rtGovOps: A Runtime Framework for Governance in Large-scale Software-defined IoT Cloud Systems

Stefan Nastic, Michael Vögler, Christian Inzinger, Hong-Linh Truong, and Schahram Dustdar
Distributed Systems Group, Vienna University of Technology, Austria
Email: {*lastname*}@*dsg.tuwien.ac.at*

***Abstract*—The ongoing convergence of cloud computing and the IoT gives rise to the proliferation of diverse, large-scale IoT and mobile cloud systems. Such novel IoT cloud systems offer numerous advantages for all involved stakeholders. However, due to scale, complexity, and inherent geographical distribution of such systems, governing new IoT cloud resources poses numerous challenges. In this paper we introduce rtGovOps, a novel framework for on-demand runtime operational governance of software-defined IoT cloud systems. To illustrate the feasibility of our framework and its practical applicability to implement and execute *operational governance processes* in large-scale software-defined IoT cloud system, we evaluate our approach using a real-world case study on managing fleets of electric vehicles.**

I. Introduction

Current advances in the Internet of Things (IoT) and mobile cloud computing research have enabled the creation of unified IoT and mobile cloud infrastructures [1]–[5] that offer large pools of IoT cloud resources. Recently, software-defined IoT cloud systems have been introduced [6] to abstract from low-level resources (i.e., hardware) and enable their programmatic management through well-defined APIs. This enables refactoring the underlying IoT cloud infrastructure into finer-grained resource components whose functionality can be (re)defined after they have been deployed. While such systems create numerous opportunities by exploiting novel IoT and mobile cloud resources, they also introduce a number of challenges not previously encountered in traditional systems, to operate and govern such resources at runtime. Unfortunately, traditional governance approaches are hardly applicable for IoT cloud systems, mainly due to their dynamicity, heterogeneity, geographical distribution, and large scale. Supporting tools and mechanisms for runtime operational governance of IoT cloud systems remain largely undeveloped, thus placing much of the burden on operations managers to perform operational governance processes.

This calls for a systematic approach to govern IoT cloud resources throughout their entire lifecycle. In our previous work [7], we introduced the GovOps methodology to effectively manage runtime governance in software-defined IoT cloud systems. The main purpose of GovOps is to close the gap between high-level governance objectives (e.g., costs, legal issues or compliance) and underlying operations processes that support such objectives. Therefore, GovOps mostly focuses on designing and realizing *operational governance processes* (e.g., similar to [8], [9]), which represent a subset of the overall IoT cloud governance and incorporate relevant aspects of both high-level governance strategies and underlying operations management. Continuing along this line of research, in this paper, we introduce the rtGovOps framework for dynamic, on-demand operational governance of software-defined IoT cloud systems at runtime. The rtGovOps framework provides runtime mechanisms and enabling techniques to reduce the complexity of IoT cloud operational governance, thus enabling operations managers to perform custom operational governance processes more efficiently in large-scale IoT cloud systems.

The remainder of the paper is structured as follows: Section II presents a motivating case study and summarizes our background work; Section III outlines main concepts and the design of the rtGovOps framework; In Section IV, we explain major runtime mechanisms of rtGovOps; Section V describes preliminary experimental results and outlines the current prototype implementation; Section VI discusses related work; Finally, Section VII concludes the paper and gives an outlook of our future research.

II. Motivation and background

A. Scenario

Let us consider a realistic application scenario in the domain of vehicle management that we will refer to throughout the paper. This scenario is based on an ongoing collaboration with industry partners[1] and our current effort in software-defined IoT cloud systems.

The Fleet Management System (FMS) is a real-world software-defined IoT cloud system responsible for managing fleets of zero-emission, electric vehicles deployed worldwide, e.g., on different golf courses. Vehicles communicate with the cloud via 3G or Wi-Fi networks to exchange telematic and diagnostic data. On the cloud, FMS provides different applications and services to manage this data. Relevant services include realtime vehicle status, remote diagnostics, and remote control. The FMS is currently used by the following three types of stakeholders: vehicle manufacturers, distributors, and golf course managers. These stakeholders have different business models. For example, when a manufacturer only leases vehicles to customers, they are interested in the status and upkeep of the complete fleet, will perform regular maintenance, as well as monitor crashes and battery health. Golf course managers are mostly interested in vehicle security to prevent misuse and ensure safety on the golf course (e.g., using geofencing features). In general, the stakeholders rely on the FMS and its services to optimize their respective business tasks.

1) FMS IoT cloud infrastructure: The FMS runs atop a nontrivial IoT cloud infrastructure that includes a variety of IoT cloud resources. Figure 1 gives a high-level overview of the FMS infrastructure. For our discussion, the two most relevant types of IoT cloud resources are on-board physical gateways (G) and cloud virtual gateways (VG). Most of

[1] http://pcccl.infosys.tuwien.ac.at/

the vehicles are equipped with on-board gateways that are capable to host lightweight services such as geofencing or local diagnostics services. For legacy cars that are not equipped with such gateways, a device acting as a CAN-IP bridge is used (e.g, Teltonika FM5300[2]). In this case FMS hosts virtual gateways on the cloud that execute the aforementioned services on behalf of the vehicles.

We notice that the FMS is a large-scale system that manages thousands of vehicles and relies on diverse cloud communication protocols. Further, the FMS depends on IoT cloud resources that are geographically distributed on different golf courses around the globe. Jurisdiction over these resources can change over time, e.g., when a vehicle is handed over from the distributor to a golf course manager. In addition, these resources are usually constrained. This is why the FMS heavily relies on cloud services, e.g., for computationally intensive data processing, fault-tolerance or to reliably store historical readings of vehicle data. While the cloud offers the illusion of unlimited resources, systems of such scale as FMS can incur very high costs in practice (e.g., of computation or networking). Finally, due to the large number of involved stakeholders, the FMS needs to enable runtime customizations of infrastructure resources in order to exactly meet stakeholder requirements and allow for operation within specified compliance and legal boundaries.

Therefore, the IoT cloud resources need to be managed and governed throughout their entire lifecycle. In our approach, this is captured and modeled as *operational governance processes*.

2) Example operational governance processes: Subsequently, we highlight some basic operational governance processes in FMS that are facilitated through our framework:

- Typically, the FMS polls diagnostic data from vehicles (e.g., with CoAP). However, a golf course manager could design an operational governance process that is triggered in specific situations such as in case of emergency. Such process could, for example, increase the update rate of the vehicle sensors and change the communication protocol to MQTT in order to satisfy a high-level governance objective, e.g., company's compliance policy to handle emergency updates in (near) real-time.
- To increase fault-tolerance and guarantee history preservation of vehicle data (e.g., due to governance objectives related to legal requirements), a distributor could decide to spin up additional virtual gateways in a different availability zone.
- After multiple complaints about problems with vehicles of type X, a manufacturer would need to add additional monitoring features to all vehicles of type X to perform more detailed inspections.

This is by no means a comprehensive list of operational governance processes in software-defined IoT cloud systems. However, due to dynamicity, heterogeneity, geographical distribution, and the large scale of IoT cloud systems, traditional approaches to realize even basic operational governance processes are hardly feasible in practice. This is mostly because such approaches implicitly make assumptions such as physical on-site presence, manually logging into gateways, understanding device specifics, etc., which are difficult, if not impossible, to meet in IoT cloud systems. Therefore, due to a lack of systematic approaches for operational governance in IoT cloud systems, operations managers currently have to rely on ad-hoc solutions to deal with the characteristics and complexity of IoT cloud systems when performing operational governance processes.

[2]http://www.teltonika.lt/en/pages/view/?id=1024

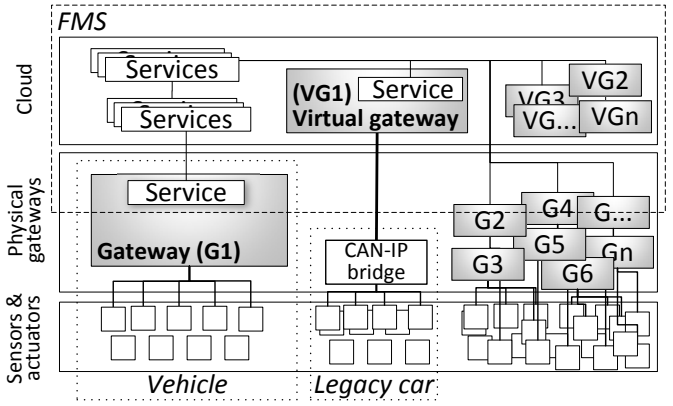

Fig. 1. Overview of FMS infrastructure.

B. Background

As we have shown earlier in [6], in software-defined IoT cloud systems, IoT cloud resources (e.g., gateways) are described as software-defined IoT units. The software-defined IoT units enable abstracting the underlying IoT cloud resources and allow for their management through well-defined APIs, exposed by these units. One of the main advantages of software-defined IoT cloud systems is opening up the traditional infrastructure silos and moving one step higher in the abstraction, i.e., effectively making applications independent of the underlying rigid infrastructure. The most important consequence is that the *functionality and states* of the underlying IoT cloud resources can be redefined in software during runtime. For example, new features such as additional cloud communication protocols can be added to the units at runtime.

In our previous work [7] we introduced the general GovOps approach for runtime governance in software-defined IoT cloud systems, as well as the concepts of *operational governance processes* that manipulate the states of IoT units (IoT cloud resources) at runtime. Such processes can be seen as a sequence of operations, which perform runtime state transitions from a current state to some desired target state (e.g., that satisfies some non-functional properties, enforces compliance, or exactly meets custom requirements). We presented a GovOps reference model that provides suitable abstractions to specify such operational governance processes. We also outlined the GovOps methodology on how to design the operational governance processes and realize governance strategies. At this point it is worth reminding that, from a technical perspective, GovOps does not make any assumptions about the implementation of operational governance processes, in the sense that such processes can be realized as business processes (e.g., using BPMN), with governance policies, via Domain Specific Languages (DSLs), or even as dedicated governance applications or services.

III. Overview of the rtGovOps framework

The main aim of our rtGovOps (*runtime GovOps*) framework is to facilitate operational governance processes for software-defined IoT cloud systems. To this end, rtGovOps provides a set of runtime mechanisms and does most of the

"heavy lifting" to support operations managers in implementing and executing operational governance processes in large-scale software-defined IoT cloud systems, without worrying about scale, geographical distribution, dynamicity, and other characteristics inherent to such systems that currently hinder operational governance in practice.

To facilitate performing the operational governance processes, while considering the characteristics of the software-defined IoT cloud systems, the rtGovOps framework follows a set of design principles, which include:

Central point of operation (R1) – Enable conceptually centralized interaction with the software-defined IoT cloud system to enable a unified view on the system's operations and governance capabilities (available at runtime), without worrying about low-level infrastructure details.

Automation (R2) – Allow for dynamic, on-demand governance of software-defined IoT cloud systems on a large scale and enable governance processes to be easily repeatable, i.e., enforced across the IoT cloud, without manually logging into individual gateways.

Fine-grained control (R3) – Expose the control functionality of IoT cloud resources at fine granularity to allow for precise definition of governance processes (to exactly meet requirements) and flexible customization of IoT cloud system governance capabilities.

Late-bound directives (R4) – Support declarative directives that are bound later during runtime in order to allow for designing generic and flexible operational governance processes.

IoT cloud resources autonomy (R5) – Provide a higher degree of autonomy to IoT cloud resources to reduce communication overhead, increase availability (e.g., in case of network partitions), enable local exception and fault handling, support protocol independent interaction, and increase system scalability.

Figure 2 gives a high-level architecture and deployment overview of the rtGovOps framework. Generally, the rtGovOps framework is distributed across the cloud and IoT devices. It is designed based on the microservices architecture[3], which among other things enables flexible, evolvable, and fault-tolerant system design, while allowing for flexible management and scaling of individual components. The main components of rtGovOps include: i) the *governance capabilities*, ii) the *governance controller* that runs on the cloud, and iii) the *rtGovOps agents* that run in IoT devices. In the remainder of this section, we will discuss these components in more detail.

A. Operational governance capabilities

As we described in Section II, operational governance processes govern software-defined IoT units throughout their entire lifecycle. Generally, *Governance capabilities* represent the main building blocks of operational governance processes and they are usually executed in IoT devices. The governance capabilities encapsulate governance operations which can be applied on deployed IoT units, e.g., to query the current version of a service, change a communication protocol, or spin up a virtual gateway. Such capabilities are described via well-defined APIs and are usually provided by domain experts who develop the IoT units. The rtGovOps framework enables such

[3]http://martinfowler.com/articles/microservices.html

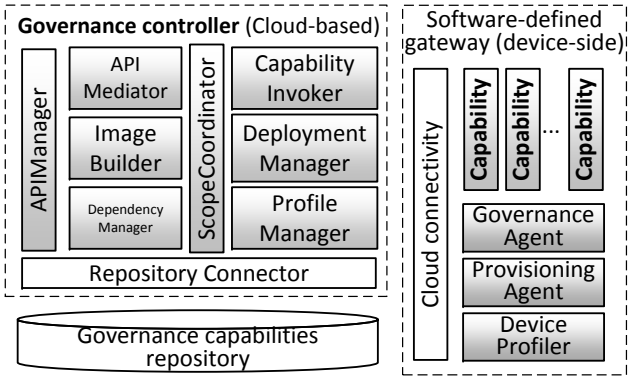

Fig. 2. Overview of rtGovOps architecture and deployment.

capabilities to be dynamically added to the system (e.g, to gateways), and supports managing their APIs. From a technical perspective, they behave like add-ons, in the sense that they extend the resources with additional operational functionality. Internally, IoT devices host rtGovOps agents that behave like an add-on manager, responsible for installing/enabling, starting/stopping a capability, and managing the APIs they expose. Generally, rtGovOps does not make any assumptions about concrete capability implementations. However, it requires them to be packaged as shown in Figure 3. Subsequently, we highlight relevant examples of governance capabilities related to our FMS application.

- *Configuration-specific capabilities* include changes to the configuration models of software-defined IoT cloud systems at runtime. For example: setting sensor poll rate, changing communication protocol for cloud connectivity, configuring data point unit and type (e.g., temperature in Kelvin as unsigned 10-bit integer), mapping a sensor or CAN bus unit to a device's virtual pin, or activating a low-pass filter for an analog sensory input.
- *Topology-specific capabilities* address structural changes that can be performed on the deployment topologies of software-defined IoT systems. Examples include replicating a virtual gateway to increase fault-tolerance or data source history preservation and push data processing logic from the application space towards the edge of the infrastructure.
- *Stream-specific capabilities* deal with managing the runtime operation of sensory data streams and continuous Complex Event Processing (CEP) queries. Therefore, to enable features like scaling out or stream replaying, operations managers need capabilities such as: filter placement near the data source to reduce network traffic, allocation of queries to gateways, and stream splitting, i.e., sending events to multiple virtual gateways.
- *Monitoring-specific capabilities* deal with adding a general monitoring metric, e.g., CPU load, or providing an implementation of a custom metric to IoT cloud resources.

For the sake of simplicity, in this paper, we assume that the capabilities are readily available[4]. In reality, they can be obtained from a central repository, provided by a third-party in a market-like fashion, or custom developed in-house.

[4]We provide example governance capabilities under https://github.com/tuwiendsg/GovOps/

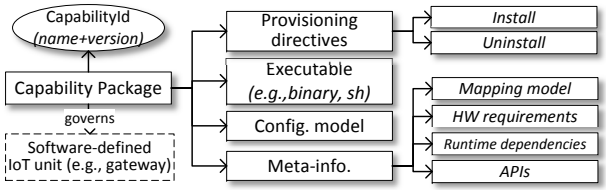

Fig. 3. Overview of capability package structure.

As mentioned above, governance capabilities are dynamically added to the IoT cloud resources. There are several reasons why such behavior is advantageous for operations managers and software-defined IoT cloud systems. For example, as we usually deal with constrained resources, static provisioning of such resources with all available functionality is rarely possible (e.g., factory defaults rarely contain the desired configuration for FMS vehicle gateways). Further, as we have seen in Section II, jurisdiction over resources (in this case FMS vehicles) can change during runtime, e.g., when a vehicle is handed over to a golf course manager. In such cases, because the governing stakeholder changes, it is natural to assume that the requirements regarding operational governance will also change, thus requiring additional or different governance capabilities. As opposed to updating the whole device image at once, we reduce the communication overhead, but also enable changing device functionality without interrupting the system, e.g., to reboot. This provides greater flexibility and enables on-demand governance tasks (e.g, by temporally adding a capability), which are often useful in systems with a high degree of dynamicity. Finally, executing capabilities in the IoT devices improves scalability of the operational governance processes and enables better resource utilization.

B. Operational governance processes and governance scopes

Operational governance processes represent a subset of the general IoT cloud governance and deal with operating and governing IoT cloud resources at runtime. Such processes are usually designed by operations managers in coordination with business stakeholders [7]. The main purpose of such processes is to enable supporting high-level governance objectives such as compliance and legal concerns, which influence system's runtime behavior. To be able to dynamically govern IoT cloud resources, the operational governance processes rely on the governance capabilities. This means that individual steps of such process usually invoke governance capabilities in order to enforce the behavior of IoT cloud resources in such manner that it complies with the governance objectives. In this context, our rtGovOps framework provides runtime mechanisms to enable execution of these operational governance processes.

As we have mentioned earlier (Section II-B), we use software-defined IoT units to describe IoT cloud resources. However, these units are not specifically tailored for describing non-functional properties and available meta information about IoT cloud resources, e.g., location of a vehicle (gateway) or its specific type and model. For this purpose, rtGovOps provides governance scopes. The governance scope is an abstract resource which represents a group of IoT cloud resources that share some common properties. For example, an operations manager can specify a governance scope to include all the vehicles of type X. The *ScopeCoordinator* (Figure 2) provides mechanisms to define and manage the governance scopes. The rtGovOps framework relies on the *ScopeCoordinator* to determine which IoT cloud resources need to be affected by an operational governance process. Generally, the governance scopes enable implementing the operational governance processes in a scalable and generic manner, since the IoT cloud resources do not have to be individually referenced within such process.

C. Governance controller and rtGovOps agents

The *Governance controller* (Figure 2) represents a central point of interaction with all available governance capabilities. It provides a mediation layer that enables operations managers to interact with IoT cloud systems in a conceptually centralized fashion, without worrying about geographical distribution of the underlying system. Internally, the governance controller comprises several microservices, among which the most important include: *DeploymentManager* and *ProfileManager* that are used to support dynamical provisioning of the governance capabilities, as well as *APIManager* and previously mentioned *ScopeCoordinator* that support operational governance processes to communicate with the underlying capabilities. The *APIManger* exposes governance capabilities to operational governance processes via well-defined APIs and handles all API calls from such processes. It is responsible to resolve incoming requests, map them to respective governance capabilities in the IoT devices and deliver results to the calling process. Among other things, this involves discovering capabilities by querying the capabilities repository, and parameterizing capabilities via input arguments or configuration directives.

Since governance capabilities are usually not "pre-installed" in IoT devices, the *DeploymentManager* is responsible to inject capabilities into such devices (e.g., gateways) at runtime. To this end it exposes REST APIs, which are used by the devices to periodically check for updates, as well as by the operational governance processes to push capabilities into the devices. Finally, the *ProfileManager* is responsible to dynamically build and manage device profiles. This involves managing static device meta-information and periodically performing profiling actions in order to obtain runtime snapshots of current device states.

Another essential part of the rtGovOps framework are the *rtGovOps agents*. They include: *ProvisioningAgent*, *GovernanceAgent* and *DeviceProfiler*. These agents are very lightweight components that run in all IoT cloud resources that are managed by rtGovOps such as the FMS vehicles. Figure 4 shows a high-level overview of the *GovernanceAgent* architecture. It is responsible to manage local governance capabilities, to wrap them in well-defined APIs and to expose them to the *Governance controller*. The rtGovOps agents offer advantages

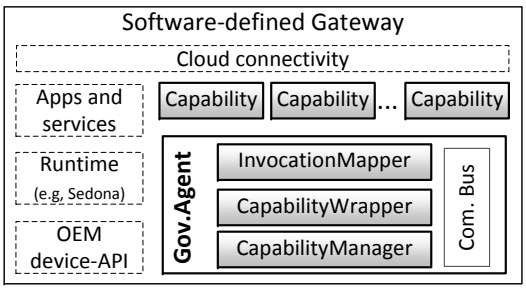

Fig. 4. Overview of the governance agent architecture.

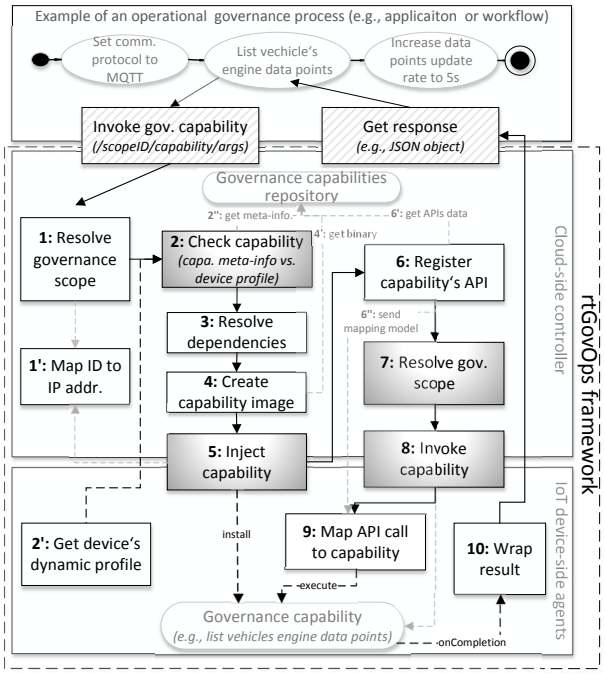

Fig. 5. Execution of an operational governance process.

in terms of general scalability of the system and provide a degree of autonomy to the IoT cloud resources.

IV. The rtGovOps framework main concepts and enabling techniques

Generally, the rtGovOps framework supports operations managers to handle two main tasks. First, the rtGovOps framework enables dynamic, on-demand *provisioning of governance capabilities*. For example, it allows for dynamically injecting capabilities into IoT cloud resources, and coordinating the dynamic profiles of these resources at runtime. Second, our framework allows for runtime management of governance capabilities throughout their entire lifecycle that, among other things, includes *remote capability invocation and managing dynamic APIs* exposed to users.

As we have mentioned earlier, in order to achieve a high-level governance objective such as enforce (part of) compliance policies for handling emergency situations an operations manager could design an operational governance process similar to the one shown in Figure 5 (top). Individual actions of such processes usually reference specific governance capabilities and rely on rtGovOps to support their execution. Figure 5 depicts a simplified sequence of steps executed by the rtGovOps framework when a governance capability gets invoked by an operational governance process. For the sake of clarity, we omit several steps performed by the framework and mainly focus on showing the most common interaction, i.e., we assume no errors or exceptions occur. We will discuss the most important steps performed by rtGovOps below. Note that all of these steps are performed transparently to operations managers and operational governance processes. The only thing that such processes observe is a simple API call (similar to REST service invocation) and a response (e.g., a JSON array in this case). Naturally, the process is responsible to provide arguments and/or configuration directives that are used by rtGovOps to parametrize the underlying capabilities.

A. Automated provisioning of governance capabilities

In order to enable dynamic, on-demand provisioning of governance capabilities whenever a new capability is requested (i.e., referenced in an operational governance process), the rtGovOps framework needs to perform the following steps: i) the *ScopeCoordinator* resolves the governance scope to get a set of devices to which the capability will be added; ii) the *ProfileManager* checks whether the governance capability is available and compatible with the device; iii) the *Dependency Manager* resolves runtime dependencies of the capability; iv) the *ImageBuilder* creates a capability image; v) Finally, the *DeploymentManager* injects the capability into devices; An overview of this process is also shown in steps $1-5$ in Figure 5.

Algorithm 1 shows the capability provisioning process in more detail. An operational governance process requests a capability by supplying a capability ID (currently consisting of capability name and version) and an operational governance scope (more detail in Section IV-B). After that rtGovOps tries to add the capability (together with its runtime dependencies) to a device. If successful, it continues along the steps shown in Figure 5. The algorithm performs in a similar fashion to a fail-safe iterator, in the sense that it works with snapshots of devices states. For example, if something changes on the device side inside $checkComponent$ (Algorithm 1, lines $2-5$) it cannot be detected by rtGovOps and in this case the behavior of rtGovOps is not defined. Since we assume that all the changes to the underlying devices are performed exclusively by our framework, this is a reasonable design decision. Other errors, such as failure to install a capability on a specific device, are caught by rtGovOps and delivered as notifications to the operational governance process, so that they do not interrupt its execution.

Algorithm 1: Governance capability provisioning.

```
   input: capaID : A capability ID.
          gscope : Operational governance scope.
   result: Capability added to device or error occurred.
 1 func checkComponent(component, device)
 2     capaMeta ← queryCapaRepo(component)
 3     devProfile ← getDeviceProfile(device)
 4     status ← isCompatible(capaMeta, devProfile)
 5     return status
 6 end
   /* Begin main loop.                                    */
 7 components ← resolveDependencies(capaID)
 8 components ← add(capaID)
 9 for device in resolveGovScope(gscope) do
10     for component in components do
11         if not checkComponent(component, device) then
12             error
13         end
14     end
       /* Inject capability.                              */
15     capaImg ← createImg(components)
16     deployCapa(capaImg, device)
17     installCapa(capaImg) // On device-side
18 end
```

1) Capability checking: From the steps presented in Algorithm 1 $checkComponent$ (lines $1-6$) and $injectCapability$ (lines $15-17$) are the most interesting. The framework invokes $checkComponent$ for each governance capability and all of its dependencies for the currently considered device. At this point rtGovOps verifies that the component can be installed on this specific device. To this end, the *ProfileManager* first queries the *central capabilities repository*. Besides the capability bi-

naries, the repository stores capability meta-information, such as required CPU instruction set (e.g., ARMv5 or x86), disk space and memory requirements, as well as installation and decommissioning directives. After obtaining the capability meta-information the framework starts building the current device profile. This is done in two stages. First, the gateway features catalog is queried to obtain relevant static information, such as CPU architecture, kernel version and installed userland (e.g., BusyBox[5]) or OS. Second, the *ProfileManager* in coordination with *DeviceProfiler* executes a sequence of runtime profiling actions to complete the dynamic device profile. For example, the profiling actions include: currently available disk space, available RAM, firewall settings, environment information, list of processes and daemons, and list of currently installed capabilities. Finally, when the dynamic device profile is completed, it is compared with the capability's meta information in order to determine if the capability is compatible with the device.

2) Capability injection: The rtGovOps *capability injection mechanism*, deals with uploading and installing capabilities on devices, as well as managing custom configuration models. This process is structured along three main phases: Creating a capability image, deploying the capability image on a device and installing the capability locally on the device.

i) After the *ProfileManager* determines a capability is compatible with the gateway, the *ImageBuilder* creates a capability image. The capability image is rtGovOps internal representation of the capability package (see Figure 3). In essence it is a compressed capability package containing component binaries and a dynamically created runlist. The runlist is an ordered list of components that need to be installed. It is created by the *DependencyManager* and its individual steps reference component installation or decommissioning directives that are obtained from the *capabilities repository*.

ii) In the second phase, *DeploymentManager* deploys the image to the device. We support two different deployment strategies. The first strategy is *poll-based*, in the sense that the image is placed in the update queue and remains there for a specified period of time (TTL). The *ProvisioningAgent* periodically inspects the queue for new updates. When an update is available, the device can poll the new image when it is ready, e.g., when the load on it is not too high. A governance process can have more control over the poll-based deployment by specifying a capability's priority in the update queue. Finally, on successful update the *DeploymentManager* removes the update from the queue. The second deployment strategy allows governance capabilities to be asynchronously *pushed* to gateways. Since the capability is forced onto the gateway, it should be used cautiously and for urgent updates only, such as increasing a sensor poll-rate in emergency situations. Finally, independent of the deployment strategy, the framework performs a sequence of checks to ensure that an update was performed correctly (e.g., compares checksums) and moves to the next phase.

iii) In the final phase, the *ProvisioningAgent* performs a local installation of the capability binaries and its runtime dependencies, and performs any custom configurations. Initially, *ProvisioningAgent* unpacks the previously obtained capability image and verifies that the capability can be installed based on the current device profile. In case the conditions are not satisfied, e.g., due to disk space limitation, the process is aborted and an error is sent to the *DeploymentManager*. Otherwise, the *ProvisioningAgent* reads the runlist and performs all required installation or decommissioning steps.

A limitation of the current rtGovOps prototype is that it only provides rudimentary support to specify installation and decommissioning directives. Therefore, capability providers need to specify checks, e.g., if a configuration file already exists, as part of the installation directives. In the future we plan to provide a dedicated provisioning DSL to support common directives and interactions.

B. rtGovOps APIs and invocation of governance capabilities

When a new governance capability is injected into a gateway, the rtGovOps framework performs the following steps: i) register capability with *APIManager*; ii) *ScopeCoordinator* resolves the governance scope; iii) *APIMediator* provides a mapping model to the *GovernanceAgent*; iv) the *GovernanceAgent* wraps the capability into a well-defined API, dynamically exposing it to the outside world; v) *CapabilityInvoker* invokes the capability and deliver the result to the invoking operational governance process when the capability execution completes. A simplified version of this process is also shown in steps $6-10$ in Figure 5.

Before we dive into technical details of this process, it is worth mentioning that currently in the capabilities repository, besides aforementioned capability meta-information and binaries, we also maintain well-defined capability API descriptions, e.g., functional, meta and lifecycle APIs. These APIs are available to operations managers as soon as a capability is added to the repository and independent of whether the capability is installed on any device. Additionally, we provide a general rtGovOps API that is used to allow for more control over the system and its capabilities. It includes *CapabilityManager* API (e.g., list capabilities, check if capability installed/active), capability lifecycle API (e.g., start, stop or remove capability), and *ProvisioningAgent* API (e.g., install new capability). Listing 1 shows some examples of such APIs as REST-like services (version numbers are omitted for clarity).

```
1 /* General case of capability invocation.      */
2 /govScope/{capabilityId}/{methodName}/{arguments}?
3 arg1={first-argument}&arg2={second-argument}&...

4 /* Data points capability invocation example. */
5 /deviceId/DPcapa/setPollRate/arguments?rate=5s
6 /deviceId/DPcapa/list

7 /* Capabilities manager examples.              */
8 /deviceId/cManager/capabilities/list
9 /deviceId/cManager/{capabilityId}/stop
```

Listing 1. Examples of capabilities and rtGovOps APIs.

1) Single invocation of governance capabilities: In the following we mainly focus on explaining the steps that are performed by the rtGovOps framework when a capability is invoked on a single device. The more general case involving multiple devices and using operational governance scopes is discussed in the next section.

When a capability gets invoked by an operational governance process for the first time, *APIManager* does not know anything about it. Therefore, it first needs to check, based on

[5]http://busybox.net

the API call (e.g., see Listing 1), if the capability exists in the central capabilities repository. After the capability is found and provisioned (Section IV-A), the rtGovOps framework tries to invoke the capability. This involves the following steps: registering the capability, mapping the API call, executing the capability, and returning the result.

i) First, the *APIManager* registers the API call with the corresponding capability. This involves querying the capability repository to obtain its meta-information (such as expected arguments), as well as building a dynamic mapping model. Among other things, the mapping model contains the capability ID, a reference to a runtime environment (e.g., Linux shell), a sequence of input parameters, the result type, and further configuration directives. The *APIMediator* forwards the model to the device (i.e. *GovernanceAgent*) and caches this information for subsequent invocations. During future interactions, the rtGovOps framework acts as transparent proxy, since subsequent steps are handled by the underlying devices.

ii) In the next step, rtGovOps needs to perform a mapping between the API call and the underlying capability. Currently, there are two different ways to do this. By default, rtGovOps assumes that capabilities follow the traditional Unix interaction model, i.e., that all arguments and configurations (e.g., flags) are provided via the standard input stream (stdin) and output is produced to standard output (stdout) or standard error (stderr) streams. This means, if not specified otherwise in the mapping model, the framework will try to invoke the capability by its ID and will forward the provided arguments to its stdin. For capabilities that require custom invocation, e.g., property files, policies, or specific environment settings, the framework requires a custom mapping model. This model is used in the subsequent steps to correctly perform the API call.

iii) Finally, the *CapabilityInvoker* in coordination with the *GovernanceAgent* invokes the governance capability. As soon as the capability completes, the *GovernanceAgent* collects and wraps the result. Currently, the framework provides means to wrap results as JSON objects for standard data types and it relies on the mapping model to determine the appropriate return type. However, this can be easily extended to support more generic behavior, e.g., by using Google Protocol Buffers[6].

2) Operational governance scopes: When an operational governance process gets invoked on a governance scope, the aforementioned invocation process remains the same, with the only difference that rtGovOps performs all steps on a complete governance scope in parallel instead on an individual device. To this end, the *ScopeCoordinator* enables dynamic resolution of the governance scopes.

There are several ways how a governance scope can be defined. For example, an operations manager can manually assign a set of resources to a scope, such as all vehicles belonging to a golf course, or they can be dynamically determined depending on runtime features by querying governance capabilities to obtain dynamic properties such as current configuration model. To bootstrap defining the governance scopes, the *ScopeCoordinator*, defines a global governance scope that is usually associated with all the IoT cloud resources administered by a stakeholder at the given time. Governance scope specifications are implemented as composite predicates referencing device meta information and profile attributes, The predicates are applied to the global scope, filtering out all resources that do not match the provided attribute conditions. The *ScopeCoordinator* uses the resulting set of resources to initiate capability invocation with the *CapabilityInvoker*. The *ScopeCoordinator* is also responsible to provide support for gathering results delivered by the invoked capabilities. This is needed since the scopes are resolved in parallel and the results are asynchronously delivered by the IoT devices.

[6]http://code.google.com/p/protobuf/

V. Evaluation & Prototype implementation

A. Prototype implementation

In the current prototype, the rtGovOps *Governance controller* microservices are implemented in Java and Scala programming languages. The rtGovOps agents are based on lightweight httpd server and are implemented as Linux shell scripts. The complete source code and supplement materials providing more details about current rtGovOps implementation are publicly available in Git Hub[7].

B. Experiments setup

In order to evaluate how our rtGovOps framework behaves in a large-scale setup (hundreds of gateways), we created a virtualized IoT cloud testbed based on CoreOS[8]. In our testbed we use Docker containers to virtualize and mimic physical gateways in the cloud. These containers are based on a snapshot of a real-world gateway, developed by our industry partners. The Docker base image is publicly available in Docker Hub under dsgtuwien/govops-box[9].

For the subsequent experiments we deployed a CoreOS cluster on our local OpenStack cloud. The cluster consists of 4 CoreOS 444.4.0 VMs (with 4 VCPUs and 7GB of RAM), each running approximately 200 Docker containers. Our rtGovOps agents are preinstalled in the containers. The rtGovOps Governance controller and capabilities repository are deployed on 3 Ubuntu 14.04 VMs (with 2VCPUs and 3GB of RAM). The operational governance processes are executing on a local machine (with Intel Core i7 and 8GB of RAM).

C. Governing FMS at runtime

We first show how our rtGovOps framework is used to support performing operational governance processes on a real-world FMS application for monitoring vehicles (e.g., location and engine status) on a golf course (Section II). The application consists of several services. On the one side, there is a light-weight service running in the vehicle gateways that interfaces with vehicle sensors via the CAN protocol, and feeds sensory data to the cloud. On the cloud-side of the FMS application, there are several services that, among other things, perform analytics on the sensory data and offer data visualization support. In our example implementation of this FMS application, the gateway service is implemented as a software-defined IoT unit that among other things provides an API and mechanisms to dynamically change the cloud communication protocol without stopping the service.

The FMS application polls diagnostic data from vehicles with CoAP. However, in case of an emergency, a golf course

[7]http://github.com/tuwiendsg/GovOps

[8]http://coreos.com/

[9]https://registry.hub.docker.com/u/dsgtuwien/govops-box/

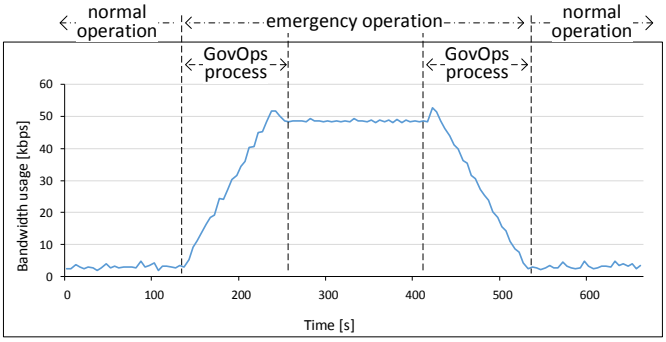

Fig. 6. Example execution of operational governance process in the FMS.

manager needs to increase the update rate and switch to MQTT in order to handle emergency updates in (near) real-time. This can be easily specified with an operational governance process that contains the following steps: change communication protocol to MQTT, list vehicle engine and location data points and set data points update rate, e.g, to 5 seconds. These steps are also depicted in Figure 5 (top). The golf course manager relies on rtGovOps governance capabilities to realize individual process steps and rtGovOps mechanisms (Section IV) to execute the operational governance process.

Figure 6 shows the bandwidth consumption of the FMS application that monitors 50 vehicles over a period of time. We notice two distinct operation modes: normal operation and operation in case of an emergency (emergency operation). Most notable are the two transitions: first, from normal to emergency operation and second, returning from emergency to normal operation. These transitions are described with the aforementioned operational governance process that is executed by the rtGovOps framework. The significant increase in bandwidth consumption happens during the execution of the operational governance process, because it changes the communication protocol from polling the vehicles approximately every minute with CoAP, to pushing the updates every 5 seconds with MQTT.

Typically, when performing processes such as the transition from normal to emergency operation without the rtGovOps framework, golf course managers (or generally operations managers) need to directly interact with vehicle gateways. This usually involves long and tedious tasks such as manually logging into gateways, dealing with device specific configurations or even an on-site presence. Therefore, realizing even basic governance processes, such as the one we presented above, involves performing many manual and error prone tasks, usually resulting in a significant increases in operations costs. Additionally, in order to be able to have a timely realization of governance processes and consistent implementation of governance strategies across the system, very large operations and support teams are required. This is mainly due to the large scale of the FMS system, but also due to geographical distribution of the governed IoT resources, i.e., vehicles.

Besides the increased efficiency, the main advantage that rtGovOps offers to operations managers is reflected in the flexibility of performing operational governance processes at runtime. For example, in Figure 6 the execution of the operational governance process took around 2 minutes. In our framework this is, however, purely a matter of operational governance process configuration (naturally with upper limits as we show in the next section). This means, the operational governance process can be easily customized to execute the protocol transition "eagerly", in the sense to force the change as soon as possible, even within seconds, or "lazy", to roll-out the change step-wise, e.g., 10 vehicles at the time. The most important consequence is the opportunity to effectively manage tradeoffs. For example, executing the process eagerly incurs higher costs, due to additional networking and computation consumption, but it is needed in most emergency situations. Conversely, executing the process in a lazy manner can be desirable for non-emergency situations, since operations managers can prevent possible errors to affect the whole system.

Figure 6 also shows that rtGovOps introduces a slight communication overhead. This is observed in the two peaks at the end of the first process execution, when the framework performs the final checks that the process completed successfully and also when the second process gets triggered, i.e., when the capabilities get invoked on the vehicles. However, in our experiments this overhead was small enough not to be statistically significant. An additional performance-related concern of using rtGovOps is that network latency can slow down the execution of the operational governance process. However, since rtGovOps follows the microservices architecture style, it is possible to deploy relevant services (*API-* and *DeploymentManager*) on Cloudlets [3] near the vehicles, e.g., on golf courses, where they can utilize local wireless networks.

D. Experiments results

To demonstrate the feasibility of using rtGovOps to facilitate operational governance processes in large-scale software-defined IoT cloud systems, we evaluate its performance to govern approximately 800 vehicle gateways that are simulated in the previously described test-bed. In our experiments, we mainly focus on showing the scalability of the two main mechanisms of the rtGovOps framework: (i) capability invocation and (ii) automated capability provisioning. We also consider the performance of capability checking and governance scope resolution. The reason why we put an emphasis on the scalability of our framework is that it is one of the key factors to enable consistent implementation of governance objectives across a large-scale systems. For example, if the execution of an operational governance process were to scale exponentially with the size of the resources pool, theoretically it would take infinitely long to have a consistent enforcement of the governance objectives in the whole system, with sufficiently large resource pool. The results of the experiments are averaged results of 30 repetitions and we have experimented with 5 different capabilities that have different properties related to their size and computational overhead.

Figure 7 shows the execution time of the first invocation of a capability (stacked bar) and an average invocation time of capability execution (plain bar). We notice that the first invocation took approximately between 10 and 15 seconds and average invocation varied between 4 and 6 seconds depending on the scope size (measured in the number of gateways). The main reason for such a noticeable difference is the invocation caching performed by rtGovOps. This means that most of the steps, e.g., building capability image and building the mapping model are only performed when a capability is invoked for the first time, since in the subsequent invocations the capability is already in the gateways and the mapping can be done in cache. In Figure 8, we present the average execution time of a capability (as it is observed by an invoking operational governance process on the locale machine), average execution

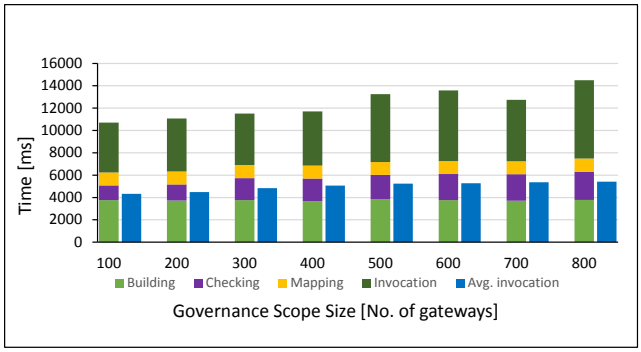

Fig. 7. Capabilities first invocation.

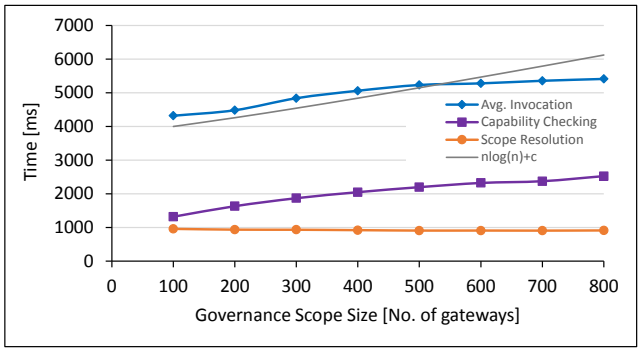

Fig. 8. Average invocation time of capabilities on a governance scope.

of capability checking mechanism and governance scope resolution. As a reference, the diagram also shows a plot of a $nlog(n) + c$ function. We can see that the mechanisms scale within $O(nlog(n))$ for relatively large governance scopes (up to 800 gateways), which can be considered a satisfactory result. We also notice that computational overheads of the capabilities have no statistically significant impact on the results, since they are distributed among the underlying gateways. Finally, it is interesting to notice that the scope resolution time actually decreases with increasing scope size. The reason for this is that in the current implementation of rtGovOps, scope resolution always starts with the global governance scope and applies filters (lambda expressions) on it. After some time Java JIT "kicks-in" and optimizes filters execution, thus reducing the overall scope resolution time.

In Figure 9, we show the general execution times of the rtGovOps capability provisioning mechanism (push-based deployment strategy) for two different capabilities. The first one has a size order of magnitude in MB and second capability size is measured in KB. There are several important things to notice here. First, the capability provisioning also scales similarly to $O(nlog(n))$. Second, after the governance scope size reaches 400 gateways there is a drop in the capability provisioning time. The reason for this is that the rtGovOps load balancer spins-up additional instances of the *DeploymentManger* and *ImageBuilder*, naturally reducing provisioning time for subsequent requests. Finally, the provisioning mechanism behaves in a similar fashion for both capabilities. The reason for this is that all gateways are in the same network, what can be seen as an equivalent to vehicles deployed on one golf course.

E. Discussion and lessons learned

The observations and results of our experiments show that rtGovOps offers advantages in terms of realizing operational governance processes with greater flexibility, and also makes such processes easily repeatable, traceable and auditable, which is crucial for successful implementation of governance strategies. Generally, by adopting the notion of governance capabilities and by utilizing resource agents, rtGovOps allows for operational governance processes to be specified with *finer granularity (R3)*, but also give a *degree of autonomy (R5)* to the managed IoT cloud resources. Therefore, by selecting suitable governance capabilities, operations managers can precisely define desired states and runtime behavior of software-defined IoT cloud systems. Further, since the capabilities are executed locally in IoT cloud resources (e.g., in the gateways), our framework enables better utilization of the "edge of infrastructure" and allows for local error handling, thus increasing system availability and scalability. Further, the main advantage of approaching provisioning and management of governance capabilities in the described manner is that operation managers do not have to worry about geographically-distributed IoT cloud infrastructure nor deal with individual devices, e.g., key management or logging in. They only need to *declare (R4)* which capabilities are required in the operational governance process and specify a governance scope. The rtGovOps framework takes care of the rest, effectively giving a *logically centralized view (R1)* on the management of all governance capabilities. Further, by *automating (R2)* the capability provisioning, rtGovOps enables installing, configuring, deploying, and invoking the governance capabilities in a scalable and easily repeatable manner, thus reducing errors, time, and eventually costs of operational governance.

It should be also noted that there is a number of technical limitations of and possible optimizations that can be introduced in the current prototype of the rtGovOps framework. As we have already mentioned, rtGovOps currently offers limited support for specifying provisioning directives. Additionally, while experimenting with different types of capabilities, we noticed that in many cases a better support to deal with streaming capabilities would be useful. Regarding possible optimizations, in the future we plan to introduce support for automatic composition of capabilities on the device level, e.g., similar to Unix piping. This should reduce the communication overhead of rtGovOps and improve resource utilization in general. In spite of the current limitations, the initial results are promising, in the sense that rtGovOps *increases flexibility* and enables *scalable execution of operational governance processes* in software-defined IoT cloud systems.

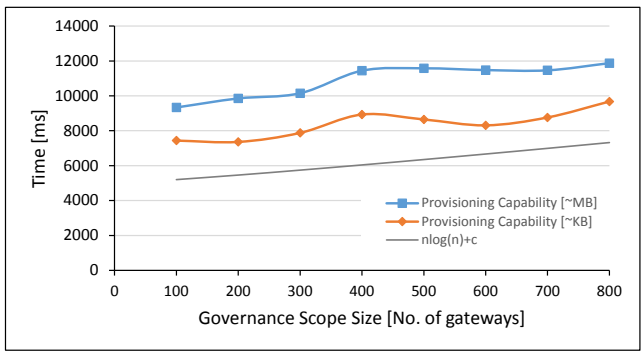

Fig. 9. Average capability provisioning duration (push-based strategy).

VI. Related work

Recently, IoT governance has been receiving a lot of attention. For example, in [10] the author evaluates various governance aspects, such as privacy, security, ethics, etc., and defines main principles of IoT governance, e.g., legitimacy, transparency, openness, and accountability. Governance approaches such as CMMI [11] or COBIT [12] also provide models and methodologies to manage governance objectives. Such approaches are complementary to our own and can be used along with rtGovOps to specify, manage and trace such high-level governance objectives.

Also approaches dealing with IoT mobile cloud operations and resources management have recently emerged. For example, in [1], [13]–[16] the authors mostly deal with IoT infrastructure virtualization and its management on the clouds. Approaches presented in [3]–[5], [17]–[19] address the issues to aggregate and manage the computational resources provided by various IoT devices, mobile devices and the clouds. In [2], [20] the authors focus on utilizing clouds for additional storage resources. In [16] the authors develop an infrastructure virtualization framework, based on asynchronous event exchange. They provide an event matching algorithm to enable coordination and management of sensory event streams. In [1] the authors propose virtualizing physical sensors on the cloud and provide template-based management and monitoring mechanisms for the virtual sensors. SenaaS [13] mostly focuses on providing a cloud semantic overlay atop physical IoT infrastructure. It defines an ontology to manage interaction with heterogeneous devices and mediate different data formats. OpenIoT framework [14] utilizes semantic web technologies and CoAP, to enable discovering, linking and orchestrating Internet connected objects. In [15] the authors focus on enabling sensing and actuating as a service, providing support for device management and identification, as well as their selection and aggregation. Edge Cloud Composits [5] provide support to enable mobile devices to enhance their resources, by utilizing additional near-by devices or remote clouds. Such approaches provide various governance capabilities such as template-based controlling of sensor groups, registering and decommissioning sensors, orchestrating IoT devices, as well as monitoring the IoT cloud systems QoS. Aforementioned approaches provide techniques for optimizing IoT cloud resources utilization, such as computation offloading, service migration or context-aware resource aggregation. Our rtGovOps conceptually builds on such approaches, and goes one step further by providing support for dynamic, on-demand provisioning of the governance capabilities and enabling their management throughout the entire lifecycle. By doing so rtGovOps increases flexibility and facilitates execution of operational governance processes in large-scale IoT cloud systems.

VII. Conclusion and future work

In this paper, we introduced the rtGovOps framework for runtime operational governance in software-defined IoT cloud systems. We presented rtGovOps runtime mechanisms and enabling techniques that support operations managers to handle two main tasks: (i) perform dynamic, on-demand provisioning of governance capabilities and (ii) remotely invoke such capabilities in IoT cloud resources remotely, via dynamic APIs. We demonstrated, on a real-world case study, how our framework can be used to facilitate execution of operational governance processes in large-scale software-defined IoT cloud systems. The initial results are promising in several aspects. We showed that the rtGovOps framework enables operational governance processes to be executed in a scalable manner across relatively large IoT cloud resource pools. Additionally, we discussed how rtGovOps enables flexible execution of operational governance processes by automating the execution of such processes to a large extent, offering finer-grained control over IoT cloud resources and providing a logically centralized interaction with IoT cloud resource pools.

In the future we plan to address the current limitations of rtGovOps, described in Section V. We also plan to extend the rtGovOps framework to support specifying and managing high-level governance objectives.

Acknowledgment

This work is sponsored by Pacific Controls Cloud Computing Lab (PC3L).

References

[1] M. Yuriyama and T. Kushida, "Sensor-cloud infrastructure-physical sensor management with virtualized sensors on cloud computing," in *NBiS*, 2010.

[2] P. Stuedi, I. Mohomed, and D. Terry, "Wherestore: Location-based data storage for mobile devices interacting with the cloud," in *MCS*, 2010.

[3] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[4] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *MobiSys '10*, pp. 49–62, ACM, 2010.

[5] K. Bhardwaj, S. Sreepathy, A. Gavrilovska, and K. Schwan, "Ecc: Edge cloud composites," in *MobileCloud 2014*, pp. 38–47, IEEE, 2014.

[6] S. Nastic, S. Sehic, D.-H. Le, H.-L. Truong, and S. Dustdar, "Provisioning software-defined iot systems in the cloud," in *FiCloud*, 2014.

[7] S. Nastic, C. Inzinger, H.-L. Truong, and S. Dustdar, "Govops: The missing link for governance in software-defined iot cloud systems," in *WESOA14*, 2014.

[8] SOA Software, "Integrated SOA governance." URL: http://www.soa.com/solutions/integrated_soa_governance. [Online; accessed June-2014].

[9] IBM , "SOA pages - Definition of SOA governance." URL: http://ibm.com/software/solutions/soa/gov/. [Online; accessed June-2014].

[10] R. H. Weber, "Internet of things–governance quo vadis?," *Computer Law & Security Review*, vol. 29, no. 4, pp. 341–347, 2013.

[11] D. M. Ahern, A. Clouse, and R. Turner, *CMMI distilled: a practical introduction to integrated process improvement*. Addison-Wesley Professional, 2004.

[12] G. Hardy, "Using IT governance and COBIT to deliver value with IT and respond to legal, regulatory and compliance challenges," *Information Security technical report*, vol. 11, no. 1, pp. 55–61, 2006.

[13] S. Alam, M. Chowdhury, and J. Noll, "Senaas: An event-driven sensor virtualization approach for internet of things cloud," in *NESEA*, 2010.

[14] J. Soldatos, M. Serrano, and M. Hauswirth, "Convergence of utility computing with the internet-of-things," in *IMIS*, pp. 874–879, 2012.

[15] S. Distefano, G. Merlino, and A. Puliafito, "Sensing and actuation as a service: a new development for clouds," in *NCA*, pp. 272–275, 2012.

[16] M. M. Hassan, B. Song, and E.-N. Huh, "A framework of sensor-cloud integration opportunities and challenges," in *ICUIMC*, 2009.

[17] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Conference on Computer systems*, ACM, 2011.

[18] K. Kumar and Y.-H. Lu, "Cloud computing for mobile users: Can offloading computation save energy?," *Computer*, vol. 43, no. 4, pp. 51–56, 2010.

[19] R. Kemp, N. Palmer, T. Kielmann, and H. Bal, "Cuckoo: a computation offloading framework for smartphones," in *Mobile Computing, Applications, and Services*, pp. 59–79, Springer, 2012.

[20] A. Zaslavsky, C. Perera, and D. Georgakopoulos, "Sensing as a service and big data," *arXiv preprint arXiv:1301.0159*, 2013.