and contracts at the same time. On the other hand, various contract terms have been proposed, so our work will focus on how to map such terms into programmable scripts and configuration for IoT contract development.

8. Conclusions and Future Work

As we discussed in Section 2, developers for IoT microservices are faced with several questions on how to engineer service contracts and to enforce them during the development of microservices deployed in IoT infrastructure-as-a-service. We advocate the DevOps Contract for IoT microservices as a means for supporting quality of IoT microservices in IoT infrastructure-as-a-service. We have presented basic building blocks of contracts and enforcement scripts during the development of IoT units and services. Our languages and methods are based on common ones used in IoT development, like JSON and JavaScript. Many types of contractual terms, like access to data, quality of data, QoS and price, have been developed. We have demonstrated them with telcos use cases. Major issues in describe and specify contracts and templates have been discussed and presented. We also show services enabling storage and automatic deployment of contracts. They support various activities in engineering service contracts for IoT microservices and providing core functions for runtime services and tools. Since languages and utilities for contracts are close to typical flows of DevOps but for IoT, it is easy to adapt, change, and fix contract-related issues for IoT software services. We have presented our approach and tools addressing our questions in Section 2 through a predictive maintenance use case for telcos infrastructures.

The DevOps Contract can be language independent. For example, one can select suitable languages and tools for enforcement scripts and IoT units, while contract building blocks in JSON can be the same. However, it is quite challenging to support this generalization. Our approach needs to be improved and verified with different IoT infrastructure-as-a-services. In our future work, we will improve the methods and also verify if the programming of contracts creates mistakes. Penalty/compensation will be considered in contract enforcement in the future.

Another important aspect is that, as expected, the number of IoT devices and their capabilities will be increased for a given IoT infrastructure-as-a-service. This leads to three different situations related to our work in the future. First, it is possible that the IoT microservices in our DevOps Contract approach will not be increased or changed (e.g., due to the interest of the services for a particular type of data), but the contracts might need to be changed for dealing with different constraints related to quality, quantitative and time, e.g. for data points and control points. The second situation is that new IoT microservices will be developed to exploit new IoT device capabilities. This might require further extension of service contracts as well as enforcement scripts. For this to be successful, IoT devices, API profiles and related execution environment information might need to be automatically discovered and collected for the development of suitable contracts. Third, new models of microservices will be also exploited, e.g., a combination of serverless (function-as-a-service) with light-weighted containers exposed their APIs via known protocols like REST and MQTT. While, contract specifications might not be very different, enforcement and integration will be challenging. Currently, we focus more on the first and the third situations.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work in this paper was partially carried out during Peter Klein master thesis at TU Wien, Austria. Results are also partially reported in his master thesis.

References

[1] Y. Benazzouz, C. Munilla, O. Gunalp, M. Gallissot, L. Gurgen, Sharing user iot devices in the cloud, in: Internet of Things (WF-IoT), 2014 IEEE World Forum on, 2014, pp. 373–374, doi:10.1109/WF-IoT.2014.6803193.
[2] M. Comuzzi, B. Pernici, A framework for qos-based web service contracting, ACM Trans. Web 3 (3) (2009), doi:10.1145/1541822.1541825. 10:1–10:52
[3] D. Muthukumaran, D. O'Keeffe, C. Priebe, D. Eyers, B. Shand, P. Pietzuch, Flowwatcher: Defending against data disclosure vulnerabilities in web applications, in: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15, ACM, New York, NY, USA, 2015, pp. 603–615, doi:10.1145/2810103.2813639.
[4] E. Yuan, J. Tong, Attributed based access control (abac) for web services, in: Proceedings of the IEEE International Conference on Web Services, ICWS '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 561–569, doi:10.1109/ICWS.2005.25.
[5] C. Pahl, B. Lee, Containers and clusters for edge cloud architectures – a technology review, in: 2015 3rd International Conference on Future Internet of Things and Cloud, 2015, pp. 379–386, doi:10.1109/FiCloud.2015.35.
[6] J. Moore, G. Kortuem, A. Smith, N. Chowdhury, J. Cavero, D. Gooch, Devops for the urban iot, in: Proceedings of the Second International Conference on IoT in Urban Space, Urb-IoT '16, ACM, New York, NY, USA, 2016, pp. 78–81, doi:10.1145/2962735.2962747.
[7] A.M. Haubenwaller, K. Vandikas, Computations on the edge in the internet of things, Procedia Computer Science 52 (2015) 29–34.
[8] M. Satyanarayanan, Cloudlets: At the leading edge of cloud-mobile convergence, in: Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures, QoSA '13, ACM, New York, NY, USA, 2013, pp. 1–2.
[9] A. Ciuffoletti, Occi-iot: an api to deploy and operate an iot infrastructure, IEEE Internet Things. 4 (5) (2017) 1341–1348, doi:10.1109/JIOT.2017.2734068.
[10] S. Lee, J. Choi, J. Kim, B. Cho, S. Lee, H. Kim, J. Kim, Fact: Functionality-centric access control system for iot programming frameworks, in: Proceedings of the 22Nd ACM on Symposium on Access Control Models and Technologies, in: SACMAT '17 Abstracts, ACM, New York, NY, USA, 2017, pp. 43–54.
[11] B. Beyer, C. Jones, J. Petoff, N.R. Murphy, Site Reliability Engineering: How Google Runs Production Systems, 1st edition, O'Reilly Media, Inc., 2016.
[12] C. Ebert, G. Gallardo, J. Hernantes, N. Serrano, Devops, IEEE Softw. 33 (3) (2016) 94–100, doi:10.1109/MS.2016.68.
[13] I. Jacobson, I. Spence, P.-W. Ng, Is there a single method for the internet of things? Queue 15 (3) (2017), doi:10.1145/3121437.3123501. 20:25–20:51
[14] P. Wieder, J.M. Butler, W. Theilmann, R. Yahyapour, Service Level Agreements for Cloud Computing, Springer Publishing Company, Incorporated, 2011.
[15] M.G. Jaatun, D.S. Cruzes, J. Luna, Devops for better software security in the cloud invited paper, in: Proceedings of the 12th International Conference on Availability, Reliability and Security, in: ARES '17, ACM, New York, NY, USA, 2017, doi:10.1145/3098954.3103172. 69:1–69:6
[16] S. Guth, B. Simon, U. Zdun, A contract and rights management framework design for interacting brokers, in: 36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the, 2003, p. 10, doi:10.1109/HICSS.2003.1174818.
[17] P.R. Krishna, K. Karlapalem, Electronic contracts, IEEE Internet Computing 12 (4) (2008) 60–68.
[18] A. Karkouch, H. Mousannif, H. Al Moatassime, T. Noel, Data quality in internet of things, J. Netw. Comput. Appl. 73 (C) (2016) 57–81, doi:10.1016/j.jnca.2016.08.002.
[19] Odrl community group, (https://www.w3.org/community/odrl). Accessed: 2017-12-16.
[20] C.-J.M. Liang, B.F. Karlsson, N.D. Lane, F. Zhao, J. Zhang, Z. Pan, Z. Li, Y. Yu, Sift: Building an internet of safe things, in: Proceedings of the 14th International Conference on Information Processing in Sensor Networks, IPSN '15, ACM, New York, NY, USA, 2015, pp. 298–309, doi:10.1145/2737095.2737115.
[21] R. Neisse, G. Steri, G. Baldini, Enforcement of security policy rules for the internet of things, in: 2014 IEEE 10th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), 2014, pp. 165–172, doi:10.1109/WiMOB.2014.6962166.
[22] MQTT Message Queuing Telemetry Transport, http://mqtt.org Accessed: 2018-13-01.
[23] S. Gusmeroli, S. Piccione, D. Rotondi, Iot access control issues: A capability based approach, in: 2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, 2012, pp. 787–792, doi:10.1109/IMIS.2012.38.
[24] O. Adinolfi, R. Cristaldi, L. Coppolino, L. Romano, Qos-monaas: A portable architecture for qos monitoring in the cloud, in: 2012 Eighth International Conference on Signal Image Technology and Internet Based Systems, 2012, pp. 527–532, doi:10.1109/SITIS.2012.82.
[25] A. Al-Hasnawi, L. Lilien, Pushing data privacy control to the edge in iot using policy enforcement fog module, in: Companion Proceedings of the10th International Conference on Utility and Cloud Computing, UCC '17 Companion, ACM, New York, NY, USA, 2017, pp. 145–150, doi:10.1145/3147234.3148124.