FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

On Provisioning and Configuring Ensembles of IoT, Network Functions and Cloud Resources

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Lingfan Gao
Registration Number 01633394

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Dr.techn. Hong-Linh Truong

Vienna, 27th August, 2018

Lingfan Gao   Hong-Linh Truong

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Lingfan Gao
Vienna

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 27. August 2018

Lingfan Gao

Acknowledgements

I would first like to thank my thesis supervisor Priv.-Doz. Dr. Hong-Linh Truong. He has been an incredibly hardworking supervisor who on many occasions provided me with advice and comments during the non-work hours. He always had a clear idea of the destination that the thesis work should arrive at while giving me guidance. He was never quick to dismiss any idea but saw the opportunity to enrich our work with them.

I would also like to thank BSc Michael Hammerer who is a fellow master student of Software Engineering and Internet Computing. He is also working in the domain of resource ensembles with my thesis supervisor. Some of the conceptual design and implementation was done with the assistance of Michael. However, our topics are clearly separate as he deals with interoperability of resource ensembles. I wish him the best of luck with his master thesis.

Finally, I want to express my profound gratitude to my parents for all their support both financial and emotional throughout years of study.

Kurzfassung

Mit dem rasanten Wachstum des Internets der Dinge (engl. IoT) ist ein effizientes Management der Ressourcen, aus denen ein IoT-basiertes System besteht, notwendig. Da IoT-Systeme oft zusammen mit der Cloud verwendet werden, bezeichnen wir diese Systeme als IoT-Cloud-Systeme. Es gibt eine Reihe von IOT-Cloud-Systemen, die derzeit in der Literatur untersucht werden, vom Flottenmanagement bis zur Datenzentren-Wartung. Diese IoT-Cloud-Systeme bestehen aus Ressourcen die materiell oder virtuell sein können, wie Sensoren, Netzwerkfunktionen, Software-Artefakte oder Cloud-Dienste. IoT-Cloud-System Szenarien erfordern eine schnelle Fertigstellung und Konfiguration von Ressourcen in Echtzeit, um sich an veränderte Nutzeranforderungen anzupassen. Mit der wachsenden Popularität von *aaS (X as a Service) werden immer mehr Organisationen und Dritte zu Anbietern spezialisierter Ressourcen. Jeder Ressourcenanbieter verwendet verschiedene Methoden und APIs zur Verwaltung ihrer Ressourcen. Dies funktioniert jedoch heutzutage zu Lasten der Nutzer, die komplette Ressourcen-Pakete sich beschaffen und schließlich benutzen müssen. Der Mehraufwand, der erzeugt wird, um sich separat mit verschiedenen Ressourcenanbietern zu verbinden, ist signifikant. Unser Ziel ist es, diesen Mehraufwand und das für den Nutzer erforderliche Wissen zu reduzieren, um Pakete von IoT, Netzwerkfunktionen und Cloud-Dienste zur Erstellung funktionaler Systeme schnell fertigzustellen und zu konfigurieren. Deshalb schlagen wir eine neuartige Architektur und ein System vor, das darauf abzielt, komplette Ressourcen-Pakete in einer dynamischen Umgebung auf Abruf zu konfigurieren und fertigzustellen. Unser Framework harmonisiert die Repräsentation von Ressourcen und Ressourcen Anbietern durch die Abstraktion höherer Informationsmodelle. Mit den Abstraktionen, die diese Informationsmodelle bieten, sind wir in der Lage, eine einheitliche API zur Verwaltung von Ressourcen zur Verfügung zu stellen, ohne uns mit den dazugehörigen Informationen auf niedrigerem Level für verschiedene Arten von Ressourcen und Infrastrukturen zu befassen. Schließlich bieten wir eine Bewertung unseres Frameworks auf einer funktionalen sowie Leistungsebene.

Abstract

With the rapid growth of the Internet of Things (IoT) and their integration with cloud computing systems, there is a need for effective management of resources that make up an IoT-based system. These IoT Cloud Systems are made up of resources which may be physical or virtual such as sensors, network functions, software artifacts or cloud services. There are a number of IoT Cloud Systems currently studied in literature with various applications from vehicle fleet management to datacenter maintenance. IoT Cloud System scenarios require rapid provisioning and configuration of resources at runtime to adapt to changing user requirements.

With the growing popularity of *aaS (X as a Service) many organizations and third parties who have become IoT and Cloud resource providers. Each resource provider uses different methods and APIs to manage their resources. However, this is at the expense of today's users who need to provision and use entire end-to-end ensembles of resources. The overhead that is generated in order to interface separately with different resource providers is significant. We aim to reduce this overhead and the knowledgebase required for a user to rapidly provision and configure ensembles of IoT, network functions and cloud services to form functional systems. To this end, we propose a framework that aims to provision and configure end-to-end resource ensembles in a dynamic and on-demand environment. Our framework harmonizes the resource and resource provider representation through the abstraction of higher level information models. With the abstractions these information models provide, we are able provide a unified API to manage resources without dealing with the associated low-level information for different types of resource and infrastructures. Finally, we provide an evaluation of our framework on functionality and performance.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

1.1.1 Background

An IoT Cloud System is typically composed of many various components (both virtual and physical) that create complex interactions far more numerous than the number components themselves. IoT Cloud Systems have been studied in literature and many scenarios have been proposed. The authors of [Tho15] give the example of a vehicle management system that pulls data from embedded sensors and may remote control the vehicle in emergency. The paper [NVI+15] also uses a vehicle related scenario similar to [Tho15] but applies it on a larger scale by monitoring and controlling a fleet of vehicles. IoT Cloud Systems can also be applied to facility management of datacenters in the cloud as stated in a scenario from the authors of [TCD+15].

The dynamic environment in which these IoT Cloud Systems are used calls for rapid deployment of resources [TN16] [ACIM15]. By resources we mean the different elements that make up an IoT Cloud System as: sensors, software artifacts, gateways, communication brokers and data storage.

Deploying an IoT Cloud System involves the correct provisioning and configuration of each of its constituent resources. We therefore use the term *resource ensemble* to refer to all these resources as a whole [TN16] which communicate and interoperate between each other. A resource ensemble differs from a deployment of resources. The resource ensemble belongs to a user who deploys it as a functional system in order to satisfy his/her requirements. Once the user's requirements change then resource ensemble should be (re)configured to reflect those changes. The presence of a user context is novel and not seen in related works on IoT Cloud System deployment such as [TCD+15].

Three important resource types[TN16][LNT16] can be identified which can belong to an ensemble that composes an IoT Cloud System: IoT resources, cloud services and network function services. IoT resources are made up of devices such as sensors and actuators which reside far from the cloud either at IoT gateways or edge devices. Network function services are resources that are virtualized network functions, e.g. firewalls, routers and DNS name servers. A large collection of network function services can be found on Google's Virtual Private Cloud [SRR18]. Cloud services are the most popular of the resource categories and include software analytics services and data storage services.

Our term resource ensemble is closely linked to the concept of resource slices mentioned in [TN16]. We use the term *ensemble* as an abstraction of the resources which are structured into a resource slice. Our term *resource ensemble* is defined as a group of resources that function together to form a system that fulfills a user's requirements. This definition also covers that of the resource slice. In this thesis these two terms are used interchangeably.

1.1.2 Motivating Scenario

We illustrate a basic IoT Cloud System's resource ensemble in Figure 1.1. This example is inspired from [TB17] and it is used for the monitoring and predictive maintenance of Base Transceiver Stations(BTSs). We explain this scenario here briefly and take a more detailed look in Chapter 3. Figure 1.1 breaks the system down to its basic resources. BTS sensors sends monitoring data through MQTT brokers to analytics clients that store the data in data storage services. Each component seen in the figure is a resource in its resource ensemble.

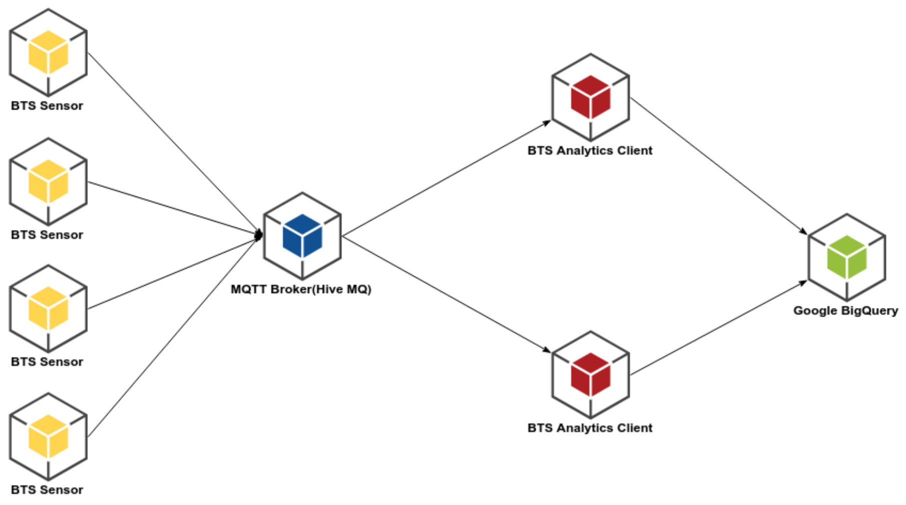


Figure 1.1: An example of a resource ensemble with the BTS scenario

There are many different providers offering their own resources. Without some form of resource discovery mechanism, there is an overhead associated with locating the appropriate resources that fit user requirements.This is a key problem that we address in this thesis. A user is faced with the problem of discovering relevant resources without any kind of search method. Then the form of the different resources must be resolved, this can be through containers, executables or even an organization's deployment API. Furthermore, each resource can differ greatly with how they are managed and configured. The heterogeneity of the available resource pool is a challenge that is currently difficult to overcome.

Figure 1.2 illustrates the example where an end user tries to deploy a BTS resource ensemble without the support of a framework that we aim to develop in this thesis. We assume that the user has managed to locate the BTS specific sensor and analytics client resources. The user is then faced with the problem of how to handle the configuration of the two resources so that communication between them is possible. The user must choose between the two choices of MQTT brokers that are available. In order to make this choice the user must study their documentation and deployment methods. Finally, a choice of storage method has to be made between the two available options: BigQuery[Goo18a] and Cassandra [Apa18a].

For chosen resources, there is then the question of configuration and monitoring. Each resource could require different methods of configuring and monitoring which might not be well documented by their providers. Moreover, the overhead of researching documentation and expanding a knowledge base just for one resource that makes up an ensemble is detrimental to the flexibility of the system. A user cannot rapidly integrate new resources into his/her system to accommodate changing requirements.

1.2 Research Questions

As our motivating scenario highlights, the current deployment process for a resource ensemble involves directly interacting with various resource providers. This process can create significant overhead for applying rapid changes to resource ensembles resulting from changing user requirements.

We aim to provide an easier deployment process for users to deploy and manage resource ensembles on-demand at runtime. This thesis aims to answer the following research questions:

- **RQ1 Concept and Modeling** How can we abstract the low-level information from different providers for different types of infrastructures and resources? How do we specify a set of resources and their relationships in order to capture the context of a resource ensemble?

- **RQ2 Design and Implementation** What features do we need to manage resource ensembles so that they can be provisioned and reconfigured on-demand at runtime? What are the possible existing technologies we can use to provide these features? What new software services and frameworks do we need to implement to provide these features?

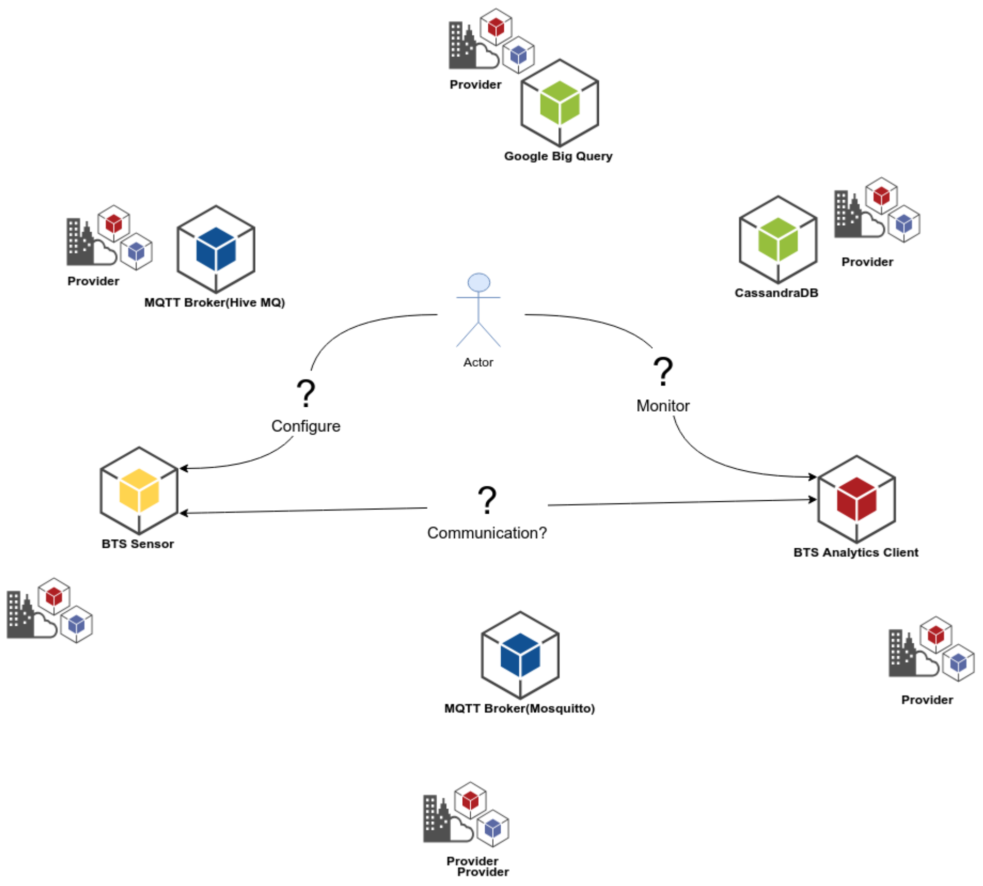


Figure 1.2: Illustration of the challenges behind resource provisioning without the support of any deployment tools

- **RQ3 Evaluation** How do we evaluate our proposed solution? What kind of experiments can we run with our final framework? How do we provide or emulate different resources and providers required to test the framework? On what criteria do we evaluate the features of our framework?

1.3 Approach

We classify seven main phases that we carried out during the course of our work on this thesis. Although the seven phases seem clearly defined here, we did adhere strictly to the "waterfall" method. During the work, we left open the possibility to identify improvements that can be made retroactively to the results of each phase.

1. In the initial phase we described the motivating scenario that we picked. We formalized this scenario in terms of stakeholders and use cases. The description of

our motivating scenario was used in later steps to constrain our work so that the purpose is clear.

2. We bootstrapped all the necessary resources and resource providers that could be used in our scenario to form functioning resource ensembles. We finished this step before starting work on our actual proposed solution since in order to create a framework to interact with resources and resource providers we had to have those artifacts available. We tried to find IoT gateways, network functions and cloud services that we could either repackage or implement. This set of example resources and providers served as part of the testbed we used to evaluate our framework.

3. Based on our motivating scenario we engineered a set of functional requirements that our framework must comply with to fulfill the use cases we laid out in our initial phase. The requirements clearly described and documented our set of functional requirements so that we could have a clear development strategy. Moreover, we also mapped our requirements to the use cases that we formalized in the initial phase to determine accurate coverage.

4. In order to answer our research question concerning modeling, we analyzed the different traits of resources and providers to come up with an information model used to represent them in our framework. Based on previous work in this area [LNT16] we adopted and adapted existing models and improved upon them. In this phase we made some preliminary models and settled on a final set of information models based on the design and implementation phase.

5. Based on the functional requirements that our framework must fulfill, we defined the usage context of our framework. From this we designed an architecture that can interact with a heterogeneous set of resources and resource providers. We also provided different deployment models that our framework could be deployed in. These deployment models involve the existing cloud but also emerging edge infrastructures.

6. We implemented a prototype that adheres to our given framework design. The prototype was built to use the resources and providers that we implemented for our chosen scenario. The implementation also involves developing a communication protocol that is compliant with our architecture. We included this protocol as a part of the prototype since many different communication methods exist and it need only be compliant with our framework design.

7. We evaluated our prototype framework on two levels: Functionality and Performance. Regarding the former, we determined a set of criteria for the evaluation of our proposed framework. By conducting functional experiments based on the use cases of our motivating scenario we evaluated the features of our framework with the criteria. Regarding performance, we ran stress and load tests to determine typical performance metrics such as response time and rate.

1.4 Contribution

We have defined the research questions that we aim to answer with this thesis. To this end we contribute a framework that includes the following key features, which take the form of functional requirements in Chapter 4:

- Interfacing with various resource providers and harmonizing data to one common information model
- Management of resource providers and their resources through a common API at runtime
- A system of query so that resources can be identified and discovered quickly through their key attributes
- A decentralized procedure where resource providers can join the framework by themselves or by interested third parties who are willing to maintain the interface
- Management of resource ensembles that form functional IoT Cloud Systems through a common API at runtime

We also implement a prototype of our proposed framework in the GitHub repository `https://github.com/SINCConcept/HINC`. The framework is built from a previous project, HINC, developed by the authors of [LNT16]. We leverage and extend its capabilities with our proposed features.The repository contains all the code and documentation of our prototype framework. Additionally, we conduct experiments on our prototype implementation for evaluation. The evaluation we proposed addresses two of the most important software testing aspects in industry today [JAAA16]: functional and non-functional.

In terms of functional evaluation, we evaluate firstly the correctness of our proposed framework. In other words, we check that the planned features of the framework function correctly. Moreover, we also devise set of use cases based on a real-world scenario in Chapter 3 . We evaluate the features of our proposed framework with respect to fulfilling the use cases of the scenario in order to gauge the usefulness of our framework. This evaluation aims to provide information on the value of our proposed framework (its usefulness).

Regaring non-functional evaluation, we evaluate the performance of our framework. This evaluation will show us how effective our prototype implementation is and whether future work can be done to create a better quality tool from our proposed framework.

The evaluation experiments will also be in the repository where we host the prototype implementation along with all the necessary tools, scripts and documentation to repeat the same experiments for other developers or researchers with an interest in our work.

As of the submission of this thesis, a separate paper submitted to the European Conference on Software Architectures (ECSA 2018) has been accepted for inclusion in the program of Posters, Tools and Demos Track.The paper [HT18] shows a high level view of the proposed framework that we work on in the thesis. The work has benefited from realistic scenarios from our industry collaboration and from INTER-IoT `http://www.inter-iot-project.eu/`. Furthermore, we have benefited from the Google Cloud Platform Education Grant (TU Wien, Advanced Services Engineering) for access to resources (VMs, container clusters...) used in our experiments.

1.5 Thesis Structure

In Chapter 2 we introduce our motivating scenario and formalize this scenario with use cases. The chapter also describes the functional requirements of our framework that were derived from the use cases. The information models that we use to represent resources and resource providers are presented in Chapter 4 along with the architecture of our proposed framework. The chapter also briefly describes some models that are important to the function of our framework. We describe in detail the prototype implementation of our framework in Chapter 5. We proceed to carry out an evaluation of our proposed framework in Chapter 6.

CHAPTER 2

State of the Art

2.1 Overview

As a starting base for the thesis, we discuss the state of the art and any related work into the deployment of resource slices. There is current research done on the topic funded by the INTER-IoT project [II18a] as a sub project known as INTER-HINC [II18b].

This thesis focuses on the provisioning and management of resource slices. The focus of this thesis implicitly means that we need to consider the provisioning and management of individual resources. Since there is a lack of current and recent research specifically into resource slices, we also study the background and related works for the following complementary topics :

- **Resource Slices** - Work related specifically to the provisioning and management of resource slices. We look for existing tools and conceptual architecture related to the management of resource slices.
- **Resource Discovery** - The query of resources based on their attributes and capabilities. The query applies for both resources that already and exist and resources that are available to be provisioned.
- **Resource Provisioning** - The deployment of resources on-demand. This topic also partly includes the configuration of the resource at the time of deployment.
- **Resource Configuration** - The configuration of resources on-demand and runtime. This also includes the storage of configuration belonging to running resources.
- **Resource Monitoring** - The retrieval of monitoring data from resources at runtime. We also consider retrieving monitoring information from resources with existing monitoring solutions.

Additionally we examine current tools available that are relevant to our work. We evaluate these related tools and discuss how well they fit as solutions to our stated problem.

2.2 Background

2.2.1 Resource Slices

The authors of [TN16] propose an information-centric approach towards IoT resources provisioning and management. The concept is called SINC - Slicing IoT, Network Functions, and Clouds. A slice here is an abstract term that refers to a logical grouping of resources. This kind of division makes sense for the end users of IoT as it allows them to create and manage groups of resources to their needs. This concept of slice is what we adopt for our proposed framework in this work. Naturally one would group different IoT resources by type. For example, we could separate sensors, communication brokers and cloud services in each of their respective groups. SINC proposes to create end-to-end slices, in any direction of an IoT System, the direction of the slice being at the discretion of the user. The high level concepts of this paper on the SINC concept were the primary drivers of our framework design.

The author of [Tru18] present an architecture for a framework to manage resource slices. The paper deals with the conceptual architecture named "Resource Slice Interoperability Hub (rsiHub)" which is illustrated in Figure 2.1. Furthermore the paper also makes reference to our motivating scenario in Chapter 3, although the scenario as mentioned in this paper, was not as mature as what we have committed to this thesis. Further focusing on IoT interoperability is also briefly mentioned. Figure 2.1 is very similar to the final architecture that is presented in Chapter 5.

2.2.2 Resource Discovery

The establishment of an information model which harmonizes various resource data is essential to resource discovery, as a structured model can be queried and filtered easily. This related paper [LNT16] proposes a high level distributed information model for IoT known as HINC - Harmonizing, IoT, Network functions and Clouds. HINC aims to model IoT and cloud resources and maintain their relationships, connecting resources across sub-networks. The work by the authors is essential for resource discovery, since effective query and discovery is only possible if the resource pool shares a common set of attributes that can be queried.

The authors of [KR15] specifically deal with an overview of service discovery in the IoT Cloud technology stack and discuss potential solutions. One of the proposals is the creation of middleware abstractions that rely on unified data representations of resources so that "neither sensors or devices nor applications or users should have to take care of the heterogeneity of the corresponding spaces (e.g. which nodes which integrates other node or executes an application)"[KR15]. This approach is also taken by the work of [LNT16] in creating unified/harmonized information models of resources.

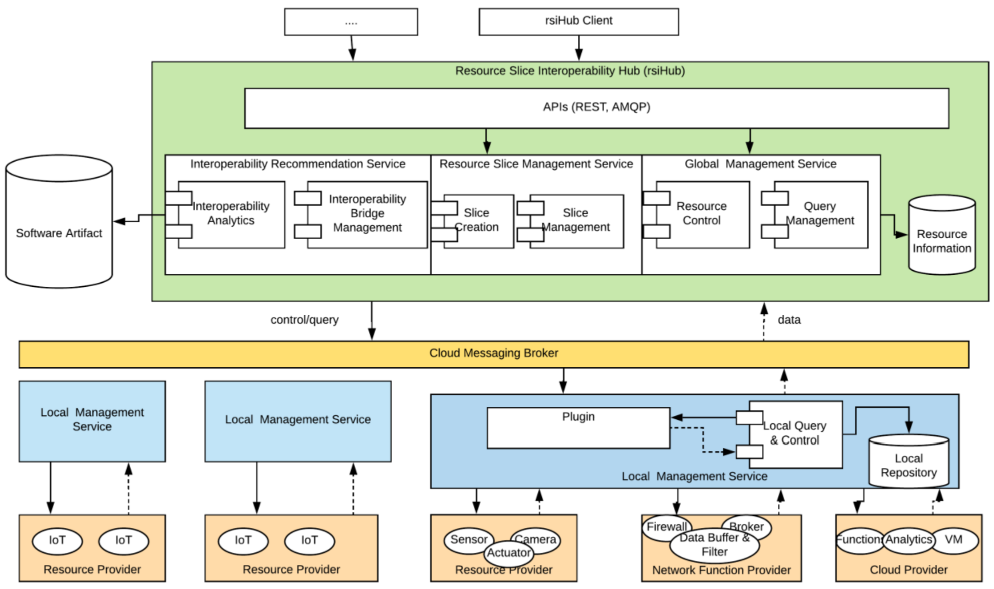


Figure 2.1: rsiHub architecture taken from [Tru18]

The requirements for web services discovery are similar to the requirements for resource discovery. Typical queries for web services include attributes such as the type of the desired service, preferred price and maximum number of returned results. Since web services are queried through their attributes just like we do for resources, the research in the web services discovery field could also apply to our work. The papers [Sim03], [CZC11] and [MC12] survey the approaches and architectures that apply to web services discovery. Service discovery can occur in two different scopes: static and dynamic. In static service discovery, details are bound at deploy time and cannot be changed. In dynamic service discovery, service details are not bound at deploy time and can be changed. Since our work has a heavy focus on runtime operations, we do not consider any type of static resource discovery. All three survey papers [Sim03][CZC11][MC12] state the need for registries where the appropriate information for services are stored. There are two main types of registries: centralized and distributed. Figure 2.2 shows an example of a centralized registry. In a central registry, only one global registry exists. This kind of service registry gives consistent results to queries since the management of the distributed resources is done centrally. In a decentralized registry, illustrated in Figure 2.3, the services are managed separated in smaller registries. The separation of the services is done either arbitrarily or more commonly by organization [CZC11]. The advantage is that, in this approach existing registries can be reused and performance can be better than a central registry if the user knows which registry to query. Although [CZC11][MC12] discuss only centralized and decentralized service registry approaches, [Sim03] presents a hybrid registry whose design resembles the architecture of [LNT16], an example of which is given in Figure 2.4. In this approach organizations may be responsible for their own registry but queries can still be directed to a single authority that is the source of truth for the ecosystem.

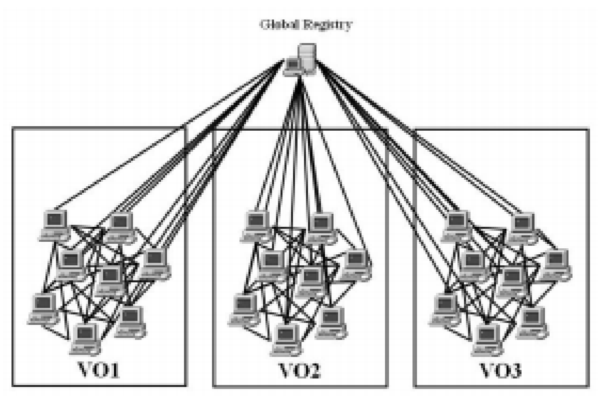


Figure 2.2: Example of a centralized service registry, taken from [Sim03]

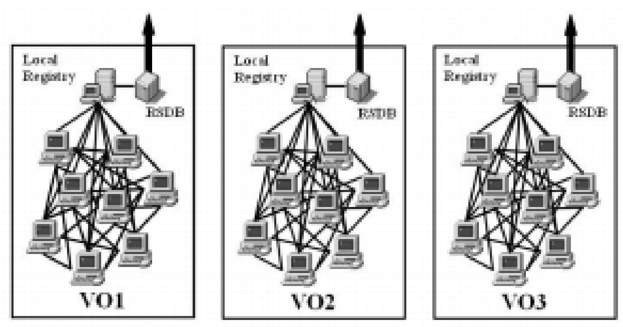


Figure 2.3: Example of a decentralized service registry, taken from [Sim03]

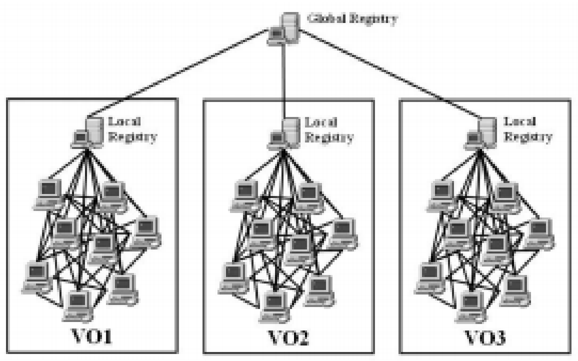


Figure 2.4: Example of a hybrid service registry, taken from [Sim03]

The concept of resource discovery has received much research in IoT where the need to discover resources in a heterogeneous pool was first noticed as a large problem [DCB15]. Therefore, the solutions proposed can fit our problem statement. The authors of [DCB15] and [VC17] have conducted surveys into the different solutions for resource discovery. The two papers conclude that the most popular solution to the resource discovery problem is the use of a centralized registry. There are other solutions such as distributed/P2P discovery, DNS discovery and resource search engines however, the centralized registry accounts for 7 out of the 10 solutions surveyed by [DCB15]. In all the solutions surveyed by the two papers, the interaction protocol in a registry used is REST, or CoAP which uses the same operation verbs as REST but designed for more resource constrained devices.

2.3 Resource Provisioning

In order to deploy resource ensembles, we first need to address the issue of resource provisioning. The resources that we deal with are part of a large pool that is heterogeneous in terms of attributes and methods. We deal with resources across IoT, network functions and cloud services. The heterogeneity between IoT resources has led to research in this area, which we can find in [DCB15] and [VC17]. Therefore, in our work we do not only deal with the differences found within each resource type but across different resource types with their differing infrastructures.

Let us take the example of an instance of MongoDB. Two popular providers of MongoDB are MongoDB Atlas [Inc18b] and MLab [mla18]. Both of these providers provide the same types of resources. However the API provided by each of the respective providers uses different authentication methods, control flows and payloads. In order to provision a MongoDB instance in this case, it is necessary to acquire the knowledgebase to interact with one or both of these providers.

The authors of [TB17] use metadata to describe provisioning parameters and methods. However, the lack of a clear architectural proposal and prototype implementation means that the only contribution that this paper makes to this area makes are concepts with some example forms of resource metadata.

The paper [TCD$^+$15] contributes a system that is capable of deploying IoT resources and cloud services. After analyzing some of their source code which is available open source at `https://github.com/tuwiendsg/iCOMOT` we have confirmed the capability of their solution to deploy various types of resources and cloud services. In order to solve the problem of deploying resources across a heterogeneous resource pool, they use a rule based system that orchestrate the deployment dependencies of resources. The resources are made compatible with the rules and uploaded to a central repository where they can be deployed. In their example to deploy sensor units, the sensor artifacts are uploaded the registry along with the data and configuration scripts.

2.3.1 Resource Configuration

We deal not only with resource configuration at its deployment, but also dynamically at runtime. This feature allows us to modify a resource ensemble to quickly adapt to changing requirements for the user(s) of the resource ensemble.

The authors of [TB17] discuss the need for automated configuration generation based on metadata from resources. Under this principle, a resource provider should attach as much metadata as is necessary to derive a configuration for this resource. Additionally we would need to rely on information services to provide information about different protocols and deployment solutions. This solution would mean that a range of different configuration options could be generated for a single resource. However, the user would need to make a choice about a final configuration.

Like with provisioning, industry standards currently favor configuration based on the resource provider. Some providers will make available a method where a resource can be configured for example through a graphical interface or a RESTful API while other providers will not allow any kind of configuration once a resource has been provisioned. This can be seen in CloudAMQP [84c18a], where once a broker has been provisioned, there is no way to change the name of the broker or reconfigure is resources such as CPU, memory or number of nodes in the cluster. The only operations available are to delete the broker or create new brokers.

However [TN16] proposes that a "single" reconfiguration step could actually be abstracted to a service chain [Ins13] in the case of network function services, where a sequence of actions are applied to a data stream as it passes through an ingress or egress point in a physical or virtual network device, effectively hiding the reconfiguration step in a series of different steps. There is however industry work in this area with many new tools that have been released which have become very popular, we discuss these tools in the related work section.

The Service Mesh concept [Jen18] that we use in our work allows us to avoid an aspect of configuration altogether. Using the service mesh we can abstract the TCP/IP communication layer and avoid directly configuring a resource to communicate with another through the deployment and control of a network of sidecar proxies. The service mesh is a new concept that has been developed by industry and no current academic research exists. However many documentation, white papers and blogs have been dedicated to the topic such as [WM17] or [AS17].

A Service Mesh is a dedicated software infrastructure layer that handles communication between services. It was created to provide a reliable communication layer in a complex topology of services, particularly the kind of complex microservice architecture topologies found in modern day clouds [WM17]. In practice, this layer is implemented with a collection of network proxies that are deployed with applications. The implementation details differ from each organization or developer, the examples that we studied [Jen18][WM17][AS17][FS18] all mention different solutions. But the concept remains the same.

The Service Mesh relies on the concept of sidecar proxies[WM17][AS17][Kle17] that are deployed with the different application instances. The application instances are generally not aware of the services with which they communicate. Furthermore, we can say that each application has no idea of any outbound destination apart from its assigned sidecar proxy. The proxy is responsible for routing that communication from our application to its correct intended destination. The configuration of the proxy determines where communication is routed, there are many ways that this can be achieved but the most important requirement is that the configuration should be able to be changed dynamically. Therefore the application is never aware of any failure in connectivity and can run normally. If a fault is detected then the proxy can choose to retry the request or even switch to a failsafe destination. By doing this, we limit the responsibility of each service in our framework by removing the networking responsibility. This decreases the complexity of orchestrating a large number of services. We can deploy services independent of each other, for example a message consumer can be deployed without the precondition that a running broker is unavailable, the consumer will function as normal (although consume no messages). When the fault is detected(no running broker) a new message broker can be deployed and the consumer's proxy reconfigured to route communication to the new running broker. Furthermore, in the event that the communication topology needs to be reconfigured, only the deployed proxies need to be updated with the new routing policies and the services themselves don't require any reconfiguration.

The authors of [TCD+15] contribute a system that can configure IoT resources dynamically at runtime. This feature is achieved in their system through the implementation of governance processes. When an action is called, the runtime services reconfigure the resource based on the available mechanisms that the resource has to effect this change. For example if the resource implements this capability with runtime variables the runtime services can stop the resource, change the variables and then restart the resource again.

The paper [NVI+15] contributes a governance tool for IoT cloud systems. The tool provides a set of runtime mechanisms that provide among many other governance operations, the capability to reconfigure resources at runtime. The tool makes no concrete assumptions about the implementations of the resources. The governance operations can consist of for example, to query the current version of a service, change a communication protocol, or spin up a virtual gateway. The tool requires that the capabilities of the resource e.g. communication protocols, to be packaged in a certain format.

2.3.2 Resource Monitoring

Like with provisioning and configuration, the current accepted industry standard is to let the provider decide on the monitoring strategy. The provider MongoDB Atlas [Inc18b], CloudAMQP [84c18a] and MLab [mla18] all provide in their own interfaces and their own logging tools to monitor their resources. Google Cloud Platform Stackdriver [Pla17b] is the solely default supported monitoring system available to the resources in Google Cloud Platform [SRR18]. Some newly developed tools exist as a solution to provide monitoring that cross different platforms and infrastructures that we discuss in the related work section.

The paper [MCTD13] has developed a customizable monitoring tool with the objective of analyzing cross-layered, multi-level elasticity of cloud services from whole cloud services to service units, based on structure dependencies. The tool is customizable in the sense that is should be able to collect monitoring information from different types of cloud services that each use their own existing monitoring solutions. In this case, they use a customizable *Data Collector* node that gathers monitoring data from these existing monitoring solutions. This approach of using customizable nodes to gather data from exiting monitoring frameworks has already been used by StatsD [Git18a] an industry solution released open source by Flickr in 2012. Moreover this approach has subsequently been adopted by Telegraf [Inf18] released in 2016 by InfluxData and Fluent Bit [Dat18] which was released in 2015. This approach makes sense given the wide array of existing monitoring solutions available in the current market. Many of the existing monitoring solutions are in fact mandatory for many. A survey from by [FEH+14] has found a total of 21 monitoring tools and frameworks that are currently being used in the cloud alone.

2.4 Related Work

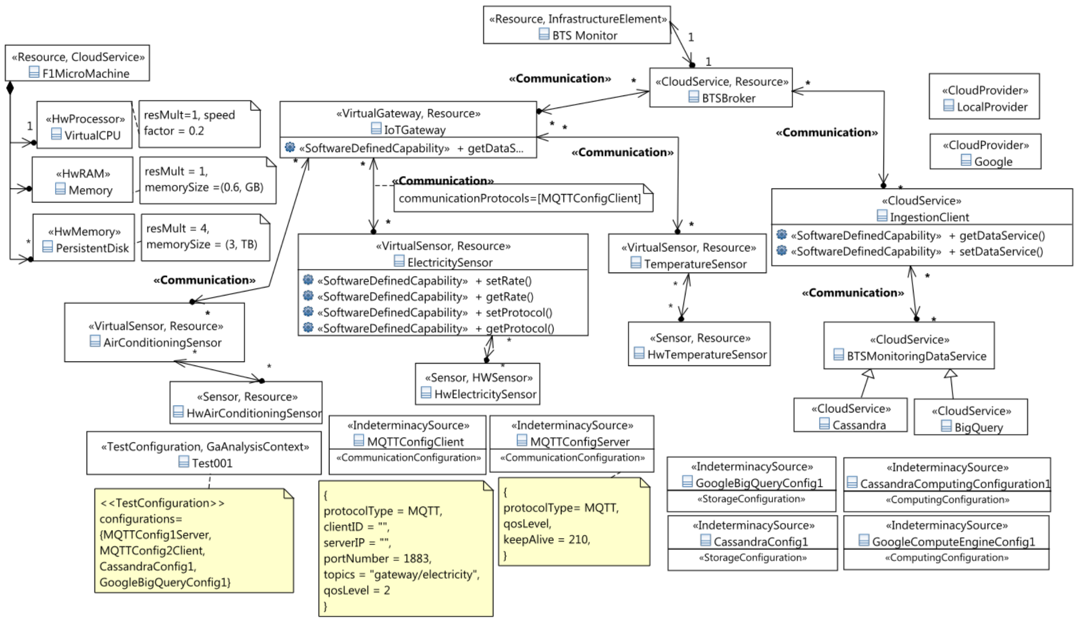
2.4.1 Resource Slices

There has not been a lot of related work that focus directly on resource slices. The authors of [TN16] conclude in their paper that their contribution was a conceptual architecture for the implementation of a framework. Their work relies very closely to [LNT16] whose architecture we did leverage and extend.

The authors of [TB17] suggest a resource slice based approach to uncertainty testing. The choice of their motivating scenario is based on base transceiver stations, which we have leveraged in our work. They took a top-down approach, by analyzing the domain and creating models for different classes of entities that could be identified. Figure 2.5 shows the resulting model that they contribute, where as Figure 2.6 displays their effort to come up with a JSON representation of their entities.

Since the work done in [TB17] was from a purely top down process, the continued work of the authors in [LNT16] makes an implementation (HINC) using certain aspects of the models in Figures 2.5 and 2.6 in Java. Although the implementation does its job by demonstrating the service discovery concepts of different resource models, we had difficulty applying their models and implementations to our use cases

We found that the work done in [TB17] was too refined, that is to say the models are too precise to be applicable on the implementation level. Initially, when trying to use this work as a base for our modeling, we found that it was extremely hard to find all the classes of separation required to describe a deployment configuration, let alone figure out how to parse such a configuration with the large number of possible entities. The JSON example provided in the work was simply a list of documents, while we approved


«Resource, InfrastructureElement» BTS Monitor
«Resource, CloudService» F1MicroMachine
«Communication»
«CloudService, Resource» BTSBroker
«CloudProvider» LocalProvider
«CloudProvider» Google
«HwProcessor» VirtualCPU
resMult=1, speed factor = 0.2
«VirtualGateway, Resource» IoTGateway
«SoftwareDefinedCapability» + getDataS...
«HwRAM» Memory
resMult = 1, memorySize =(0.6, GB)
«Communication»
communicationProtocols=[MQTTConfigClient]
«CloudService» IngestionClient
«SoftwareDefinedCapability» + getDataService()
«SoftwareDefinedCapability» + setDataService()
«HwMemory» PersistentDisk
resMult = 4, memorySize = (3, TB)
«VirtualSensor, Resource» ElectricitySensor
«SoftwareDefinedCapability» + setRate()
«SoftwareDefinedCapability» + getRate()
«SoftwareDefinedCapability» + setProtocol()
«SoftwareDefinedCapability» + getProtocol()
«VirtualSensor, Resource» TemperatureSensor
«VirtualSensor, Resource» AirConditioningSensor
«Sensor, Resource» HwTemperatureSensor
«CloudService» BTSMonitoringDataService
«Sensor, HWSensor» HwElectricitySensor
«Sensor, Resource» HwAirConditioningSensor
«CloudService» Cassandra
«CloudService» BigQuery
«IndeterminacySource» MQTTConfigClient «CommunicationConfiguration»
«IndeterminacySource» MQTTConfigServer «CommunicationConfiguration»
«TestConfiguration, GaAnalysisContext» Test001
«IndeterminacySource» GoogleBigQueryConfig1 «StorageConfiguration»
«IndeterminacySource» CassandraComputingConfiguration1 «ComputingConfiguration»
«IndeterminacySource» CassandraConfig1 «StorageConfiguration»
«IndeterminacySource» GoogleComputeEngineConfig1 «ComputingConfiguration»
1
*

```
<<TestConfiguration>>
configurations=
(MQTTConfig1Server,
MQTTConfig2Client,
CassandraConfig1,
GoogleBigQueryConfig1)
```

```
{
protocolType = MQTT,
clientID = "",
serverIP = "",
portNumber = 1883,
topics = "gateway/electricity",
qosLevel = 2
}
```

```
{
protocolType= MQTT,
qosLevel,
keepAlive = 210,
}
```

Figure 2.5: BTS scenario class diagram taken from [TB17]

of the document based approach (and embraced it) for its machine readability. We found that the fine degrees of separation was not feasible for our implementation oriented work, for example mqtt broker configuration being separate from the description of the broker itself. Furthermore the deployment configuration in Figure 2.6 did no work to try and model the relationships between different JSON objects, which raised problems with how to actually parse it.

The paper [TCD+15] presents a system for "controlling, monitoring and testing IoT cloud systems consisting of both IoT units and cloud services" . Their main work focuses on the elasticity of IoT cloud systems, but there is also a lot of overlap between the concept of resource slice. iCOMOT is capable of deploying resources. Their prototype implementation is capable of deploying a sensor topology along with a message broker for communication. The sensor can be manipulated at runtime to change communication protocol to CoAP. However, due to the focus of the work on the elasticity of IoT cloud systems, the deployment work goes no further than this basic example. Additionally, there doesn't seem to be a simple or extensible way to describe more kinds of resources to the system for deployment. Their website `http://tuwiendsg.github.io/iCOMOT/demo.html#sensorUtil` only contains one example deployment consisting of sensors and mqtt brokers. There also is no way to define the entire IoT cloud system like we propose to with the resource slice. The descriptions between the sensor and broker in their example are separate, and the only kind of relationship between the two resources must be configured manually by the user.

```
"BTSBroker1Local":  {
   "name": "BTSBroker",
    "cloudProvider": ["local"],
    "communicationConfigs": ["MQTTConfig1"],
    "type": "CloudService"
}
"BTSBroker2Google":  {
   "name": "MQTTBroker",
    "cloudProvider": ["Google"],
    "communicationConfigs": ["MQTTConfig1"],
    "type": "CloudService"
 }
"MQTTConfig1":  {
        "name": "MQTTConfigServer",
        "protocolType": "MQTT",
        "qosLevel": [],
        "keepAlive": 210,
        "type": "CommunicationConfiguration"
}
"MQTTConfig2":  {
       "name": "MQTTConfigClient",
        "protocolType": "MQTT",
        "clientID": "",
        "serverIP": "35.189.187.208",
        "portNumber": 1883,
```

Figure 2.6: Example of deployment configurations taken from [TB17]

2.4.2 Resource Discovery

The paper [LNT16] provided us with a base for the information models that we use in our final framework. The paper focused almost exclusively on harmonizing various resource data into a distributed model. The contributions of this paper were crucial in our work, which is mostly a practical systems-oriented implementation. Figure 2.7 shows an overview of the distributed model.

In Chapter 4, we develop our information models for resources and resource providers based on the work in Figure 2.7. We do not leverage these models directly, but we do leverage the concept and extend them. The authors of [LNT16] also introduce two key components into the architecture: Global and Local Resource services. These resource services serve to bridge and interface different IoT resources and provide higher level access to their information. The architecture crucially uses an adaptor system to marshal data from heterogeneous resource providers into the information models shown in Figure 2.7. The authors implemented a prototype of their proposed system design. Our work leverages the source code of their prototype and extends it to include the other features that involve resource provisioning, configuration and management. We also leverage and extend the architecture provided by [LNT16] which is illustrated in Figure 2.8.

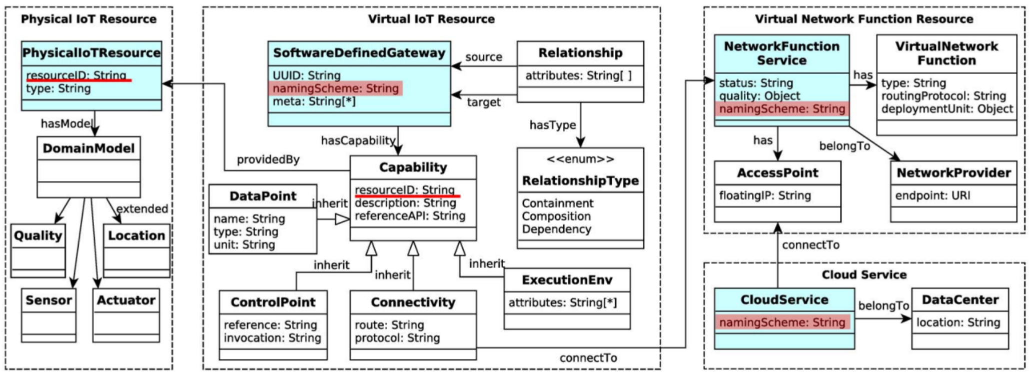


Figure 2.7: The inherited information model taken from [LNT16]

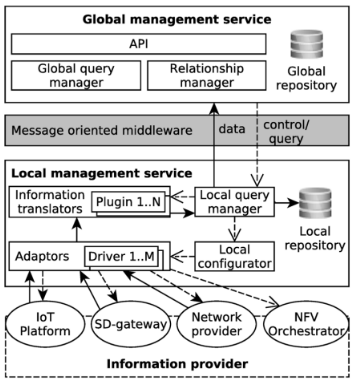


Figure 2.8: The inherited architecture inherited from [LNT16]

From our research into service discovery, we have found that the most popular solution to service discovery is the use of a central registry that is queryable [DCB15] [VC17]. Furthermore, the most popular service discovery tools in industry such as Consul [Has18], Zookeeper [Apa18b] and etcd [Cor18] all work on the basis of a central registry where services are registered. The registered services can then be queried through protocols like REST as is the case with Consul and etcd or through a client based API with Zookeeper.

Since the majority of solutions that we researched made use of a central registry, we leverage this concept to resources. Our framework registers resources into our own central registry and opens up query to a HTTP REST API.

2.4.3 Resource Provisioning

Since a lot of the research related to our work is based on directly implementing resource slices, resource management is slightly neglected with authors of papers dealing directly with the implementation of resource slices. However, the provisioning has been treated in literature and the authors of [KR15] specifically deal with an overview of the IoT Cloud technology stack and discuss potential technologies that could lead to on-demand provisioning. While not providing a solution to this problem, the paper clearly introduces the problem and its different parts. The challenges of the non-heterogeneity of IoT resources is presented as well as the recent developments in Infrastructure as a Service (IaaS) and Software as a Service (SaaS). The paper suggest a high level middleware along with a higher level abstraction of resources as basis of a solution. We leverage both of these concepts in our final system design.

Furthermore, due to the rise of large distributed platforms and an increased reliance of DevOps, we can identify a lot of very mature provisioning tools developed by industry. One category of tools that handles our aspect of resource provisioning are continuous deployment tools.

Netflix's Spinnaker tool [OSS18b] was released in 2017 as an open source multi-cloud continuous delivery platform for releasing software. Spinnaker uses a pipeline method for deployment that consists of different stages (actions) to execute before moving on to the next. By default Spinnaker can integrate with the most popular cloud deployment solutions including AWS EC2, Kubernetes, Google Compute Engine, Google Kubernetes Engine, Google App Engine, Microsoft Azure. Further deployment solutions can be integrated into spinnaker by way of custom stages, in other words code that is integrated into Spinnaker through its API.

Similarly to Netflix's Spinnaker, which is a relatively new tool, there is also the better established Jenkins tool [OSS18b][OSS18a]. Jenkins is also a tool developed for continuous delivery of software. However it is a lot more low-level than Spinnaker. The different pipeline stages in Jenkins are implemented with code and Jenkins itself has no support for any cloud provider, although there is a very large market place for 3rd party plugins that can support deployment for various cloud platforms.

Continuous delivery tools, can be very useful in the provisioning aspect of our problem. The pipeline concept is very powerful and most continuous delivery tools including Spinnaker, Jenkins, Buddy Works and Chef [Bud18][Che18] all allow custom functions to be executed when a deployment environment is not supported. The custom functions can serve as interfaces to the operations of resource providers. Our solution is not unlike the pipeline method for resource provisioning since we also rely on the development of custom implemented functions that serve as common interfaces to different resource providers.

However, while the ease of deployment and high automation are very attractive features, there are also very necessary features that are missing. Firstly, there is no configuration management built into these systems. The pipeline concept actually encourages the developer to use external tools for configuration management. Moreover, there is no service discovery or service orchestration layer at all. This is understandable since the goal of these tools is to provide deployment options for your infrastructure and not to manage it. Furthermore, monitoring features are also lacking with these tools. Therefore, we leverage the concept of using custom functions to abstract provider level details that these tools endorse but choose to implement our own solution in order to have a wider feature base.

2.4.4 Resource Configuration

Dynamic resource configuration is also a very popular problem that is being addressed with many new service orchestration tools that support service discovery for service configuration. These features could potentially apply to the resource configuration aspect of our problem.

Consul [Has18] is a service orchestration tool that provides service discovery and service configuration. The service discovery feature, relies on deploying a server-like service registry so that services can query the registry to locate upstream services. The service configuration feature is a central store that supports transactional access in order to provide access on a distributed scale in real time. The configuration is stored in a hierarchal key/value store. While Consul does have some interesting service orchestration features, in reality they are not well adapted to our use case. Current service orchestration tools like Consul, Zookeeper or etcd [Apa18b] [Cor18][Has18] rely on the fact that the services being deployed are all owned by the system. The service discovery mechanisms all require the query method to be built into the software. For example, Spring Boot has a consul library that integrates with the Spring library to automate the service discovery and configuration processes so that the HTTP API does not have to be used. The etcd tool, has an even weaker service discovery feature which relies on DNS queries. Since the resources that we intend to manipulate in our problem involve third-party owned resources, these current service orchestration tools are of little use. In Chapter 4 we discuss the how the concept of a Service Mesh deals with this issue.

While these tools market dynamic service configuration, they are in reality a simply a key/value store that can be queried for correct configuration. The actual reconfiguration needs executed either by the developer or the use of other tools. We can envision the continuous deployment tools we discussed earlier (Spinnaker, Jenkins) implementing specialized pipelines to check for configuration changes and apply them to different resources. Our framework does not only store the configuration in our central registry, but can also effect the configuration process on the resources.

Regarding the representation of resources and configurations, the work done by [TB17] using the models in [LNT16] would require a specific protocol for configuring each resource. This would depend on its configuration requirements because there is no harmonization of a resource and its configuration by the authors. In our work we go further than [TB17] and include the resource's configuration in its representation. Therefore, we may apply the same configuration procedure to each resource.

2.4.5 Resource Monitoring

Fluent Bit [Dat18] is an open source and multi-platform log processor and forwarder which allows users to collect data/logs from different sources. The tool collects logs using different input plugins that can be implemented by any third party through the tool's API. The tool also uses powerful language parsers to extract and tag important log information. Then, data can be filtered for different configurable parameters. Another set of output plugins can also be implemented to deliver the final data records to different destinations.

Another similar monitoring agent is Telegraf [Inf18] which is developed by InfluxData as a part of their TICK stack for dealing with real time streaming information. Like Fluent Bit, Telegraf uses a plugin architecture to manage input and output of data. Processor and aggregator plugins can also be used to transform the data and create aggregated metrics.

These kind of monitoring agents such as Fluent Bit [Dat18], Telegraf [Inf18] or StatsD [Git18a] are a new type of monitoring agent that have been developed in recent years due to the variety of monitoring solutions that distributed systems now use. This new kind of monitoring agent focuses on pulling monitoring data from different sources and processing them to make a set of unified records. While these tools are very powerful for the monitoring aspect of our problem, they do not have any other feature, as they are very specialized tools. The most valuable feature of these monitoring agents, is the flexibility that they provide in monitoring a heterogeneous pool of resources through the collector plugin concept. With this concept tools like Fluent Bit and Telegraf are able to monitor an entire pool of resources that do not share similarities in operating or deployment environment and be able to unify the monitoring data. This concept has been shown by other related works such as [LNT16] which also uses a plugin concept to collect uniform data from a pool of resources. However, we do not directly use these tools in our work because of current industry practices. The largest cloud providers such as Google Cloud Platform [SRR18] and even smaller providers such as MLab [mla18] and CloudAMQP [84c18a] require direct access to their APIs for monitoring services on their resources. Since we generally deal with a lack of access the technical specifics of our deployed resources we would have difficulty in using the monitoring agents for our work.

2.5 Summary

In this chapter, we discussed current and existing research into the area of resource slices. We also presented several different types of tools: deployment tools, monitoring agents and service orchestration tools that can solve different aspects of our problem statement. We discuss the features that are available in these tools and evaluate their usefulness as solutions to our deployment problem.

CHAPTER 3

Motivation Scenarios and Use Cases

3.1 Overview

In this Chapter we present our choice of motivating scenario which is based on a real world problem that could be addressed by our proposed solution. We identify the different stakeholders involved in this scenario as well as formalize use cases for them. Finally, we derive some functional requirements of our framework along with criteria for evaluation.

3.2 Scenario: Monitoring Infrastructures of Base Transceiver Stations (BTS)

Our scenario is based on a monitoring system for the infrastructures of BTSs which we briefly introduced in Chapter 1. A BTS is a piece of equipment that facilitates wireless communication between user equipment and a network. Such devices can include mobile phones, tables or computers with wireless internet connectivity. The network can be that of any of the wireless communication technologies such as GSM(2G), WCDMA(3G), Wi-Fi or any other wide area network (WAN) technology. The BTS equipment are usually housed in a shelter which protects the telecoms equipment from external conditions such dust, corrosion or adverse weather conditions.

Our scenario monitors the infrastructure of BTSs. The infrastructure concerns the various metrics that keep a BTS operational such as temperature, humidity, battery capacity, power, voltage and current. These metrics are not directly involved in the primary function of a BTS, providing wireless communication. BTS infrastructure is monitored by an array of sensors which sensors can be installed such as external temperature sensors [Ala18].

```
id,reading_time,value,station_id,parameter_id
6378227581,2017-10-23 12:01:05,2.4,1161114060,161
6378229354,2017-10-23 12:01:32,24,1161114039,116
6378224373,2017-10-23 12:01:15,61,1170410000,115
6378223549,2017-10-23 12:00:55,6,1161114044,122
```

Listing 3.1: "Sample of BTS data from `https://github.com/rdsea/IoTCloudSamples/tree/master/data/bts`"

These sensors detect the infrastructure and operating environment of the BTS, which can be used for various big data analysis of BTS infrastructures. Additionally, the sensors also facilitate the monitoring of the BTS for faults and alarms. These sensors produce data that is sent to an IoT gateway that is forwarded to a message broker. In our scenario we have procured real data from BTS sensors which include recorded metrics on temperature, humidity and battery voltage and example of which can be seen in Listing 3.1 which contains four data points for battery capacity, temperature, humidity and battery voltage all differing by the *parameter_id*. This data was obtained from BachPhu, a company developing IoT solutions in Vietnam

Further processing of this data occurs through different software artifacts with analytical capabilities. The processed data is then stored in a non-relational data store adapted for Big Data processing. A functional overview can be seen in Figure 3.1.

In the scope of our work, our scenario manifests itself in the following components:

- Data sources, in other words the BTS sensors that can collect different environmental and infrastructural metrics such as temperature, humidity and voltage. These sensors use IoT gateways to forward the data to message brokers.
- Message brokers that are responsible for the publishing of data to further software components that will use the raw sensor data.
- Ingestion clients process and analyze the data. Naturally, there can be several software artifacts that could effect a number of operations on the data, for the sake of simplifying we only use one.
- Data sinks which store or consume the data that has passed through the data pipeline.
- Firewalls can also be used to secure access to certain resources like the data source, in order to secure it, for example only authorizing one destination for the data so that it does not get sent to the wrong client. This network resource is cross cutting and can be used at any point in the data pipeline.

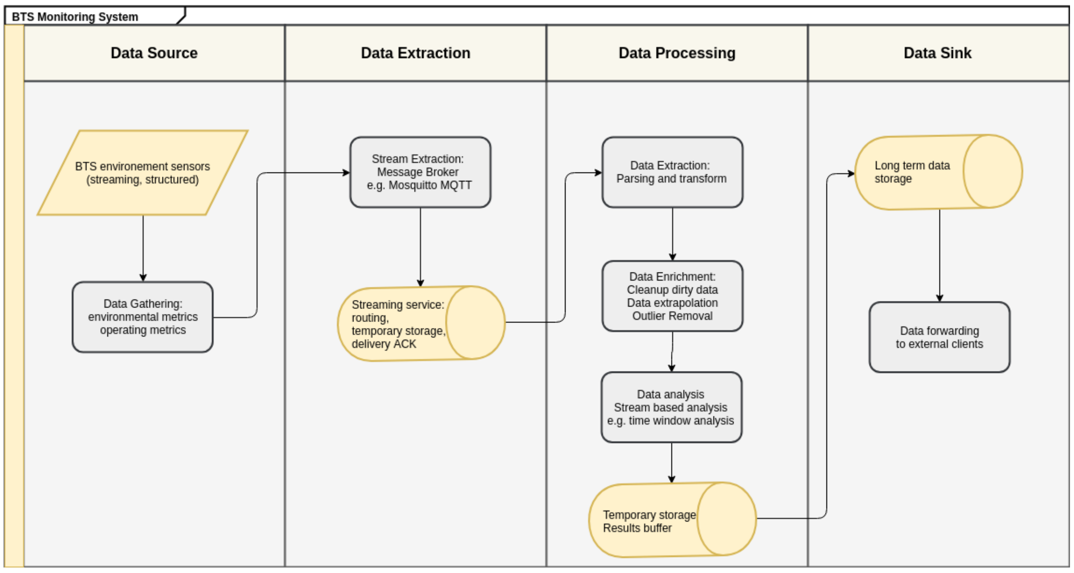

BTS Monitoring System
Data Source
Data Extraction
Data Processing
Data Sink
BTS environement sensors (streaming, structured)
Data Gathering: environmental metrics operating metrics
Stream Extraction: Message Broker e.g. Mosquitto MQTT
Streaming service: routing, temporary storage, delivery ACK
Data Extraction: Parsing and transform
Data Enrichment: Cleanup dirty data Data extrapolation Outlier Removal
Data analysis Stream based analysis e.g. time window analysis
Temporary storage Results buffer
Long term data storage
Data forwarding to external clients

Figure 3.1: A functional overview of the BTS Monitoring System scenario

3.2.1 Stakeholders

In our scenario there are four stakeholders present that interact with the BTS Monitoring System.

- **Data Consumer** - In other words, the client of our BTS Monitoring System. This is an stakeholder that consumes the output of the big data pipeline. The Data Consumer will search and visualize the final data output for profit to use in other business related applications.

- **Service Operator** - The Service Operator manages the monitoring system. He is responsible for accepting data access requests from data consumers and allocating data sources to different data pipelines. Furthermore, the Service Operator also has the relevant permissions to configure the firewall to protect different parts of the system, for example, data sources or data sinks.

- **Data Engineer** - The Data Engineer creates different algorithms and methods to analyze and process the data received from the data sources. The analysis phase enriches the data and adds value. The value created by the Data Engineer is what the Data Consumer looks for when deciding to use the System's services. Additionally, the Data Engineer can also choose to deploy custom data processing methods into the data pipeline at the request of the Data Consumer.

- **Developer** - The Developer is responsible for maintaining the system as well as developing new features. The main responsibility of the Developer in this scenario is to provide interoperability solutions in the form of deployable software artifacts into the data pipelines when required.

3.3 Use Cases

We use our chosen scenario to describe several use cases of our framework. These use cases help will help us identify key architectural and design points and make sure that we do not work outside the scope of our problem statement. We aim to achieve all these use cases with our proposed framework and detail the usage of our framework with respect to several use cases in the evaluation chapter of this thesis. Figure 3.2 gives an overview of the stakeholders involved in the BTS scenario as well as the different use cases that the BTS scenario must support.

We further document our scenario's use cases in Tables 3.1 to 3.7.

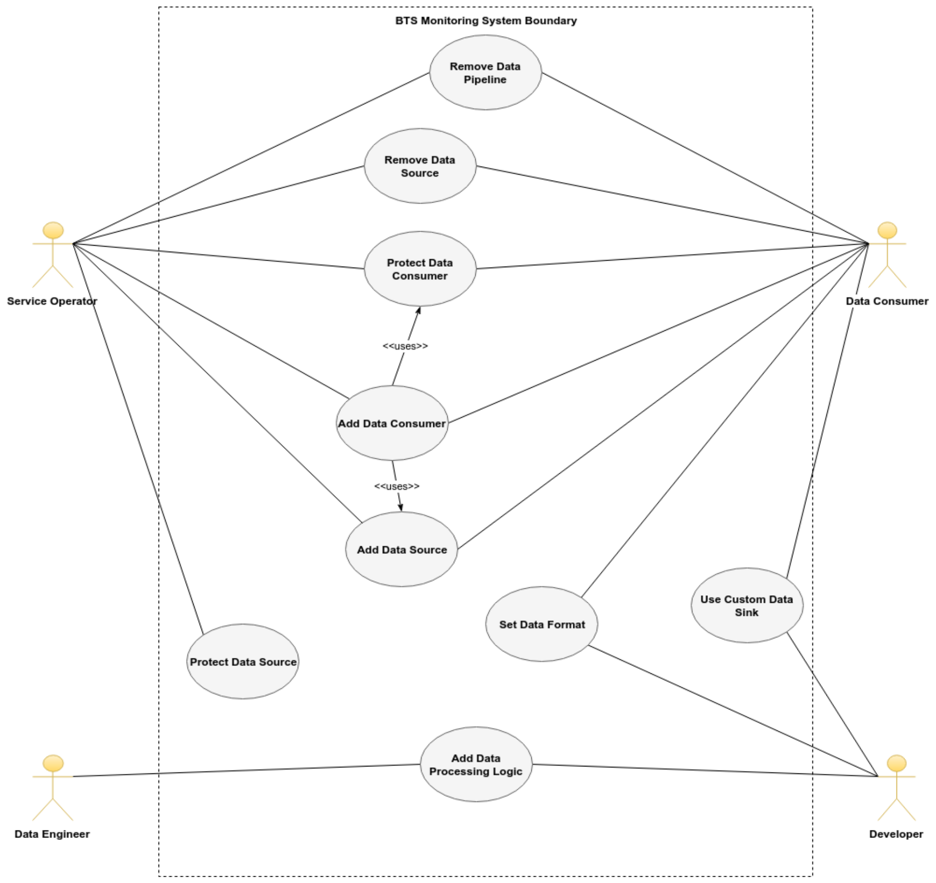


Figure 3.2: BTS Monitoring System Use Cases

ID	UC01
Title	Add Data Consumer
Stakeholders	Data Consumer, Service Operator
Problem Statement	A new client would like to consume the analyzed data from the BTS monitoring System.
Preconditions	none
Scenario	1. The Data Consumer wants to consume BTS sensor data 2. The Data Consumer requests the Service Operator for data access 3. The Service Operator creates a data pipeline in the System for the Data Consumer 4. The Service Operator adds the data sources selected by the Data Consumer to the pipeline *Uses use case UC02* 5. The Service Operator adds the level of network protection the Data Consumer requires through firewall settings *Uses use case UC03* 6. The Data Consumer receives instructions to fetch data from the data sink
Successful End States	1. A data pipeline exists for the Data Consumer 2. The Data Consumer receives the data from his/her selected data sources 3. The Data Consumer has the appropriate network firewall protection for his/her pipeline

Table 3.1: UC01: Add Data Consumer

ID	UC02
Title	Add Data Source
Stakeholders	Data Consumer, Service Operator
Problem Statement	A Data Consumer wants to add new data sources to his/her data pipeline
Preconditions	1. The Data Consumer has successfully made a data request with the Service Operator 2. There already exists a data pipeline for the Data Consumer
Scenario	1. The Data Consumer selects data sources from a catalogue 2. The Data Consumer sends selected data sources to Service Operator 3. The Service Operator adds the selected Data Sources to the Data Consumer's data pipeline
Successful End States	1. The Data Consumer receives the data from his/her selected data sources

Table 3.2: UC02: Add Data Source

ID	UC03
Title	Protect Data Consumer
Stakeholders	Data Consumer, Service Operator
Problem Statement	A Data Consumer would like network protection through a firewall for his/her data sink
Preconditions	1. The Data Consumer has successfully made a data request with th Service Operator 2. There already exists a data pipeline for the Data Consumer
Scenario	1. The Data Consumer chooses firewall rules for his/her data sink 2. The Data Consumer sends firewall rules to the Service Operator 3. The Service Operator configures the firewall with the new rules on the Data Consumer's Data Sink
Successful End States	1. The Data Consumer's data sink can only be accessed externally according to the firewall rules

Table 3.3: UC03: Protect Data Consumer

ID	UC04
Title	Add Data Processing Logic
Stakeholders	Data Consumer, Data Engineer
Problem Statement	The default data processing logic of the system is not adapted to the Data Consumer's needs. The Data Consumer consults with the Data Engineer to implement a custom data processing logic for the data pipeline.
Preconditions	1. The Data Consumer has successfully made a data request with th Service Operator 2. There already exists a data pipeline for the Data Consumer 3. The Data Consumer has added data sources
Scenario	1. The Data Consumer informs the Data Engineer of his/her constraints 2. The Data Engineer develops a new algorithm or method to process the BTS sensor data 3. The Data Engineer deploys a custom software artifact into the pipeline to process the BTS sensor data
Successful End States	1. The Data Consumer receives data processed by the new data processing logic

Table 3.4: UC04: Add Data Processing Logic

ID	UC05
Title	Set Data Format
Stakeholders	Data Consumer, Developer
Problem Statement	The default data format of the system's output is not adapted to the Data Consumer's requirements. The Data Consumer requests the Developer to change the data format of his/her pipeline.
Preconditions	1. The Data Consumer has successfully made a data request with the Service Operator 2. There already exists a data pipeline for the Data Consumer 3. The Data Consumer has added data sources
Scenario	1. The Data Consumer informs the Developer of his/her chosen data format 2. The Developer searches for an existing data transform function. If none is found, the Developer creates a new transform function and keeps it for future use 3. The Developer deploys a custom software artifact into the pipeline to transform the BTS sensor data output format before it reaches the data sink
Successful End States	1. The Data Consumer receives data in his/her chosen format

Table 3.5: UC05: Set Data Format

ID	UC06
Title	Use Custom Data Sink
Stakeholders	Data Consumer, Developer
Problem Statement	The Data Consumer wishes to receive the BTS sensor data in a data sink that is not normally supported by the system.
Preconditions	1. The Data Consumer has successfully made a data request with th Service Operator 2. There already exists a data pipeline for the Data Consumer 3. The Data Consumer has added data sources
Scenario	1. The Data Consumer informs the Developer of his/her data sink and provides any interoperability and authentication data 2. The Developer creates a temporary data sink for the BTS data that will forward the data to a new data sink. 3. The Developer deploys the custom data sink to the data pipeline
Successful End States	1. The Data Consumer receives data in his/her data sink

Table 3.6: UC06: Use Custom Data Sink

ID	UC07
Title	Protect Data Source
Stakeholders	Data Consumer, Developer
Problem Statement	The Service Operator has detected malicious attack or usage of a data source and wishes to limit access to this data source
Preconditions	1. There is malicious attack or usage associated with he data source
Scenario	1. The Service Operator identifies authorized and unauthorized access to the data source 2. The Service Operator creates firewall rules to deny unauthorized access to the data source on a white list basis 3. The Service Operator applies the firewall rules to the data source.
Successful End States	1. The data source can only be used by white listed hosts.

Table 3.7: UC07: Protect Data Source

ID	UC08
Title	Remove Data Source
Stakeholders	Data Consumer, Service Operator
Problem Statement	A Data Consumer wants to remove a data source from his/her data pipeline
Preconditions	1. The Data Consumer Has successfully made a data request with the Service Operator 2. There already exists a data pipeline for the Data Consumer
Scenario	1. The Data Consumer Selects a data source from his/her pipeline 2. The Data Consumer sends the selected data source to the Service Operator 3. The Service Operator removes the selected data source from the pipeline
Successful End States	1. The Data Consumer no longer receives data from the deleted data source.

Table 3.8: UC08: Remove Data Source

ID	UC09
Title	Remove Data Pipeline
Stakeholders	Data Consumer, Service Operator
Problem Statement	A Data Consumer no longer wishes to use the system and wishes to remove his/her data pipeline
Preconditions	1. The Data Consumer Has successfully made a data request with the Service Operator 2. There already exists a data pipeline for the Data Consumer
Scenario	1. The Data Consumer requests the Service Operator he no longer wishes to use the data pipeline 2. The Service Operator frees all resources associated with the pipeline
Successful End States	1. The data pipeline no longer exists.

Table 3.9: UC09: Remove Data Pipeline

3.4 Resource Ensemble Approach

The Big Data BTS system enables multiphase data pipeline where the output of one resource becomes the input of the next. However, each user can choose to either use the entire system or only a subset. For example, a user could deploy an entire system from data source to data store and consume the analyzed data from a data sink, or simply choose to deploy a few sensor resources along with a broker and use his/her own data consumer for further processing.

Therefore, the on-demand aspect becomes an important requirement of implementing this scenario. Additionally, modifications to the monitoring system should be made at runtime in function of changing user requirements. In the scope of our scenario for example, this could be adding more sensors that publish different metrics on the BTSs or creating a more complex data pipeline that puts different BTS metrics in their separate data stores.

In order to satisfy the on-demand aspects of our scenario, we take a resource ensemble based approach by treating the BTS data pipeline as a resource ensemble where each of the pipeline components is a resource that is part of an ensemble. By following Figure 3.1, we classify each component of our BTS monitoring system as a resource. We assume that each resource can be deployed separately. Additionally there exists a different resource provider that is responsible for provisioning the resource, and managing it during its lifecycle. We present the breakdown of resources in our scenario in Table 3.10.

	IoT Resource	Network Function Service	Cloud Service
Data Source	✓		
Message Broker		✓	✓
Ingestion Client			✓
Data Sink		✓	✓
Firewall		✓	

Table 3.10: A breakdown of the resource types in our scenario

Although this system that we describe could be the property or be maintained by the same organization, industry examples show that varying levels of ownership is far more likely.

Firstly, sensor ownership is very common in industry. Moreover, sensors can be obtained by many individuals for cost effective platforms such as the Raspberry-Pi or Arduino through Grove [See18]. Many large organizations already release their sensor data publicly for free such as Microsoft's T-Drive taxi data that contains one-week trajectories of 10,357 taxis [YZ11]. Other datasets can be bought, for example, from the databrokers market [dD18]. According to current research [Per17] [PZCG13] the sensor as a service model is very feasible and several trial projects have already begun. Therefore, we can easily envision a situation where sensors can be owned by an organization that provides/sells access to them on-demand.

Message broker services are also very common with the cloud. Cloud AMQP [84c18a] was first released in 2012 that offered managed instances of AMQP brokers as a service. Additionally there also exist public message broker instances such as MQTT brokers provided by HiveMQ. Recently with the popularity of docker, a simple broker can be deployed by an organization or an individual very easily with one command either on the cloud or on a edge node.

Software artifacts are generally implemented by developers for specific domain uses. The improvement of deployment solutions such as docker or using easy distribution techniques like maven mean that it can be very easy to provide software as a service by an organization or an individual. Furthermore tools like Node-RED [Fou18a] provide a simple tool to wire different software functions together.

Data stores are one of the most common types of services offered by a wide range of providers such as Google BigQuery, MongoDB Atlas or Amazon RDS [Goo18b][Inc18b][Ama18]. These providers all have some kind of management API through RESTful interfaces.

Therefore, we assume that each resource can be deployed separately by a different resource provider. In order to for a user to successfully deploy a configuration of a BTS system, the stakeholders must interface with the different providers for each of the resources that we described in our system.

As we stated in Chapter 1, we tackle the problem of a heterogeneous set of resources and resource providers. Without our proposed framework, the stakeholders must discover the relevant resource providers, which could be could be done with other tools. However, a stakeholder must be aware of the different API calls and conventions used by each resource provider to be able to configure or monitor resources (we illustrate these problems in Figure 1.2). Our proposed framework aims to tackle the problems faced in this scenario which were also mentioned in the introduction chapter of this thesis.

Therefore, our goal is to implement our motivating scenario using a resource ensemble based approach through our proposed deployment framework. The next section deals with the requirements of our proposed framework.

3.5 Requirements

In this section we specify the requirements that our proposed framework must fulfill in order to reach a desired level of functionality and achieve our stated use cases. These derived requirements will focus development and reduce development overhead by driving a lean framework design. As stated in the research of the State of The Art in Chapter 2, there is a lack of related tools and current work into an actual resource deployment framework. Therefore, we exclude a set of non-functional requirements in order to focus work on a functional framework. We do however evaluate the non-functional quality of performance in the Chapter 6.

Since we now discuss the requirements for our proposed framework we identify the two primary stakeholders:

- **Ensemble User** - This stakeholder represents the user of our proposed framework. The Ensemble User will interact with our framework to provision and manage resource ensembles. Regarding our motivating scenario, the Ensemble User can be stakeholders of our scenario who interact with our framework to fulfill their assigned use cases.
- **Resource Provider** - This stakeholder is responsible for provisioning and configuring resources that are used in the resource slices of our scenario. This stakeholder does not necessarily have to be a person. The Resource Provider could also be an organization or a platform that manages a resource that can be utilized in resource ensembles.

We state that the proposed framework will fulfill the following requirements. These requirements are specified using a modified version of the Volere requirement shell [MS05]:

3.5.1 Data Harmonization

Our work tries to deploy different resources from various third party resource providers. These various resource providers will have different representations of their resources as well as different management methods through their respective APIs. In order to manage so many different protocols and models in one framework, we must place requirements in place for harmonizing data into a one unified representation. By making data harmonization a requirement, we facilitate a set of common operations that can be used to interface with various resource providers and also facilitate interoperability of our operations with external systems.

ID	DH1
Title	Resource Data Harmonization
Description	Our framework will need to effectively manage resources with diverse capabilities. These resources also implement various communication protocols. The pool of resources that we manage is heterogeneous, even among those resource pools that are homogeneous in functionality. For example, when choosing, IoT sensor devices, there is a large degree of heterogeneity in data format, communication protocol or security protocols to name but a few. Since our objective is to treat different types and categories of managed resources in the scope of end-to-end resource ensembles, we need a high level model that harmonizes the low-level information and configuration parameters required by resource providers for various types of resources. This model must be extensible to cater not only information provided by resource providers, but also additional metadata that can be generated to support other logistical functions such as interoperability.
Stakeholders	Resource Provider

Table 3.11: Data Harmonization Requirement

We achieve this requirement by using the information models from [LNT16] as a starting point. The approach taken by [LNT16] is top-down while we take a bottom-up approach. Therefore we expect to make modifications to them for use in our framework.

3.5.2 Resource Management

In order to manage multi-resource ensembles, we must first establish a set of requirements that govern the management of individual resources. We use resource management operations as the building blocks for managing multi-resource ensembles.

ID	RM1
Title	Resource Discovery
Description	Our framework aims to interface with many different resource providers to manage their resources. An end user of our framework needs to efficiently identify the resources that conform the best to his/her requirements. Due to the heterogeneous nature and the large size of our resource pool, we need a discovery functionality that facilitates detailed queries. This requirement uses the previous data harmonization requirement, so that the resource discovery is structured and systematic for all the resources managed by the proposed framework.
Stakeholders	Resource Provider, Ensemble User
Dependencies	DH1

Table 3.12: Resource Discovery Requirement

ID	RM2
Title	Resource Provisioning
Description	Since our problem statement involves the provisioning of resource ensembles, we must be able to provision the resources that form our resource ensemble. The resource should be provisioned directly from our framework without any direct provider interaction from the user's end.
Stakeholders	Resource Provider, Ensemble User
Dependencies	DH1

Table 3.13: Resource Provisioning Requirement

ID Title	RM3 Resource Configuration
Description	A resource should be configured when it is provisioned by our framework. Furthermore, we aim to provide dynamic configuration to resources. The configurable parameters should be made known to the user and the resource and used without any direct provider interaction by the user. This requirement allows us to modify resource ensembles based on changing user requirements.
Stakeholders	Resource Provider, Ensemble User

Table 3.14: Resource Configuration Requirement

ID	RM4
Title	Resource Deletion
Description	Once a resource has been used adequately by the user who provisioned the resource, or when the resource is no longer of use, the user should be able to delete this resource. The deleted resource no longer appears in the framework. However, the framework handles the deletion on a purely superficial level, the actual physical resource provider side may be dealt with according to the best practices and policies of the resource provider.
Stakeholders	Resource Provider, Ensemble User
Dependencies	DH1

Table 3.15: Resource Deletion Requirement

ID	RM5
Title	Resource Monitoring
Description	Our proposed framework should be able to provide monitoring information about each resource, or notify the user of a lack of monitoring information. The monitoring information available for a resource should be provider specific (i.e. what the provider can tell us). This information can give the user a diagnostic view of his/her resource ensemble and can provide useful data for any fault detection or error-handling analytics task related to resource ensembles.
Stakeholders	Resource Provider, Ensemble User

Table 3.16: Resource Monitoring Requirement

Since we inherit the architecture from [LNT16] and extend it. We make use of the adaptor pattern of the architecture and extend it to cover the extra resource and provider operations. In our implementation we leverage and extend only the architecture. In our implementation use opt for a more distributed approach which is presented in 5.

3.5.3 Slice Management

The goal of our work is to provide a framework that allows on-demand dynamic deployment of resource ensembles. Therefore we detail a set of requirements that will govern the operations required to manage end-to-end resource ensembles. These requirements are built upon the resource management requirements that we have previously established as building blocks to managing multi-resource ensembles.

ID	SM1
Title	Slice Provisioning
Description	The framework should be able to provision a resource ensemble. A set of resources specified by the user shall be submitted to the framework. The framework shall then proceed to provision all of the resources specified by the user for this resource ensemble through the framework's resource management operations.
Stakeholders	Resource Provider, Ensemble User
Dependencies	RM2, RM3

Table 3.17: Slice Provisioning Requirement

ID	SM2
Title	Slice Configuration
Description	The framework should be able to reconfigure an existing resource ensemble. The user shall be able to submit an updated specification of a resource ensemble, that may be partially or totally different from the original ensemble to the framework. The framework shall make the appropriate changes to the resource ensemble such as provisioning new resources, changing inter-resource relationships or deleting resources that are no longer relevant to the updated ensemble using the resource management operations previously specified.
Stakeholders	Resource Provider, Ensemble User
Dependencies	RM2, RM3, RM4

Table 3.18: Slice Configuration Requirement

ID	SM3
Title	Slice Deletion
Description	The framework should be able to delete a ensemble upon user command. The resources of the ensemble should be deleted using the resource management operations previously specified.
Stakeholders	Resource Provider, Ensemble User
Dependencies	RM4

Table 3.19: Slice Deletion Requirement

ID	SM4
Title	Slice Monitoring
Description	The framework should be able to provide as much monitoring information as possible about the status of resource ensemble. This monitoring data can take the form of logs, resource usage or network connectivity information of the different resource in the ensemble. When monitoring data is not available for certain resources in the ensemble, the situation should be notified to the user. Any incident detection or interoperability warnings are not in the scope of this requirement.
Stakeholders	Resource Provider, Ensemble User
Dependencies	RM5

Table 3.20: Slice Monitoring Requirement

In order to fulfill the slice management requirements for our framework, we build a tool in our framework that leverages the features of the resource requirements. This tool should rely on the resource operations to manage the resources in the slice. The novel feature in this new tool is to manage inter-resource relationships that give context to the individual resources to form a resource slice.

Table 3.21 shows a summary of our derived requirements mapped to the use cases. In the evaluation phase we discuss how we developed our features can be used to carry out our chosen use cases.

Use Case/Req.	UC01	UC02	UC03	UC04	UC05	UC06	UC07	UC08	UC09
DH1	✓	✓	✓	✓	✓	✓	✓		
RM1		✓		✓	✓	✓			
RM2	✓	✓	✓	✓	✓	✓	✓		
RM3				✓	✓	✓			
RM4								✓	✓
RM5			✓				✓		
SM1	✓								
SM2		✓	✓	✓	✓	✓	✓	✓	
SM3									✓
SM4			✓						

Table 3.21: Overview of requirements with respect to use cases

3.6 Summary

In this chapter we have developed a set of use cases that are based on the realistic scenario of a BTS monitoring system. Our use cases involve the provisioning and management of big data pipelines which rely on various resources that make up the pipeline. We identified the stakeholders involved in the scenario and derived well defined use cases. We treat the BTS data pipeline as a resource ensemble and take an ensemble based approach to carry out the use cases. Consequently, we derived a suite of functional requirements for our framework design from our use cases. The functional requirements, which involve data harmonization, resource management and slice management requirements, make sure that we constrain our framework design to solve te motivating scenario and avoid an "over-engineered" solution.

We aim to develop the features of our framework to carry out these use cases using the concept of resource slices. The proposed deployment framework (discussed in Chapters 4 and 5) will serve as the tool to fulfill the use cases of the BTS monitoring system. In Chapter6 we conduct a functional evaluation of our framework through its use in these use cases.

CHAPTER 4

Architecture Design

4.1 Overview

This chapter first presents the operating context of our proposed framework based on the functional requirements from Chapter 3. We then explain the different information models that we developed that are used in our framework. Then, we present the design and service architecture of our framework. We go on to describe the service orchestration required by our framework to complete certain operations. Finally we discuss how we use the Service Mesh concept (introduced in Chapter 2) in our framework along with the benefits that it brings.

4.2 Operating Context

Based on the functional requirements for our framework we present an overview of the context in which or proposed framework will operate in Figure 4.1.

The context of the framework is to deploy ensembles of resources and provide configuration and monitoring capabilities. Since resource ensembles are provisioned on-demand, we need to harmonize various types of resources to facilitate runtime solutions that support IoT data delivery and control. The framework is designed to act as a very powerful middleware that interfaces with third party providers at runtime.

Figure 4.1 presents the context of the framework through different layers of the concerns. The user primary concern is to manage resource ensembles. This is achieved by the framework and the set of services it uses to manage the pool of resource providers. The management of the actual resource at the low-level are done purely through resources providers, which are managed by our framework.

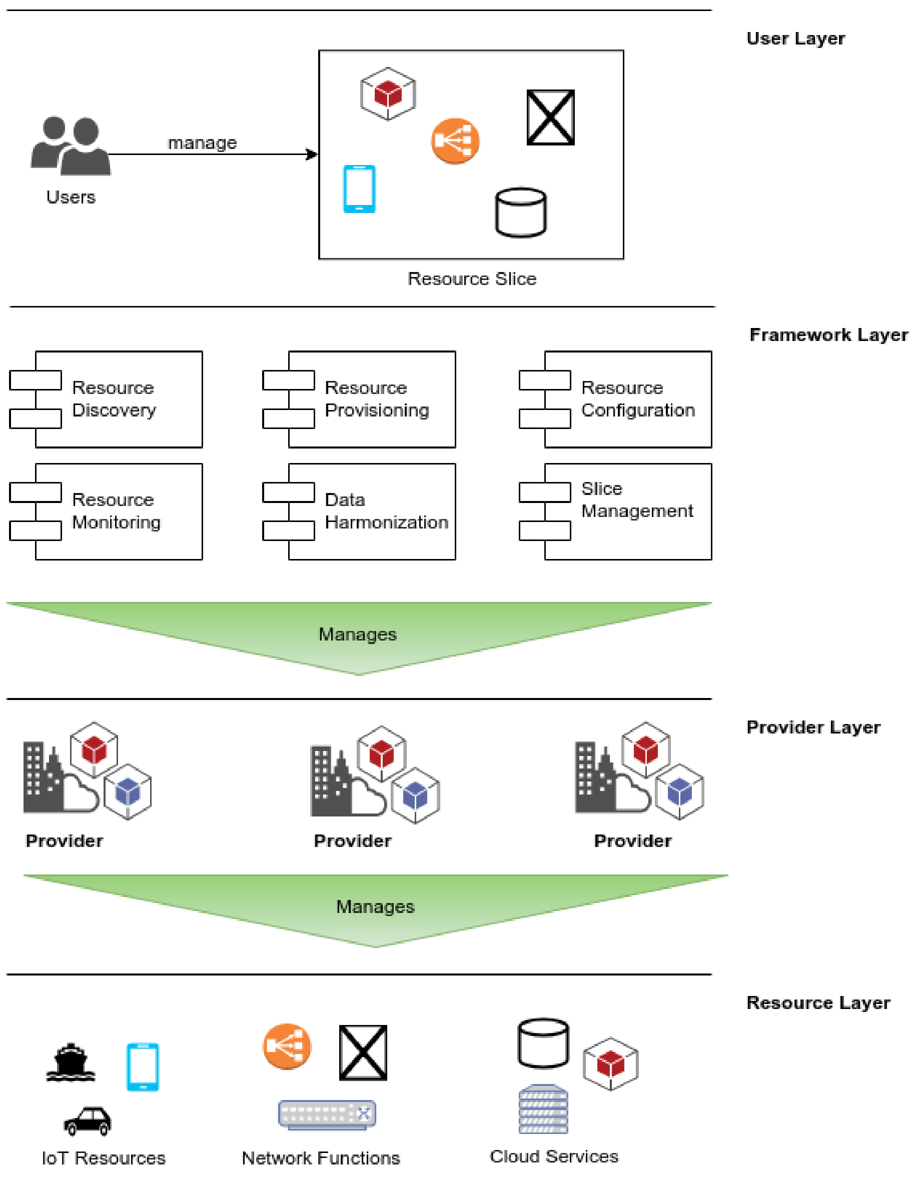


Figure 4.1: An overview of the operating context of our proposed framework

The following sections will discuss the models and architecture of our proposed framework, their implementation details, and the various metadata and information models that we use unify access to different resources and their providers.

4.3 Models

4.3.1 Resources and Resource Providers

Our work deals with ensembles or *slices* [TN16] of resources in an IoT Cloud System. For us a *resource* is a very generic way of describing a component that forms a part of an IoT Cloud System. Using the existing work [TN16] [LNT16], we can see that IoT Cloud Systems can be broken down into three types of resources:

- IoT Resources: These types of resources can be simple or complex. For example, a simple case would be a set of public sensor data that can be accessed through a repository using some kind of client API or an actual IoT device network. Access can be subscribed and controlled through actuators and gateways offered as a service model [PZCG13][Per17].

- Network Function Services: These resources generally deal with a lower level of communication infrastructure. There has been work into virtualized network functions to accelerate and facilitate provisioning of networking components in dynamic virtualized environments. We can already see the use of different types of network function services in cloud IaaS providers (e.g. Google Cloud VPC [SRR18], Amazon Web Services VPC [Ser17]). The types of network functions available from these provider include load balancers, firewalls and intrusion detection devices. Sometimes network functions can be hard to classify. For example the network functions we mentioned previously could also be classified as cloud services due to their location. Additionally, network functions located in IoT devices or gateways could be IoT resources. In general, when given a network function we could also classify it by its location.

- Cloud Services: We can find an abundance of cloud services due to the widespread usage and promotion by large industry entities. Organizations such as Google and Microsoft provide various cloud services such as storage, database as a service or analytics as a service. Many examples of IoT scenarios recently published [TB17] [PZCG13] [Tho15], require interaction between IoT and the cloud.

A resource provider is an entity that provider different types of resources. If we take Google Cloud as an example, we can see that it is as a provider for network function and cloud service resources [SRR18]. Google cloud allows users to create their own virtual private networks with resources such as firewalls rules, load balancers or cluster management solutions which fall into the Network Function Service category. While on for Cloud Services, we can see the BigQuery [Goo18b] and Storage product as well as Google Cloud Pub Sub [Pla17a] which is a messaging as a service product.

4.3.2 Harmonizing Resource Data

The different types of resources will be very distinct regarding their attributes. The differences can either be implicit (e.g. a sensors will never be equivalent to a storage bucket) or explicitly (e.g. MongoDB Atlas [Inc18b] and MLab [mla18] provide databases with different model representations). To make our framework support deployment through resource discovery and querying, the information presented to the user must be unified. This harmonizing of the different resource data and metadata is the only feasible way to facilitate a unified set of operations on these resources.

During the implementation to try to achieve the use cases that we set out and aiming for high automation, we decided to use the document based representation given by [TB17] and used in part by [LNT16]. We decided to orient the documents towards resources so that each resource and its related configurations, parameters and metadata are described in a single document. This made sense from our bottom-up perspective since a deployment requires all of the resource's available metadata and configuration to be successful. Furthermore, by keeping all the resource metadata in the same location (a single source of truth) we obtain by default, a natural service discovery mechanism. We can query different types of resources through a JSON based query like the one used by the most popular document-oriented database, MongoDB.

Figure 4.2 presents the model of our resource description as a UML class diagram. The description include its name, resource provider and unique identification (uuid). We also classify the resource through the *resourceType* field. The *metadata* object is a collection of information about that resource that can vary between resources depending on their type. A few examples of metadata could be some protocol specific interoperability data that details the protocol and data formats information to help resolve interoperability issues. The interoperability metadata is actually used by others in ongoing research, we do not deal with them in the scope of this thesis.

The *parameters* object contains all the configurable elements of the resource. In Listing 4.1 it is the mqtt connection along with the published topics which is what we call an *access point*. Access points reference network communication, since our resources all form a part of some distributed system. We model their communication access points in terms of egress (outgoing) and ingress (incoming) access points. However, the *parameters* object can also contain resource specific parameters. In Listing 4.2 the *parameters* contain specific information about the configuration of the provided BigQuery instance, such as the dataset and table IDs along with the schema definition.

This single document resource description allows us to treat resources similarly to a REST entity. This avoids transmitting state information among different software implementations that make use of our resource description and facilitates interoperability. Furthermore, by following the RESTful convention we can define a simple transactional mechanism to create, delete and update our resources, we could even go further and propose other actions that act on our resources. One action that could be useful is to fetch any related monitoring information. In our representation, we may apply the same

```
{
  "uuid":"sensor1528671664454"
  "name":"sensor humidity",
  "providerUuid":"sensorlocal1",
  "resourceType":"IOT_RESOURCE",
  "location":null,
  "parameters":{
    "ingressAccessPoints":[

    ],
    "egressAccessPoints":[
      {
        "applicationProtocol":"MQTT",
        "host":"linkerd-testslice-sensor",
        "port":7474,
        "accessPattern":"PUBSUB",
        "networkProtocol":"IP",
        "qos":0,
        "topics":[
          "topic"
        ]
      }
    ]
  },
  "metadata":{

  },
  "dataPoints":[
    {
      "name":"humidity",
      "dataType":null,
      "unit":"percent"
    }
  ],
}
```

Listing 4.1: Example of a sensor resource

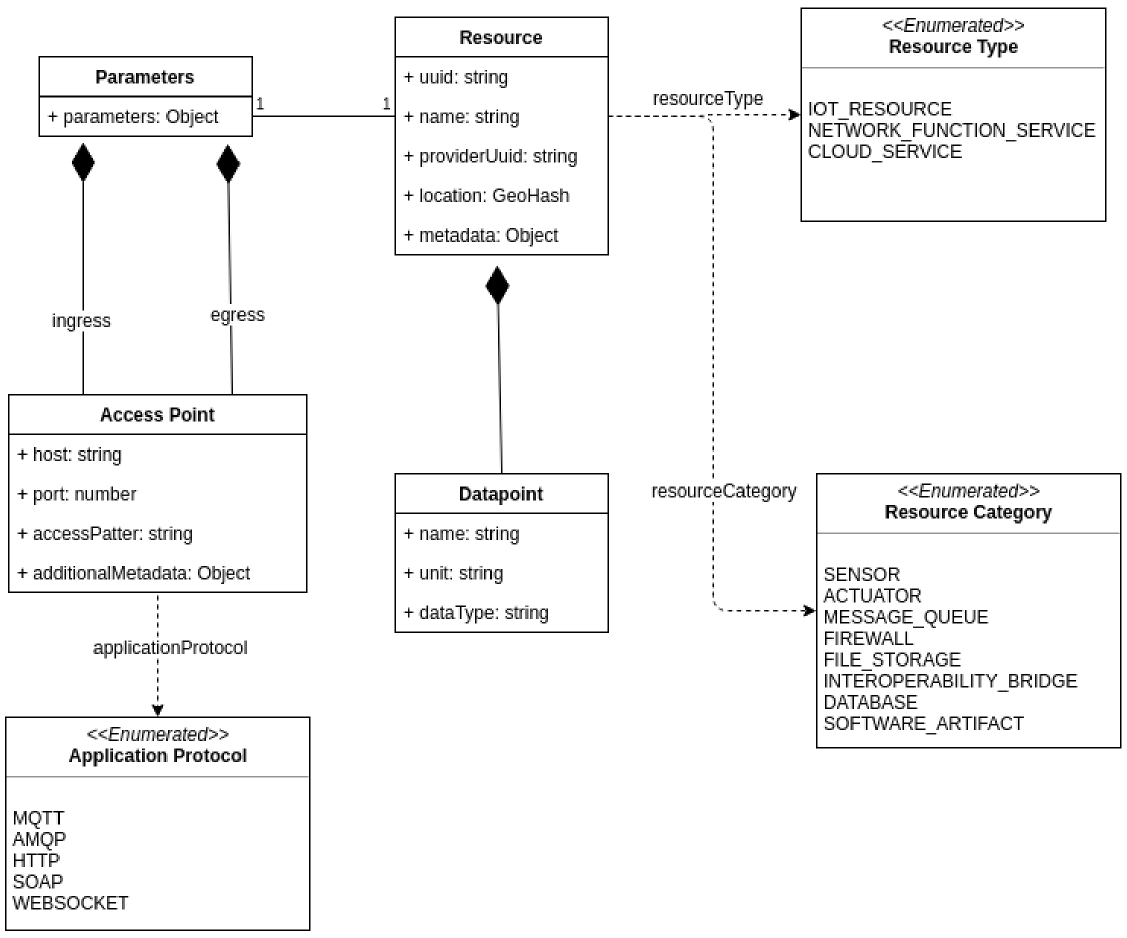


Figure 4.2: The UML class diagram of our resource object

configuration procedure to each resource. We assume that a user who creates or updates a resource understands the semantics of the resource's parameters. This can be done through documentation from the resource providers. With the widespread of different software documentation tools such as Swagger [Sma17], this would not deviate from standard industry and development practices.

4.3.3 Specifying a Resource Ensemble/Slice

In the previous section we discussed our resource representation, however that is only a part of our modeling requirement. With our semi-structured representation of resources, we address the specification an entire system in the form of a resource ensemble.

In order to facilitate the automated deployment of a resource ensemble, we must specify the resources and their relationships with respect to each other. To this end, we introduce another entity that we call a *Connectivity*. A connectivity is an entity that describes the

```
{
  "name":"bigQuery dataset",
  "pluginName":"bigquery",
  "providerUuid":"bigquerylocal1",
  "resourceType":"CLOUD_SERVICE",
  "location":null,
  "parameters":{
    "ingressAccessPoints":[

    ],
    "egressAccessPoints":[

    ],
    "datasetId":"testDataset",
    "tables":[
      {
        "id":"testTable",
        "schema":[
          {
            "description":"field description",
            "mode":"REQUIRED",
            "name":"id",
            "type":"STRING"
          },
          {
            "description":"field description",
            "mode":"REQUIRED",
            "name":"value",
            "type":"FLOAT64"
          },
        ]
      }
    ]
  }
}
```

Listing 4.2: Example of a Google BigQuery resource

relationship between two resources. Figure 4.3 shows the addition that we make to our resource model to accommodate inter-resource relationships. We can see an example of a connectivity in Listing 4.3 towards the bottom. This connectivity states that the two resources communicated data with each other through MQTT using a data JSON data format.

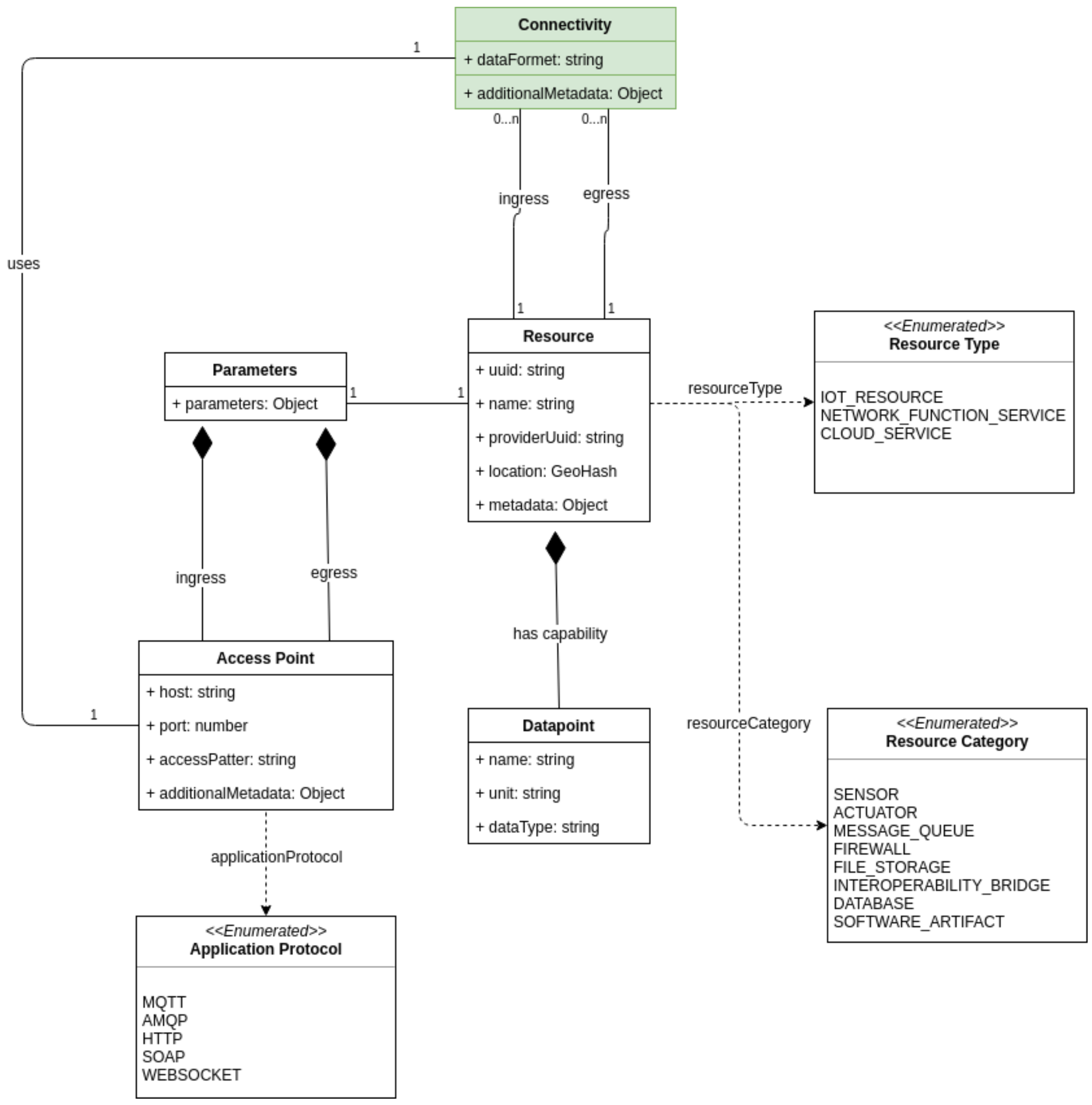


Figure 4.3: Modified Resource UML class diagram to accommodate inter-resource relationships

The extra connectivity object that we add to our resource model allows us to, for one resource, keep track of the relationships with other resources.

We express our models through the form of a JSON document. Listing 4.3 shows an example of a simple slice description involving two resources: an MQTT broker and a sensor that publishes to the broker.

```
{
    "sliceId": "testslice",
    "resources": {
        "sensor": {
            "name": "sensor humidity",
            "providerUuid": "sensorlocal1",
            "resourceType": "IOT_RESOURCE",
            "resourceCategory": SENSOR,
            "location": null,
            "parameters": {
                "ingressAccessPoints": [],
                "egressAccessPoints": [
                    {
                        "applicationProtocol": "MQTT",
                        "host": "linkerd-testslice-sensor",
                        "port": 7474,
                        "accessPattern": "PUBSUB",
                        "networkProtocol": "IP",
                        "qos": 0,
                        "topics": [
                            "topic"
                        ]
                    }
                ]
            },
            "metadata": {},
            "controlPoints": [],
            "dataPoints": [
                {
                    "name": "humidity",
                    "dataType": null,
                    "unit": "percent"
                }
            ],
            "source": [],
            "target": [
                "mqtt_connectivity"
            ],
            "adaptorName": null,
            "resourceCategory": null
        },
        "broker": {
            "name": "mosquitto broker",
            "pluginName": "mosquittobroker",
            "providerUuid": "mqttlocal1",
            "resourceType": "NETWORK_FUNCTION_SERVICE",
            "location": null,
            "parameters": {
                "ingressAccessPoints": [
                ],
                "egressAccessPoints": []
            },
            "metadata": {},
            "controlPoints": [],
            "dataPoints": [],
            "source": [
                "mqtt_connectivity"
            ],
            "target": [],
            "adaptorName": null,
            "resourceCategory": null
        }
    },
    "connectivities": {
        "mqtt_connectivity": {
            "accessPoint": {
                "applicationProtocol": "MQTT",
                "networkProtocol": "IP",
                "qos": 0,
                "topics": []
            },
            "dataFormat": "JSON",
            "ingress": {
                "label": "sensor",
                "accessPoint": 0
            },
            "egress": {
                "label": "broker",
                "accessPoint": 0
            }
        }
    },
    "createdAt": 1528671706
}
```

Listing 4.3: Example of simple resource slice

We express the specification or a resource ensemble in the form of a JSON document. The document represents a graph of nodes(resources) connected by edges(connectivities). The semi-structured nature of this document permits us a large degree of machine readability for automated operations. Any program could take this JSON representation and parse it to create the resulting graph of the resource ensemble with respect to its data stream. Therefore we can have a high degree of interoperability with our slice description.

We could use the description to provision our slice,and then submit it to other services to check for either interoperability issues, optimization or even detect faults with the slice. These features all represent current work associated with this thesis. Furthermore the JSON slice description results in a single source of truth for the resource slice. This allows us perform operations such as those we mentioned with resources in the previous section.

4.4 Detailed Design and Service Architecture

Figure 4.4 presents an overview of the entire framework architecture. The concrete design details are discussed later in this section and the prototype implementation is explained in the next chapter. The overview of the framework shows the layout of the services which interact with each other to fulfill the functional requirements laid out in the previous section. As Figure 4.4 illustrates, there are four software modules that comprise the framework:

- **Global Management Service**
- **Local Management Service**
- **Adaptor**
- **Slice Management Client**

Along with two stakeholders: **Resource Providers** and **Ensemble Users**.

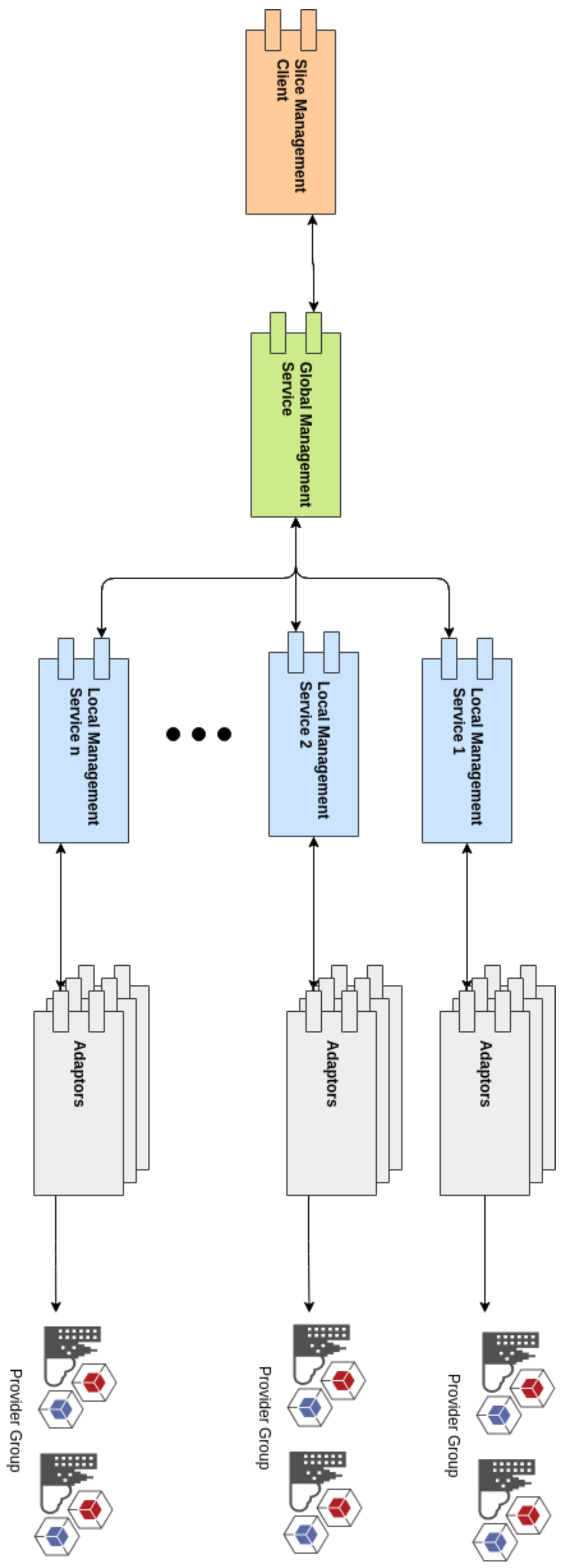

Slice Management Client
Global Management Service
Local Management Service 1
Local Management Service 2
Local Management Service n
Adaptors
Adaptors
Adaptors
Provider Group
Provider Group
Provider Group

Figure 4.4: An architectural overview of our framework

According to Figure 4.4, a Provider Group interacts with a Local Management Service to exchange provider data and controls. The Local Management Service is responsible for populating the framework with data obtained from providers and their resources. Additionally, the Local Management Service is also the point where controls are forwarded to the providers. A Local Management Service interfaces with its assigned providers through adaptors.

The responsibility of the adaptor is to transform and harmonize the data sent to and from the provider due to the heterogeneous nature of resources and providers. Local management services can be deployed in the cloud or edge (an IoT gateway or a network station) and interfaces with IoT Resource Providers to manage the resources in an organization (e.g., a company with multiple resource providers) or any kind of logical grouping of resource providers. Thanks to the adaptors that are deployed to interface with the different providers, we can have unified access to all the resources of such an organization.

The Global Management Service is responsible for serving clients which require the services of the various resource providers. Resources can be queried and controlled with the various Local Management Services. A Global Management Service can be used with multiple Local Management Services. Resource queries can be effected for a single Local Management Service or a grouped subset of Local Management Services. For example, we could run a resource query for resources served by local management services that interface with a specific organization such as Microsoft Azure. While the local management services are intended to be an intermediate step in the process to unify data going to and from the providers, the Global Management Service exposes to an end user to providers and their resources.

The Slice Management Client, uses exposed by the Global Management Service such as the query, provisioning, configuring and monitoring of resources. The Slice Management Client is a higher level module that uses the base resource level operations to manage resource ensembles. Additionally, the Slice Management Client is a point of contact with an interoperability recommendation service (based on ongoing work) which can determine interoperability issues within the ensemble and can recommend possible software artifacts that act as bridges between resources. The Slice Management Client is one way in which a user may create and manage a slice. It is also possible to directly use the resource API provided by the Global Management Service. However, this results in more overhead as the user would be responsible for managing the inter-resource relationships of a resource ensemble.

4.5 Additional Models

This section quickly references some additional models that are used in our framework. These models are not as conceptually important as the data harmonization models that we previously presented, but are important to the framework's function. The importance of these models is made clear in the next chapter that details the prototype implementation.

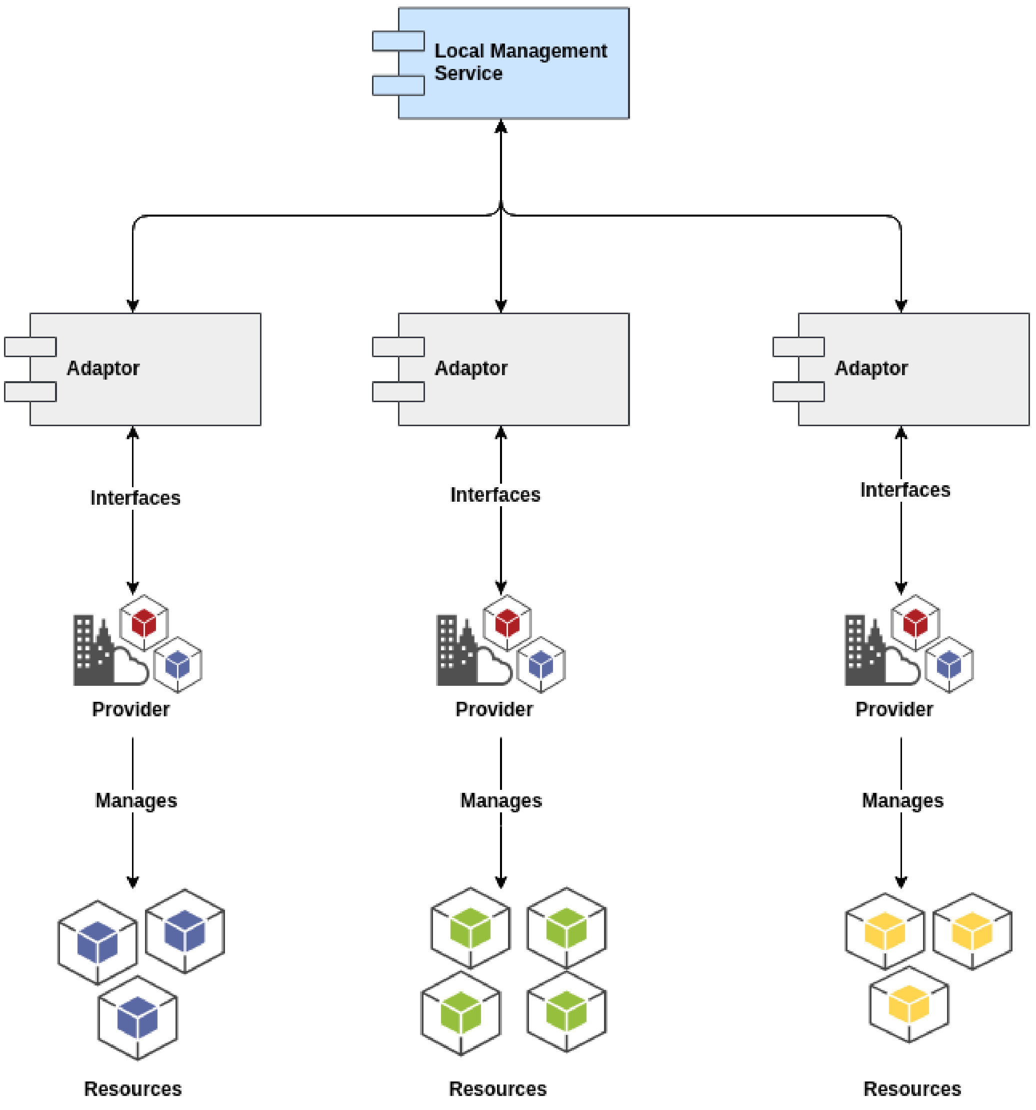


Figure 4.5: Architecture of a Local Management Service interfacing Providers

4.5.1 Control Result

This model contains the result of any controls that will be executed on the provider or its resources. It encapsulates the final state, output, time and the unique identifier of the control.

4.5.2 Adaptor

As seen in the previous chapter, the adaptor is responsible for interfacing with the providers. Since the adaptor is the only interface with the provider with our framework.

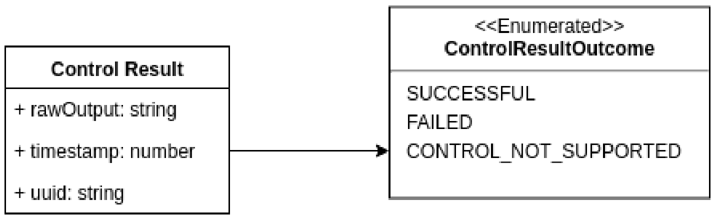


Figure 4.6: UML class diagram of a Control Result

We can assume that the adaptor is associated to the resources of the provider. There is also different metadata can be included in this model for ongoing interoperability work related to the work in this thesis.

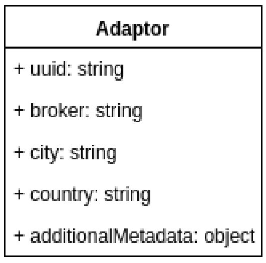


Figure 4.7: UML class diagram of a Provider Adaptor

4.5.3 Local Management Service

The model of the Local Management Service owns various metadata that will be used for ongoing interoperability work . The elements *uuid* and *groupId* will be used for inter-service orchestration and communication.

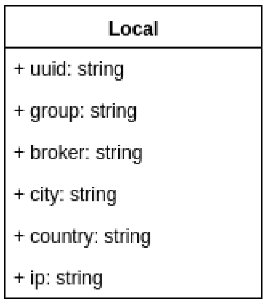


Figure 4.8: UML class diagram of a Local Management Service

4.6 Service Orchestration

Our framework consists of four different types of components that communicate with each other to function correctly. Once a user launches a query through a Global Management Service, the Global Management Service must send the query to the appropriate Local Management Service, which then acts on the provider through the appropriate adaptor. As we mentioned in the previous section, there can be many Local Management Services for a Global Management Service and many adaptors for a Local Management Service. Additionally, the Slice Management Client and external stakeholders communicate with the Global Management Service for use of provider services. In order that these services can all communicate with each other, they need to be orchestrated. In this section, the orchestration of these services is described step by step.

4.6.1 Initial Registration

The initial registration concerns the registration of our distributed components with each other. This registration occurs on two levels: Firstly, Local Management Services will register to the Global Management Service. Secondly, the provider adaptors register to their respective Local Management Services. The registration of the Local Management Services lets the Global Management Service know where to find the *menu* of resources. The registration of the provider adaptors is equivalent to the registration of the resource providers into our framework which provide the *content* to the *menu*.

1. Once the Global Management Service is up and running, it constantly listens to registration messages from new Local Management Services.

2. A Local Management Service sends a registration request to a Global Management Service. The Global Management Service keeps a list of all available Local Management Services that are registeredby their *uuid* and *groupId*.

3. Once a Local Management Service is up and running, it listens to registration messages from new resource providers adaptors.

4. An adaptor sends a registration request to a Local Management Service. The Local Management Service keeps a list of all available provider adaptors.

Figure 4.9 illustrates the described steps for the Initial Registration.

4.6.2 Query

The query process is our solution to resource discovery. The Initial Registration steps (previous subsection) established a registry of resources and means to populate it. In the next steps, we explain the orchestration process to query our resource registry.

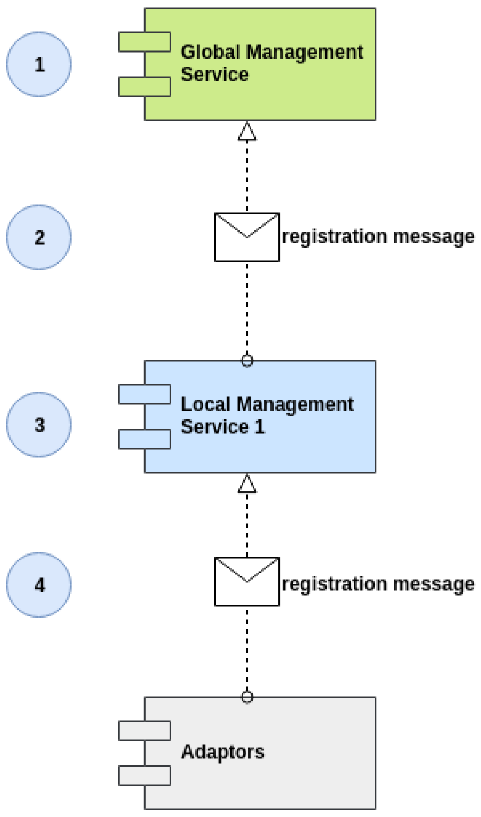


Figure 4.9: Orchestration of the Initial registration of Local Management Services and subsequently their Provider Adaptors

1. A Local Management Service keeps a list of all available adaptors that are registered. This list is queried periodically for resource and provider data.

2. Once an adaptor receives a query, it retrieves data from the provider API and harmonizes this data with our information model. Once the adaptor has harmonized the resource and provider data, it is sent to the Local Management Service where it is cached to provide a fast response to queries from the Global Management Service management service.

3. The Slice Management Client or an external stakeholders queries resources or providers by submitting the query to the Global Management Service.

4. The Global Management Service broadcasts this query to the subset of Local Management Services that are concerned by the query.

5. The Local Management Services concerned by the queries, will send their cached resource and provider data back to the Global Management Service.

6. The Global Management Service waits for the results of the query from the different Local Management Service services and sends the result back to the Slice Management Client or the external stakeholders.

Figure 4.10 illustrates the query orchestration steps.

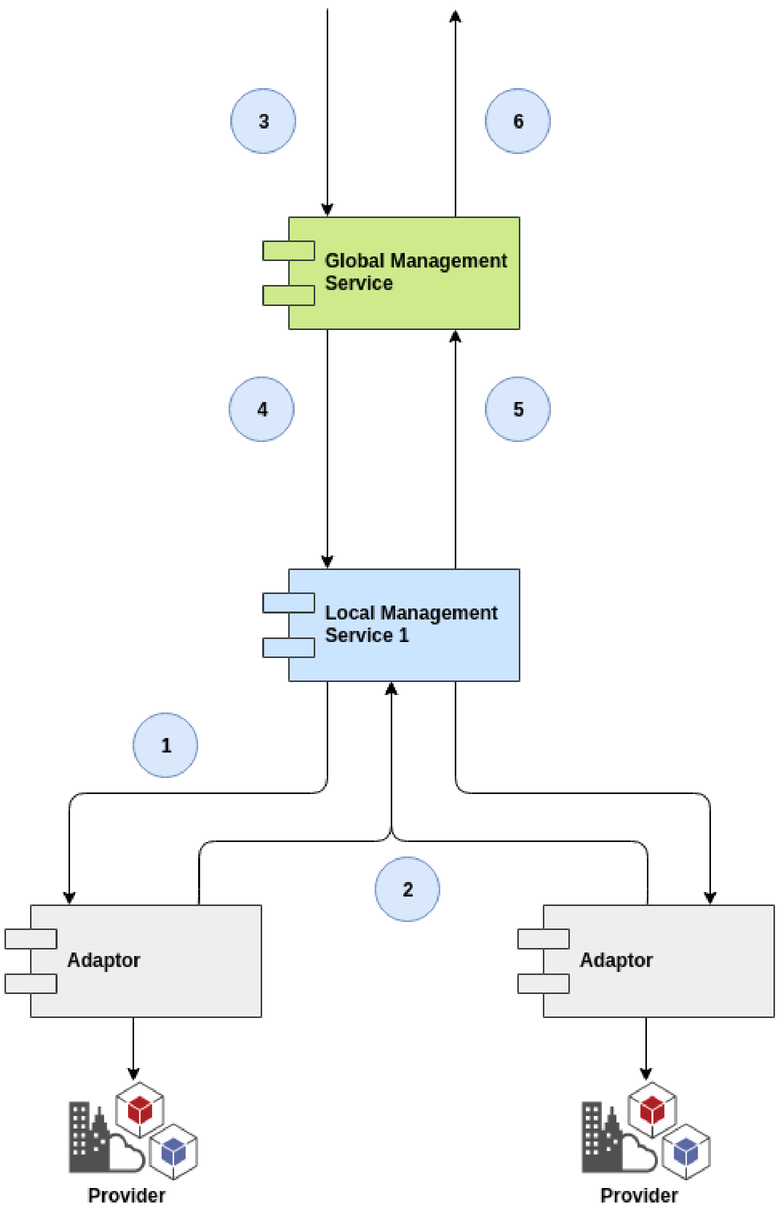


Figure 4.10: Service orchestration for resource provider query

4.6.3 Control

We interact with resources and resource providers through controls. A control can be the provision, configuration or the removal or a resource. We document here the orchestration between our components in order to send controls and and receive the execution results of those controls.

1. The Slice Management Client or an external stakeholder sends a control for a particular resource to the Global Management Service.
2. The Global Management Service sends this control to the Local Management Service responsible for the right provider.
3. The Local Management Service receives a control and sends the control to the correct provider adaptor.
4. The adaptor receives the control and extracts the parameters needed to execute the control through the resource provider's API.
5. The result of the execution is retrieved and harmonized with our information model which is then sent back to the Local Management Service.
6. The Local Management Service receives the control execution result and extracts the resource data from the control result and sends this data to the Global Management Service.
7. The Global Management Service waits for the resulting resource data and sends the result back to the Slice Management Client or the external stakeholder.

Figure 4.11 illustrates the control orchestration steps

4.7 Service Mesh

In this section explain how we leverage the Service Mesh paradigm to achieve the requirements for resource discovery and dynamic runtime configuration of resources. We also indicate the problems faced by our framework in these areas without the service mesh, to highlight its advantages.

Figure 4.12 presents a diagram of a basic resource topology using a service mesh. This is the solution that we adopt in the following chapter. In this solution, each resource is deployed alongside a proxy which determines the correct destination through a service discovery lookup. Notice that in Figure 4.12 no resource communicates directly with the another. However the communication occurs between the different resources through the proxies thanks to the address resolution feature provided by the service discovery lookup. This is only one possible architecture for a service mesh. Some other solution architectures are provided by [Jen18]

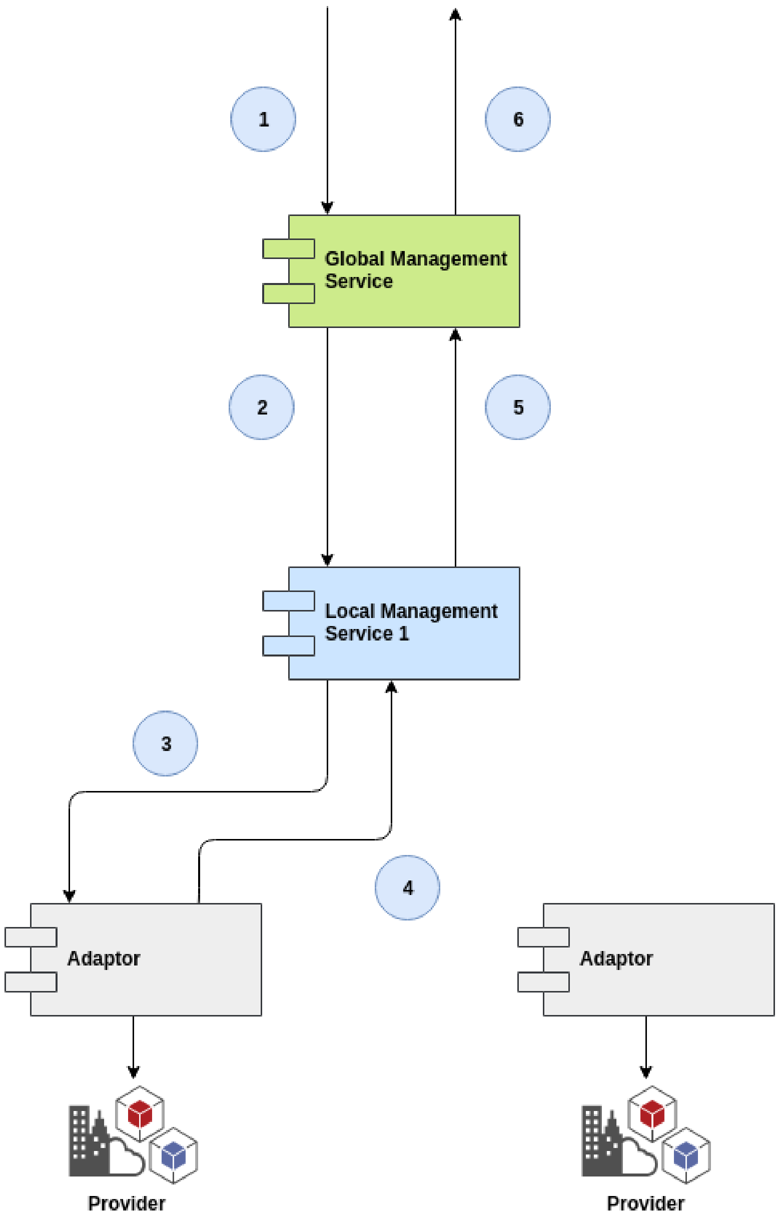


Figure 4.11: Service orchestration for resource control

One of the challenges posed by our work is that we do not own or control the majority of resources. Our control of different resources is limited to the exposed APIs, whose definition may be more fine or coarse grained depending on the provider. In the following subsections we discuss several important aspects of the Service Mesh that can address constraints posed by provider APIs.

We take the following example: A resource communicates with another through REST,

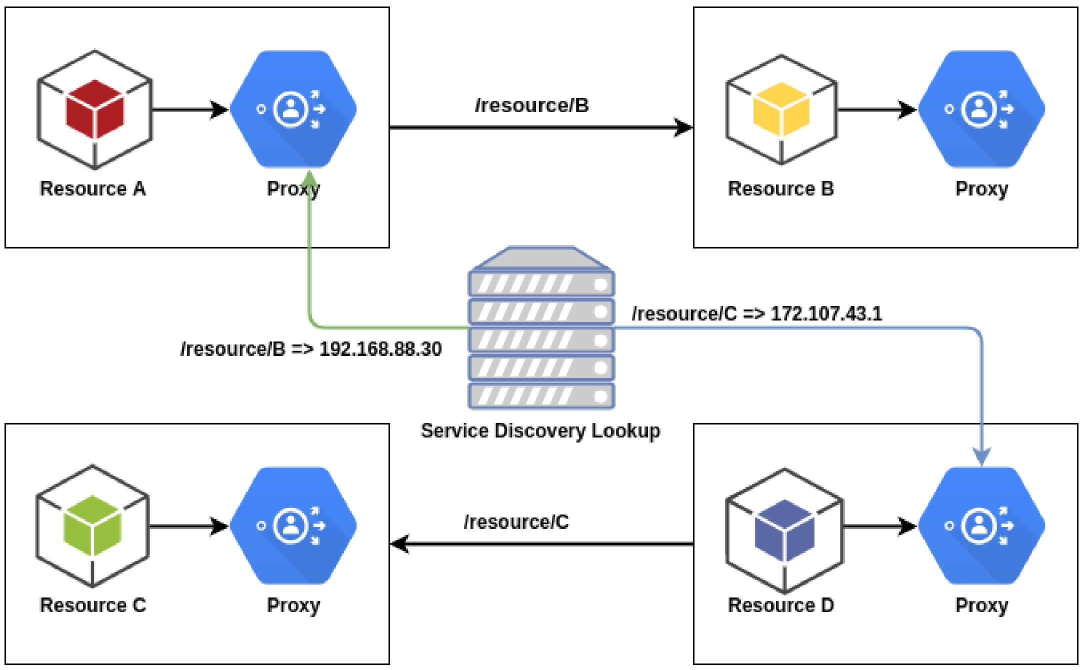


Figure 4.12: A basic Service Mesh, resources A and D only know the location of resources B and C through the lookup

we call these resources the client and server.

4.7.1 Resource Discovery

Our proposed framework can provision the two resources without difficulty, we assume that they are cloud services. However at the moment of provisioning the client does not know anything about the server. Therefore, at the end of this provisioning phase, communication between the two resources is inherently broken as the client has no idea of the server's URI to which it is publishing data.

During the configuration phase, in order to obtain a functioning set of resources (i.e. client publishing data to the server), we must reconfigure the client to inform it of the server's URI. A reconfiguration of the resource may not be possible due to limitations of the provider's exposed API. This situation presents us three very important problems:

1. In order to reconfigure a resource we must actually provision another one. This re-provisioning is hidden as a configuration step in a chain [Ins13]. A provider might provide the means to reconfigure its resources, however there is no guarantee that we can reconfigure this URI parameter as we do not own the provider or its resources.

2. Any changes to a resource ensemble would result in an more of work, due to the point above. If reconfiguration resulted in re-provisioning, we would exhibit stagnant performance for our "on-demand" framework.
3. How would resource providers react to such a wasteful use of their resources? Our framework could potentially lose trust or be blacklisted by providers for misuse of their platforms

These issues, can be seen very often in industry and is informally known as "dependency hell" regarding microservices (not software). Work has been done in the field of microservices to address this problem [GGG$^+$16] [DCLT$^+$14]. However, this work does not exactly fit in with our domain since we also aim to provision and configure resources from third parties (even though in some cases we would deploy artifacts ourselves in the form of VMs or containers).

The Service Mesh concept helps us provide a measure of of isolation between each resource and allows us to coordinate the communication topology without relying too heavily on the limitations of the resource provider. In our example provided above combined with the Service Mesh, the client would publish data to a reliable source immediately after its provisioning and configuration. Although some data might be lost during the deployment of the server, we no longer need to reconfigure the client once we the server has been provisioned.

4.7.2 Runtime Functionality

We aim to provide as much of our functionality as possible dynamically during runtime. Therefore, being able to reconfigure the communication topology of a resource ensemble in function of the user requirements of his/her system is essential. As mentioned previously, in a Service Mesh each resource is only aware of its outgoing communication through its attached proxy. In the event that resources need to be routed to different access points, or new resources need to be deployed into the ensemble, a simple reconfiguration of each proxy will ensure that inter resource communication is operating correctly without any downtime releases.

4.7.3 Usage in our framework

The Service Mesh aims to solve a very important problem in our framework. The resources that we aim to "connect" in resource ensembles need an element of network connectivity. Since the Service Mesh operates at the TCP/IP level, we use it in our framework to abstract the need to configure each resource to correctly communicate with other resources. By deploying the service mesh we only configure any TCP/IP protocols once. Any further configurations (at runtime) can be done by simply reconfiguring the service mesh.

When we provision a resource, we also provision a configurable sidecar proxy that can be controlled by our Slice Management Client as seen in Figure 4.12. The Slice

Management Client in our design is what is know as the control plane to the service mesh in [Jen18][FS18][Kle17]. Since the Slice Management Client is responsible for the service mesh of a resource slice, it is responsible for the routing of TCP/IP communication in a resource slice. The result is that the configuration regarding TCP/IP communication has been abstracted from the resource provider. When a resource is provisioned, it is configured with the TCP/IP parameters (IP/port) of the relevant sidecar proxy. Therefore, in situations where only a change in the communications topology occurs in a resource slice only the Slice Management Client is involved in the reconfiguration.

However, this solution does not mean that our framework cannot function without a Service Mesh. We explicitly state that we only adopt this architectural concept to simplify the configuration of a resource slice, which in our framework design is handled by the Slice Management Client. The relevant resource level operations can still be accessed through the Global Management Service by any other third party client. Thus, our design is flexible enough to benefit from this paradigm but does not obligate users to use it.

4.8 Summary

In this chapter we firstly presented the different models that we use to harmonize resource data in our framework. We also briefly present some other models that are important to the framework's function. We then explained the design of a novel architecture that is capable of interfacing with various resource providers to query and control resources through harmonized information model. First we presented the context in which the framework will be operating in. Based on the operating context, we present the detailed design and service architecture of the different software services that we need to implement without going into the technical implementation details. Then, we give a high level view of the service orchestration needed to completed different tasks that are required by the framework. Lastly, we present the concept of the Service Mesh, that is very important in the tackling the problem of inter-resource communication.

CHAPTER 5

Prototype Implementation

5.1 Overview

Based on the presented architecture in Chapter 4, we present in this chapter a prototype implementation of our proposed deployment framework.

Figure 5.1 presents an overview of the prototype implementation of our framework. We chose to separate the different modules of our framework into separate standalone software services with defined communication protocols. We discuss each of the modules in the sections below.

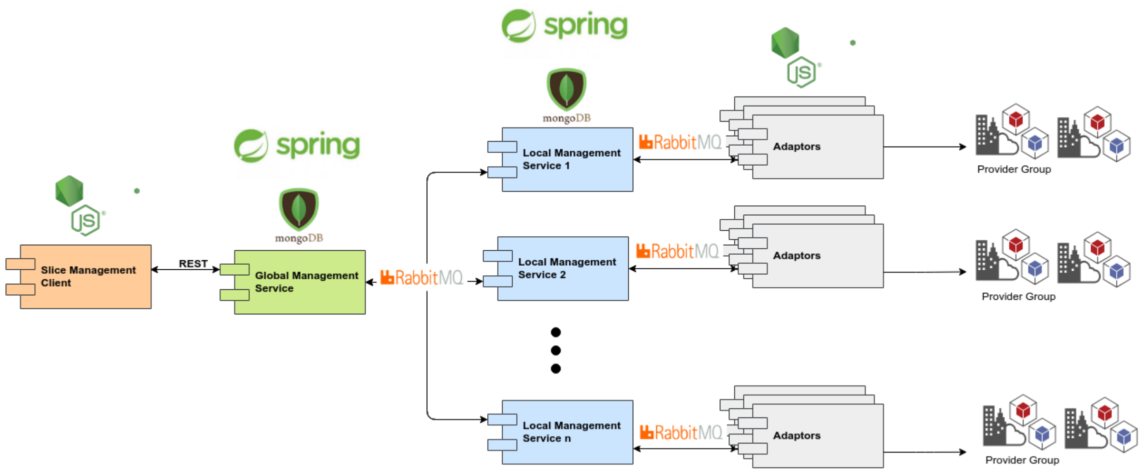


Figure 5.1: Overview of the technologies used in the prototype implementation

We chose to use AMQP through RabbitMQ [Piv18] as the primary communication medium for the "internal" part of the framework, in effect the part of the framework behind the Global Management Service. The initial AMQP communication was inherited as a part of the HINC project [LNT16] that we studied before starting work on the work of this thesis.

5.2 Implementation

5.2.1 Provider Adaptors

In our architecture presented in Chapter 4, Resource Providers are external services who expose their resources via sets of APIs. Due to the heterogeneity of the resource providers, we do not interface directly with particular providers but through an adaptor that exposes a unified API. In our prototype implementation, the adaptors are standalone software artifacts that communicate with a Local Management Service.

The implementation of an adaptor should be unique to each provider. However, we define a clear communication protocol between a Local Management Service and its adaptors to ensure that the two components can function together with expected behavior. The protocol is based on request response. Table 5.2 in Section 5.3 presents an overview of the communication protocol.

As described in the protocol, an adaptor first registers its presence to the Local Management Service to which it will be assigned. Upon registration by the Local Management Service, all the appropriate queries and controls will be received by the adaptor. Once an adaptor has been registered, it receives messages from the Local Management Service that could correspond either to queries about the provider and its resources, or controls to execute operations on the provider's resources. The exact messaging protocol and how the message routing is done is explained in detail in Section 5.3.

5.2.2 Local Management Service

The Local Management Service manages a set of resource providers through the adaptors that are deployed. The Local Management Service listens to adaptor registration messages and keeps a list of available adaptors that it can query and control. The Local Management Service in our prototype is implemented using Java. We use Spring Boot [Piv17] which provides use with lots of helper tools such as dependency injection and a high level and declarative abstraction to manage AMQP resources. We simply use Spring AMQP [Piv17] to declare all the exchanges and queue bindings, these bindings are illustrated in Figure 5.3.

In order to store our document based data, and to provide powerful querying capabilities, we opted for a document based database. We chose MongoDB, the most popular JSON document based database on the market. By using Spring Boot [Piv17], we have all the built in database access classes that can be dependency injected into the application. MongoDB [Inc17] also provides us with the ability scale our framework easily and access the database from many different Local Management Services sharing a database instance.

Service	Payload	Response	Description
CREATE RESOURCE	Resource	Resource	Provisions a specified resource
MODIFY RESOURCE	Resource	Resource	Configures a specified resource
DELETE RESOURCE	Resource	Resource	Deletes a specified resource
MONITOR RESOURCE	Resource	Resource	Fetches available monitoring information on specified Resource
QUERY RESOURCE	Resource	List<Resource>	Returns all running resources that matches provided query
QUERY RESOURCE	Resource	List<Resource>	Returns all resource available to be provisioned that matches query

Table 5.1: An overview of the services provided by the Global Management Service's API

Additionally, we also opted to build a set of DAOs for our information model ourselves, due to technical difficulties in using the built-in Spring DAO to serialize schemaless JSON nodes.

Regarding adaptors, we store all registered adaptors in memory using a hashmap for faster routing. We also create a special *AdaptorManager* class in the Local Management Service to expose all adaptor operations. We use the singleton pattern for the implementation of the *AdaptorManager* along with thread safe datastructures in order to guarantee thread safety, since the asynchronous message receiving system in Spring Boot is implemented using multi-threading.

5.2.3 Global Management Service

The Global Management Service is implemented with Java using Spring Boot[Piv17]. Like the Local Management Service, Spring Boot provides us with many useful tools for AMQP, which the Global Management Service uses to communicate with its Local Management Services.

The resource provider services that are exposed to the Slice Management Client and other external stakeholders is done so using a RESTful API at the Global Management Service. Spring Boot also provides a built in declarative RESTful API builder. Swagger is used to provide an interactive API documentation for the services that we expose. The services we expose are summarized in Table 5.1.

The first services: CREATE, MODIFY, DELETE AND MONITOR (in Table 5.1) are our resource controls, we treat a resource as a RESTful entity. Every resource control expects a specification of a resource according to our information model. Any updates to the resource as a result of the control are returned upon the completion of the control.

This is necessary because some metadata or parameters of a resource are only available after the control. For example if the resource were to be a simple HTTP server, the IP address of the server would only be known once the server has been provisioned.

Figure 5.2 describes the sequence of actions used in the resource controls of create, configure and delete. The flow is relatively simple, the request is passed down from the Global Management Service to the provider adaptor which executes the "low-level" operations on the resource provider. The adaptor then harmonizes this data to enter our framework again. In the case of these resource controls, the Global and Local Management Services only deal with our resource model illustrated in Figure 4.2.

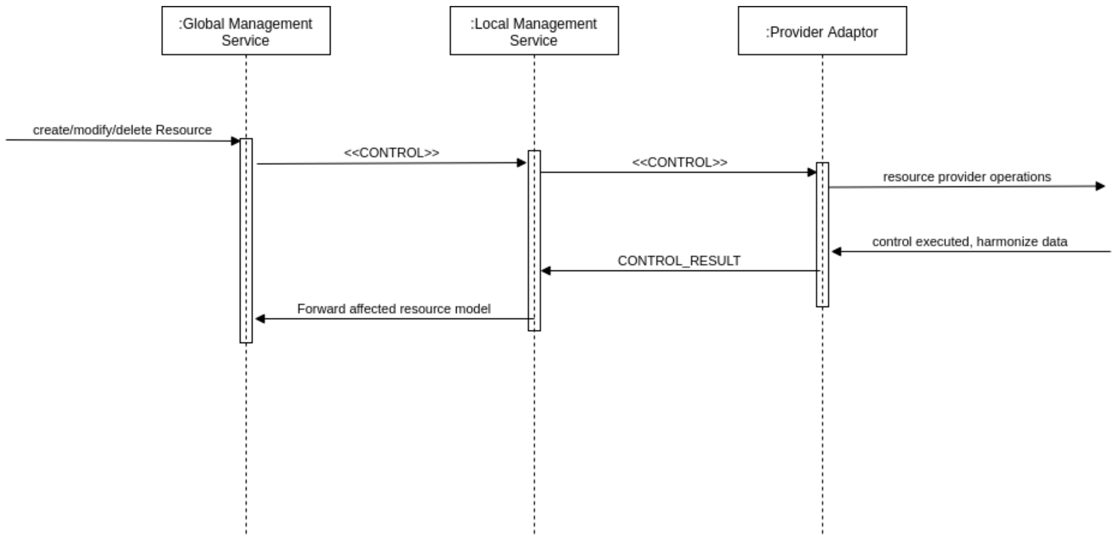

:Global Management Service
:Local Management Service
:Provider Adaptor
create/modify/delete Resource
<<CONTROL>>
<<CONTROL>>
resource provider operations
control executed, harmonize data
CONTROL_RESULT
Forward affected resource model

Figure 5.2: Sequence diagram describing a resource control

All the queries and controls of our framework follow the same principle from the Global Management Service. A request is initiated from the Global Management Service and relayed and routed through Local Management Services and provider adaptors and sent back to the Global Management Service. The differences between requests concern mainly the payload, which we document in Section 5.3.

Although the Global Management Service exposes resource related services, it does not manage resource relationships. A resource is treated like a REST entity by the Global Management Service. A resource can be created, configured and deleted. It is possible to manage multiple resources as a resource ensemble through the Global Management Service's API, if the responsibility of the maintaining the relationships among the resources are handled by the client using the Global Management Service. The REST API means to provide low level access to resource operations. We subsequently gain advantage of interoperability by allowing external parties to use our unified resource API.

In order to deal with the deployment of end to end resource slices, we use another service described in the next section to manage the inter-resource relationships. However, thanks to the interoperability of the Global Management Service's REST API, any external stakeholders or organization can create resource slices in a similar way. This also leaves the open possibility of proposing better ways of managing resource slices.

5.2.4 Slice Management Client

The Slice Management Client is the module in our framework that is responsible for managing resource slices. It does this by maintaining resource relationships. We implemented the Slice Management Client as a command line interface (CLI) using Node.js with the help of the *yargs* [Git18b] package. Using these tools we created a POSIX-like documented CLI with positional arguments and nested commands like the popular *git* CLI. Due to the usage of Slice Management Client as a manager for resource *slices* we also refer to it as *pizza.js* or *Pizza*.

Pizza supports resource querying, it is possible to query a provider's available resources. The purpose of this resource querying is important for slice deployment since a user must first identify what resources are necessary for his/her slice. Since our information models are implemented using JSON, the query process resembles the traditional MongoDB [Inc17] style document query.

Creating or modifying a slice is done through the command line using a JSON version of our slice model. In the case of creating a new slice through the *create* command, *Pizza* first deploys a Service Mesh through Google Cloud [SRR18] by deploying a Linkerd [Buo18] sidecar proxy for each resource. Then, *Pizza* interfaces with the Global Management Service to correctly provision and configure all of the slice resources. Once all resources have been correctly configured, *Pizza* routes the Service Mesh proxies using the deployed resources.

When a resource slice is modified, a JSON description of the new resource slice should be submitted to *Pizza* through the *update* command. *Pizza* parses the updated slice description and compares it with the current slice stored in *Pizza*. New resources are provisioned and configured and modified resources are reconfigured through the Global Management Service. Finally, new sidecar proxies of the Service Mesh are deployed and the service mesh is reconfigured according to the updated slice.

While currently the command line Slice Management Client can be seen as cumbersome to use due to the very verbose JSON resource and resource slice descriptions, all resource related operations are still executed through the APIs exposed by the Global Management Service. The main function of the Slice Management Client is to handle the inter-resource relationships that make up the slice as well as the Service Mesh that facilitates resource communication. This design allows future work to accommodate a more user-friendly Slice Management Client.

5.3 Messaging Communication

In our prototype implementation, we use AMQP with RabbitMQ [Piv18] for the purpose of inter-service communication between the Global Management Service, Local Management Service and Provider Adaptors. In this section, we discuss how we implemented our messaging protocol.

5.3.1 Local Management Service and Provider Adaptors

Figure 5.3 presents an overview of the messaging topology between the Local Management Service and its Provider Adaptors, we use the BTS sensor provider as an example. The Local Management Service uses only one queue to receive messages from the various provider adaptors that it handles. However, a Local Management Service can use multiple consumers to process the messages faster (this is done with multi-threading thanks to Spring AMQP). Since only one queue is used to receive all adaptor messages, it is bound to a simple fanout exchange which sends messages to all queues bound to it.

When a new adaptor is registered, its input queue is bound to two different exchanges by the Local Management Service. We assume that the adaptor's input message queue is declared by the adaptor itself. The Local Management Service first binds the adaptor's input queue to the broadcast exchange, which sends messages to all adaptors. The broadcast exchange is a simple fanout exchange that requires no routing key since the messages are sent to every bound queue. The second exchange is the adaptor unicast exchange, which sends messages that are routed to specific adaptor input queues. The adaptor input queue is bound to a routing key which is the adaptor ID that is sent as a part of the adaptor registration message.

Our messaging protocol involves pairs of request and response messages. A message that is sent from the Local Management Service to an adaptor and vice versa will expect a message with a response. However, the response may be empty or not used once it is received. Table 5.2 gives an overview of the different types of messages that are used in the communication between the Local Management Service and its provider adaptors. Tables 5.3 to 5.9 describe the specific protocols.

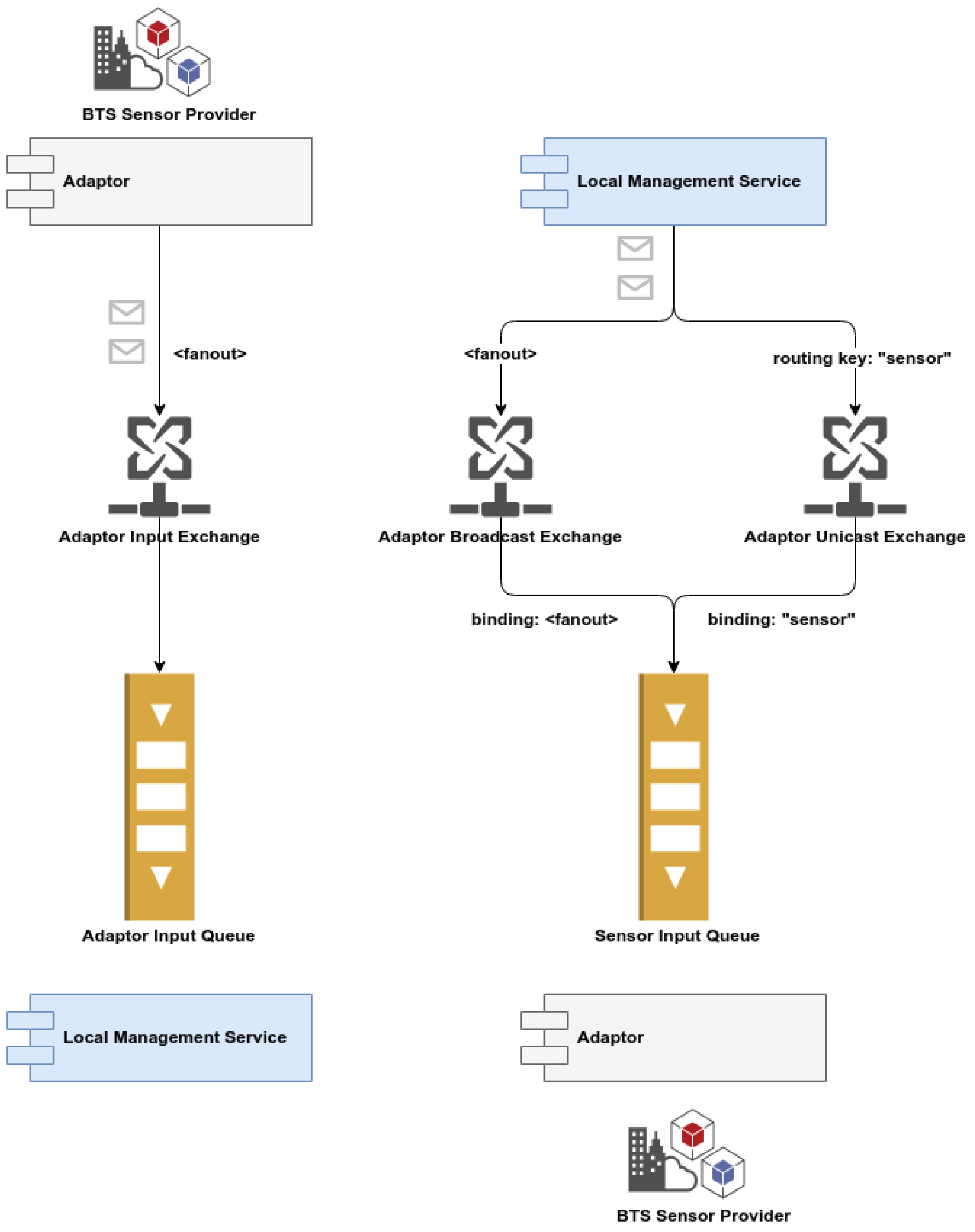


Figure 5.3: Messaging topology of a Local Management Service and a Provider Adaptor. We use the BTS sensor provider as an example

Request	**Payload**	**Response**	**Description**
REGISTER	Adaptor	N/A	This request is sent by the adaptor to register the adaptor with a Local Management Service
QUERY RESOURCES	Resource	UPDATE RESOURCES	This requests queries all provisioned resources owned by the provider
QUERY PROVIDER	Resource	UPDATE PROVIDERS	This request queries the provider for all its available resources, that is to say, all the resources that this provider is able to provision
PROVISION	Resource	CONTROL RESULT	This request controls the provider to provision a resource that is provided along with the request body
DELETE	Resource	CONTROL RESULT	This request deletes an existing resource that is provided along with the request body
CONFIGURE	Resource	CONTROL RESULT	This request configures an existing resource that is provided along with the request body
GET LOGS	Resource	(Provider specific)	This request fetches monitoring information of an existing resource

Table 5.2: An overview of the communication protocol between Provider Adaptor and Local Management Service

Request	REGISTER
Request	This request is sent by the adaptor to register the adaptor with a Local Management Service. The payload simply includes the adaptor UUID. Once the Local Management Service receives this request, the adaptor is registered and the Local Management Service will begin to query provider data. Moreover, the local will be able to control the provider through this adaptor.
Request Payload	Adaptor 4.7
Response	There is no response received, once the adaptor is registered, the Local Management Service will simply begin using the adaptor.
Response Payload	N/A

Table 5.3: Messaging Protocol 1 of the Table 5.2

Request	QUERY RESOURCES
Request	This request queries all provisioned resources owned by the provider. This request is sent by the Local Management Service to an adaptor. Upon receiving the request, the adaptor gathers resource data from the provider and transforms all resource data to our information model.
Request Payload	Resource 4.2 [1]
Response	The adaptor replies with the response UPDATE_RESOURCES with a list of resource models. Upon receiving the response, the Loal Management Service updates its data repository with the resources received from the adaptor.
Response Payload	List«Resource» 4.2

Table 5.4: Messaging Protocol 2 of the Table 5.2

[1] can be only a partial description

Request	QUERY PROVIDER
Request	This request queries the provider for all its available resources, that is to say, all the resources that this provider is able to provision. Upon receiving the request the adaptor interfaces with the provider and retrieves this data and transforms it to our information model
Request Payload	Resource 4.2 [2]
Response	The adaptor replies with the response UPDATE_PROVIDER with a list of resource models and additional provider metadata. Upon receiving the response, the Local Management Service updates its data repository with the provider information received from the adaptor.
Response Payload	List«Resource» 4.2

Table 5.5: Messaging Protocol 3 of the Table 5.2

Request	PROVISION
Request	This request controls the provider to provision a resource. The request contains the description of the resource to provision. This request is sent by the Local Management Service only to the adaptor that is responsible for the resource to be provisioned.
Request Payload	Resource 4.2
Response	Upon receiving the request, the adaptor parses the provided resource description and provisions the resource through the provider using the parameters and metadata provided through the resource description. The response body contains the execution result of the provision control. A successful execution will return the appropriate status and the provisioned resource. A failed execution will return the appropriate status and error messages.
Response Payload	Resource 4.2

Table 5.6: Messaging Protocol 4 of the Table 5.2

[2]can be only a partial description

Request	DELETE
Request	This request deletes an existing resource that is provided along with the request body. This request is sent by the Local Management Service only to the adaptor that is responsible for the resource to be deleted.
Request Payload	Resource 4.2
Response	Upon receiving the request, the adaptor parses the provided resource description and deletes the resource through the provider using the parameters and metadata provided through the resource description. The response body contains the execution result of the delete control. A successful execution will return the appropriate status and the deleted resource. A failed execution will return the appropriate status and error messages.
Response Payload	Resource 4.2

Table 5.7: Messaging Protocol 5 of the Table 5.2

Request	CONFIGURE
Request	This request configures an existing resource that is provided along with the request body. This request is sent by the Local Management Service only to the adaptor that is responsible for the resource to be configured.
Request Payload	Resource 4.2
Response	Upon receiving the request, the adaptor parses the provided resource description and configures the resource through the provider using the parameters and metadata provided through the resource description. The response body contains the execution result of the configure control. A successful execution will return the appropriate status and the configured resource. A failed execution will return the appropriate status and error messages.
Response Payload	Resource 4.2

Table 5.8: Messaging Protocol 6 of the Table 5.2

Request	GET LOGS
Request	This request fetches monitoring information of an existing resource. The monitoring data retrieved depends on what the provider makes available to be used (e.g. google stackdriver). This request is sent by the Local Management Service only to the adaptor that is responsible for the resource.
Request Payload	Resource 4.2
Response	Upon receiving the request, the adaptor parses the provided resource description fetches the available monitoring data through the provider. The response body has no structure and depends on the format of the data made available by the resource provider.
Response Payload	N/A (Provider specific)

Table 5.9: Messaging Protocol 7 of the Table 5.2

5.3.2 Global Management Service and Local Management Services

Figure 5.4 presents an overview of the messaging topology between a Global Management Service and the Local Management Services that it controls. Like with the Provider Adaptors, the Local Management Services must self register to a Global Management Service.

The Global Management Service uses only one queue to receive messages from ites Local Management Services. However, a Global Management Service can use multiple consumers to process the messages faster (In fact this is done with multi-threading thanks to Spring AMQP). Since only one queue is used to receive all Local Management Service messages, it is bound to a simple fanout exchange (an exchange sends received messages to all queues bound to it).

The Local Management Service only uses one input queue to receive the messages sent from the Global Management Service. This input queue is bound to three different exchanges upon its registration which is illustrated in Figure 5.4. The Local Management Service's broadcast exchange is a simple fanout exchange that routes a message to all queues that are bound to it. This effectively means that a message sent to the broadcast exchange will reach all the Local Management Services that are handled by the Global Management Service.

The Local Management Service's unicast exchange routes messages to specific Local Management Services whose queues are bound to a routing key that is the ID of the Local Management Service. Similarly, the Local Management Service's groupcast exchange routes messages to queues that are routed to the ID of a group of Local Management Services.

Similar to the Local Management Service and Adaptors, the protocol involves pairs of request and response messages. Table 5.10 gives an overview of the different types of messages that are used in the communication between the Global Management Service and its Local Management Services. Tables 5.11 to 5.17 describe the specific protocols.

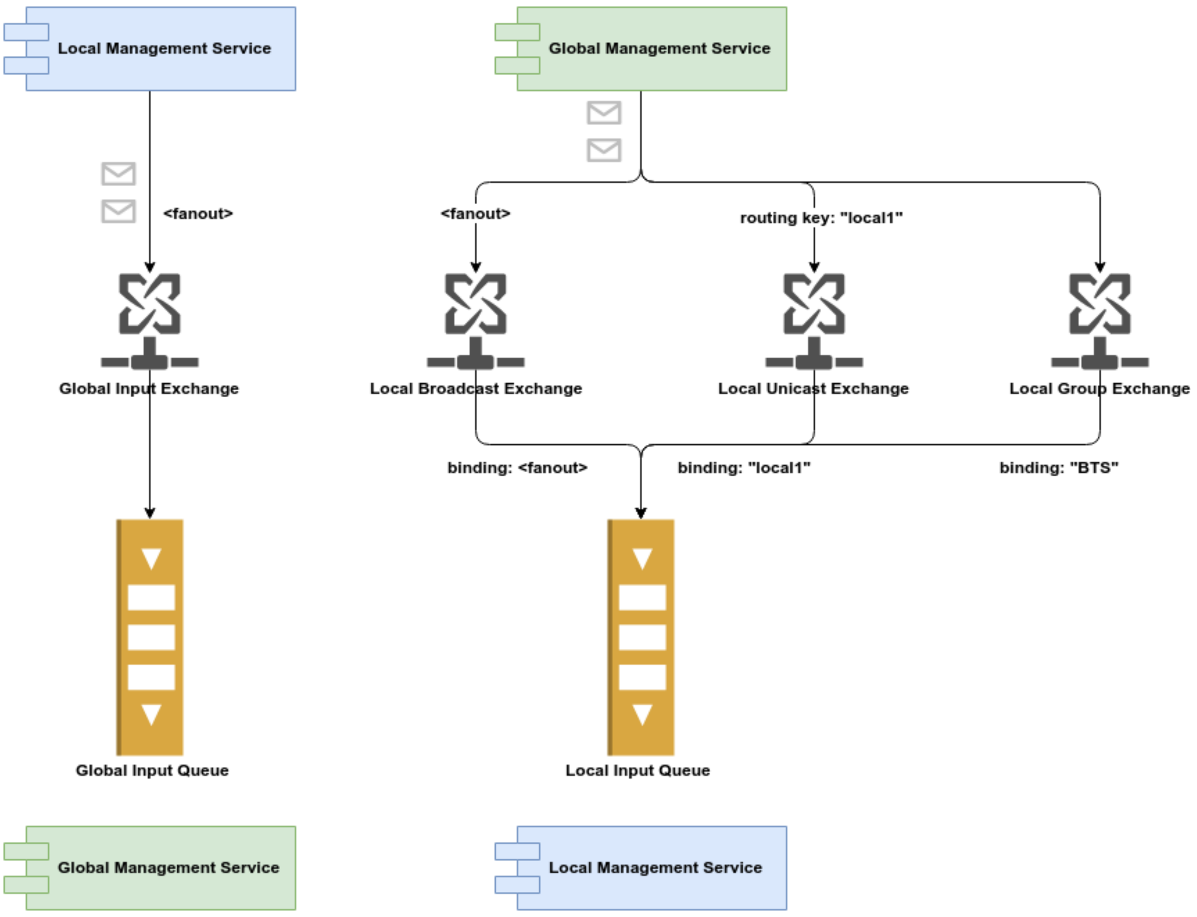


Figure 5.4: Messaging Topology of a Global Management Service and Local Management Services

Request	**Payload**	**Response**	**Description**
REGISTER	Local Management Service	N/A	This request registers a Local Management Service with a Global Management Service
FETCH RESOURCES	Resource	DELIVER RESOURCES	This request fetches all provisioned resources from Local Management Services
FETCH PROVIDERS	Resource	DELIVER PROVIDERS	This request fetches all resources that are able to be provisioned through the affected Local Management Service
PROVISION	Resource	PROVISION RESULT	This request is sent by a Global Management Service to a Local Management Service to provision a specified resource
DELETE	Resource	DELETE RESULT	This request is sent by a Global Management Service to a Local Management Service to delete a specified resource
CONFIGURE	Resource	CONFIGURE RESULT	This request is sent by a Global Management Service to a Local Management Service to configure a specified resource
GET LOGS	Resource	GETLOGS RESULT	This request is sent by a Global Management Service to a Local Management Service to fetch available monitoring data for a specific resource

Table 5.10: An overview of the communication protocol between Global and Local Management Services

Request	REGISTER
Request	This request registers a Local Management Service with a Global Management Service. The payload includes the ID, group ID and other metadata about the Local Management Service. The Local Management Service is registered, and its providers can be queried and controlled by the Global Management Service.
Request Payload	Local Management Service 4.8
Response	There is no response received, once the Local Management Service is registered the Global Management Service will simply begin to use the Local Management Service in queries and controls.
Response Payload	N/A

Table 5.11: Messaging Protocol 1 of the Table 5.10

Request	FETCH RESOURCES
Request	This request fetches all provisioned resources from Local Management Services. The payload may contain a JSON query with a complete or partial resource model to filter by different resource attributes. This request is sent by the Global Management Service to Local Management Services
Request Payload	Resource 4.2
Response	Upon receiving this request the affected Local Management Service sends resource queries to the providers that it handles through its provider adaptors. The Global Management Services waits until a set timeout for all the resources that are sent back from the different Local Management Services.
Response Payload	List«Resource» 4.2

Table 5.12: Messaging Protocol 2 of the Table 5.10

Request	FETCH PROVIDERS
Request	This request fetches all resources that are able to be provisioned through the affected Local Management Service (i.e. a resource catalogue). The payload may contain a JSON query with a complete or partial resource model to filter by different resource attributes. This request is sent by the Global Management Service to Local Management Services
Request Payload	Resource 4.2
Response	Upon receiving this request the affected Local Management Service sends provider queries to the providers that it handles through its provider adaptors. The Global Management Services waits until a set timeout for all the resources that are sent back from the different Local Management Services. The resources received have not actually been deployed and might contain some placeholders or descriptions of fields that will be populated once they have been provisioned.
Response Payload	List«Resource» 4.2

Table 5.13: Messaging Protocol 3 of the Table 5.10

Request	PROVISION
Request	This request is sent by a Global Management Service to a Local Management Service to provision a specified resource.
Request Payload	Resource 4.2
Response	Upon receiving this request the affected Local Management Service sends provider provision controls to the provider that can provision this resource. Once the Local Management Service gets the provisioned resource description (with all metadata and parameters) from the adaptor, the resource is sent back to the Global Management Service.
Response Payload	Resource 4.2

Table 5.14: Messaging Protocol 4 of the Table 5.10

Request	DELETE
Request	This request is sent by a Global Management Service to a Local Management Service to delete a specified resource.
Request Payload	Resource 4.2
Response	Upon receiving this request the affected Local Management Service sends provider provision controls to the provider that can delete this resource. Once the Local Management Service gets confirmation that the resource has been deleted from the adaptor, the resource description is sent back to the Global Management Service.
Response Payload	Resource 4.2

Table 5.15: Messaging Protocol 5 of the Table 5.10

Request	CONFIGURE
Request	This request is sent by a Global Management Service to a Local Management Service to configure a specified resource.
Request Payload	Resource 4.2
Response	Upon receiving this request the affected Local Management Service sends provider provision controls to the provider that can configure this resource. Once the Local Management Service gets confirmation that the resource has been reconfigured from the adaptor, the updated resource description is sent back to the Global Management Service.
Response Payload	Resource 4.2

Table 5.16: Messaging Protocol 6 of the Table 5.10

Request	GET LOGS
Request	This request is sent by a Global Management Service to a Local Management Service to fetch available monitoring data for a specific resource.
Request Payload	Resource 4.2
Response	Upon receiving this request the affected Local Management Service sends provider provision controls to the provider that can provide monitoring data for this resource. Once the Local Management Service receives this data from the adaptor, it is sent back to the Global Management Service.
Response Payload	N/A (Provider specific)

Table 5.17: Messaging Protocol 7 of the Table 5.10

5.4 Deployment

To run our framework, all the described software components need to be deployed to an infrastructure. The design of the architecture allows several different deployment models. In this section we discuss two different types of deployment models, that are used in different deployment contexts.

In the deployment models we do not mention the Slice Management Client. Since the Slice Management Client is a command line interface, we expect the service to be distributed among the users of our framework. In any case, the Slice Management Client simply uses the services exposed by the Global Management Service. However, the Slice Management Client is responsible for creating the Service Mesh which will always be deployed in a dedicated cloud datacenter.

5.4.1 Deployment Model 1

Figure 5.5 presents our first deployment model. It exploits the usage of cloud datacenters to house all our services. We deploy all the services of our framework into a a cloud datacenter. This provides our global and local management services with a large amount of scalability. Google Cloud Platform [SRR18] provides auto-scaling for instances up to specifications of up to 96 cores with options of high memory or high CPU instances. Although in reality our framework will never use the highest levels of scalability provided by a cloud datacenter, it assures that sufficient resources will always be available if needed to handle high load. Additionally we have the option of using less resources and can mitigate the costs of keeping the framework running.

In this deployment model we assume that we have been granted open access to organizations and entities that own the resource providers in our system. The access to resource providers should be available publicly through the Internet since in our deployment model the provider adaptors are also deployed in the cloud datacenter. In this situation we also assume that the resource providers have little to no participation in the deployment of the framework since all access to the resource providers are free.

The external services in our framework consist of the RabbitMQ [Piv18] broker and the MongoDB [Inc17] cluster. These services may be deployed along with our framework in the same datacenter. Additionally, these services can be managed by external providers. There are many DaaS providers [mla18][Inc18b] that can host MongoDB clusters in different cloud data centers. Likewise, Messaging as a Service providers [84c18a] exists that can host RabbitMQ clusters in different cloud data centers. In this deployment model, where these external services are deployed is not important to our deployment model. We assume that we use the external services. Therefore we increase the number of cloud datacenters we use by two.

Since we use AMQP as our communication throughout the framework. We must make sure that the RabbitMQ broker is in a network infrastructure where it is publicly available. This way, we can we can deploy the other services regardless of their network infrastructures as long as they have an available Internet connection and configured to use the broker.

The Global Management Service is the central point of communication to interact with resource providers through our unified API. Multiple instances can be deployed and load balanced in order to achieve a higher fault tolerance. Once possible API gateway is Kong [Inc18a] which is based on the widely adopted NGINX HTTP server that processes client requests to upstream services. The lower bound deployment of one instance is still functional but in the event of a failure in the Global Management Service, the clients will lose access to all the resource providers. However, the configuration required for clients is still the same regardless of the number of instances, they will only use one IP/Domain to access the APIs offered by the Global Management Service.

According to our description in Chapter 4, the Local Management Service manages the resources in an organization, so therefore instances deployed depends on the number of logical/organizational separations among the resource providers. A deployment may include more than one instance of a Local Management Service to the same set of resource providers to increase fault tolerance but we do not take this direction. Although they are single points of failure regarding the providers that they interface with, the message based communication with RabbitMQ provides the Local Management Service with a high amount of fault tolerance since messages that are queued will not be lost on failure. In case of a failover, messages that were not handled will be consumed on a restart in the order that they are queued. Since we deploy our Local Management Services in a cloud datacenter, we can assume that in the event of a failure, the actual downtime of the service will be short. Google Cloud Platform, already has mechanisms in place [SRR18] to detect the crash of an application running in a VM (Compute Engine) or a container (Container Engine). New instances will subsequently be started.

Since in this deployment model, we assume public access to all resource providers over the Internet, the provider adaptors can be deployed in the cloud datacenter along with all our other services. The deployment on the cloud means that we can also leverage the fault tolerance of the message based communication to deploy only one Adaptor per resource provider.

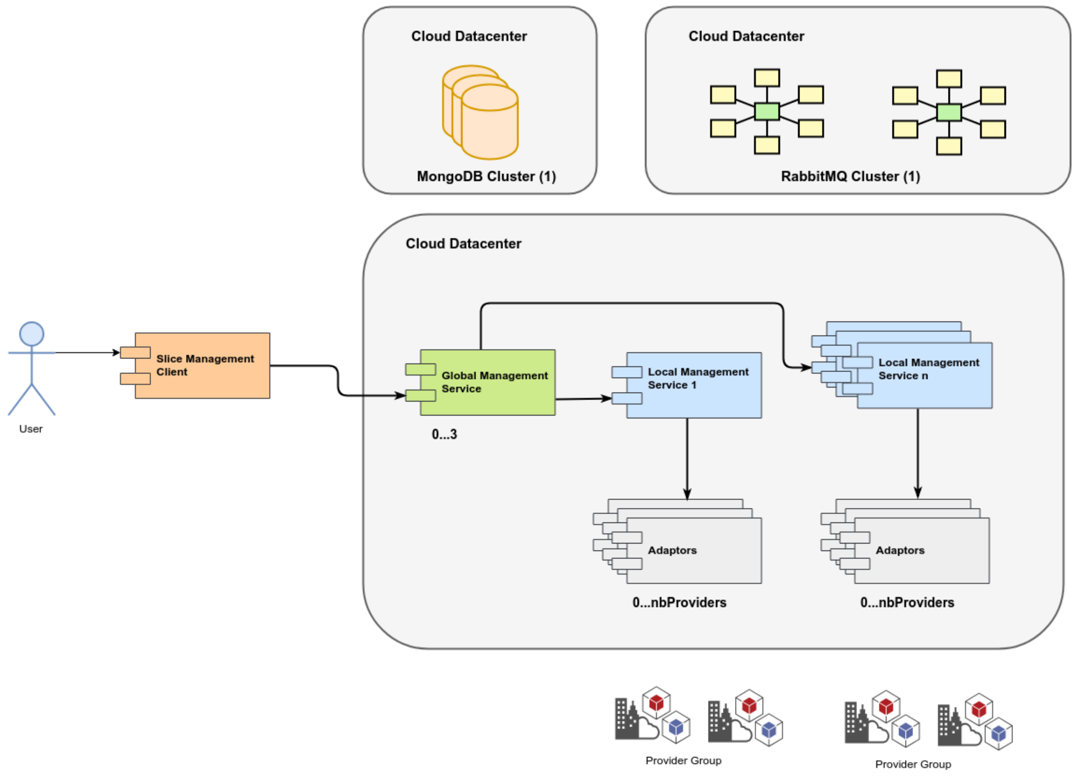


Figure 5.5: Illustration of Deployment Model 1

5.4.2 Deployment Model 2

Figure 5.6 presents our second deployment model. This deployment is similar to our first deployment model. However in this deployment model, we assume that we no longer have open access to the organizations that own the resource providers. In this situation we can make the assumption that resource providers are a part of large organizations in industry that do not wish to reveal sensitive details of their resource APIs and processes.

We can see two differences in the deployment compared to the first deployment model. The two differences both accommodate certain resource providers' wish that the owner of our framework does not get a knowledge of their internal APIs. The first change in deployment is the non-deployment of provider adaptors. The deployment of the adaptors can be outsourced to the organizations. Since the adaptors interface with our framework through self-registration, we only need to deploy a Local Management Service and communicate the RabbitMQ details to the organizations. The resource providers take the responsibility of harmonizing data and unifying their API with that of our framework.

The second change could be the outsourcing of the entire Local Management Service cluster to these organizations. This would occur in the event that an organization that owns several resource providers wishes to control access to their resources. Since the Local Management Service manages the resources in an organization, the idea that the entire responsibility of deploying and managing it goes to the organization is not infeasible. In this case we communicate the RabbitMQ broker details to the organization so that they can register their Local cluster with the Global Management Service.

Furthermore, in this deployment model we can make a distinction between the capabilities of the organizations who choose to handle their own deployments. We first consider the case of a large organization which has access to a cloud data center where deployment of Provider Adaptors and Local Management Services mirror our own in the first deployment model.

Additionally, we can also consider the other extreme case where the organization has limited capabilities. Without access to a cloud datacenter this type of provider would rely on edge devices which could be a specific edge node or simply old desktop PCs that can house running instances of Provider Adaptors. In this case the resource providers will most likely be deployed in other devices in a closed network. However, we do not treat resource providers as a part of our framework, and hence we do not treat them as a part of this deployment model. This edge case can also apply to organizations who cannot use cloud datacenters, for example political organizations such as governments or international bodies [ZT13].

Having taken responsibility for their own deployments of provider adaptors or Local Management Services, these organizations must also be responsible for the uptime and maintenance of these services.

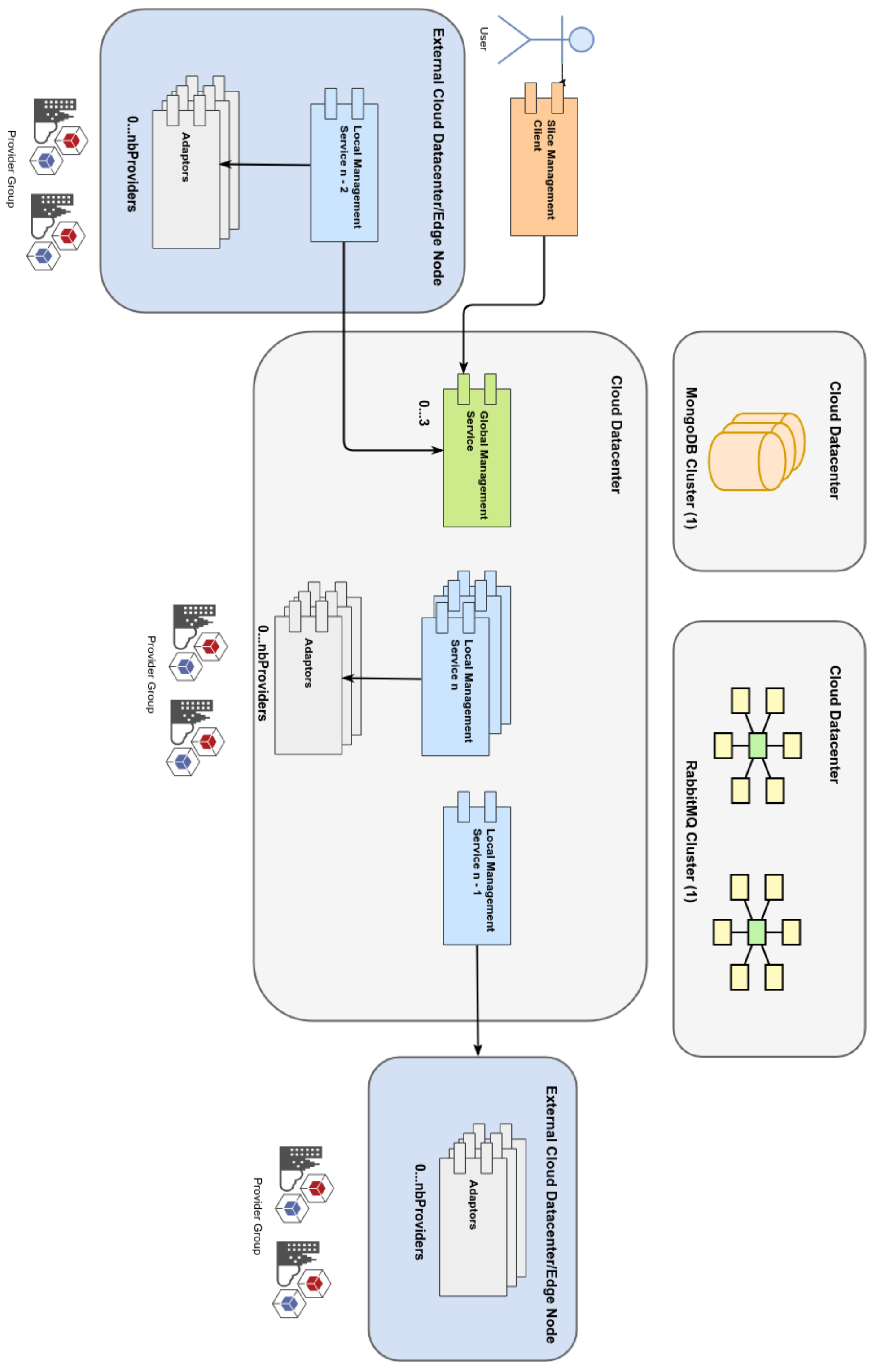


Figure 5.6: Illustration of Deployment Model 2

5.5 Summary

In this chapter we presented the prototype implementation of our deployment tool. The final prototype uses implementations of the software services that were presented in Chapter 4 as part of the framework design. Additionally since there are many different deployable software services which make up our framework, we have presented two deployment models that utilize both cloud and edge platforms under different contexts. The deployment models represent the different degrees of openness to resource provider access that organizations can grant our framework.

CHAPTER 6

Evaluation

6.1 Test System

In order to thoroughly evaluate the features of our prototype, we must evaluate the features with respect to our motivating scenario. One of the main contributions of our work is a novel method of using resource slices to implement IoT Cloud Systems. In order the do this, we must first have a test system with a functional set of resources and resource providers. We can then use this test system with our deployment framework to create resource slices. We implement resources and providers that are part of our motivating scenario in Chapter 3.

6.1.1 Resource and Providers

In order to evaluate the features of our prototype implementation using the identified use cases from Chapter 3. We implemented resources and providers ourselves. We try to use as many third party resources and providers as possible such as Cloud MQTT brokers, Google Firewall or BigQuery. However, we do need to implement certain resources with specific behaviors that we use in our scenarios. We implemented these resources and their providers using Node.js due to the short amount of time needed to setup and maintain a simple project without being constrained to the complications of using an established language like java, such as multi-threading and compiling.

In the scope of our work we implemented the following resources:

- **Generic Sensor** - This simple software module reads data provided in a CSV formatted file and outputs the result in either CSV or JSON format in a selection of protocols that include MQTT, AMQP or HTTP. We use this resource to simulate the sensor data streams of our scenarios by providing the real sensor data we mentioned in Chapter 3.

- **Simple Analytics Client** - This software module can ingest data through a selection of communication protocols that include MQTT, AMQP or HTTP. Since the focus of our work is not data analytics, we mock a simple mechanism for implementing the analytics. Any data ingested by this module can be output in a variety of data sinks. These data sinks can be storage solutions such as Google BigQuery[Goo18b]. Additionally, there is also a data sink to a HTTP POST endpoint.
- **Mosquitto MQTT Broker** - This MQTT broker [Fou12] is developed by the eclipse foundation. We simply package a version of it for easy deployment in our scenarios.
- **Node-RED** - Node-RED [Fou18a] provides a browser-based flow editor that makes it easy to wire together flows using the wide range of nodes in the palette. Flows can be then deployed to the runtime in a single-click. JavaScript functions can be created within the editor using a rich text editor. A built-in library allows you to save useful functions, templates or flows for re-use [Fou18a]. We use Node-RED as simple software artifacts that can support multiple input and output sources including MQTT, HTTP and TCP. Node-RED is extensible and new nodes can be created through plugins.

For each of these custom resources, we also implement a provider that will provision, configure and monitor these resources. These providers are implemented in Node.js and are simple REST servers that act as management APIs, that we can see in many industry service providers such as Cloud AMQP [84c18a], MongoDB Atlas [Inc18b] or Google BigQuery [Goo18b] .

The resources themselves are deployed by their providers on in the form of docker containers on the Google Container Engine (GCE) using Kubernetes [Pla14]. We chose to use the Kubernetes engine on Google Cloud to deploy our resource for several reasons:

- GCE supports a high level of automation. Creating container deployments and exposing the deployment with a public IP takes a matter of seconds and can be completely automated using JSON templates.
- GCE allows us to easily limit what resources a container deployment can have. We can use this feature to limit CPU and memory in order to simulate low resource edge devices and to create sets of test environments.
- GCE can expose containers directly with an external IP. However, we can also take advantage of the internal DNS that resolves services based on their name. This lets set up different test environments very quickly without having to wait for allocation of external IP addresses.
- GCE has its own firewall service and container clusters can be configured to follow firewall rules. This can be the network function service provider in our test system.

We also interfaced with third party providers to obtain the following resources:

- **Google BigQuery** - Google BigQuery[Goo18b] is a cloud-based big data analytics web service for processing very large read-only data sets. BigQuery was designed for analyzing data on the order of billions of rows, using a SQL-like syntax. It runs on the Google Cloud Storage infrastructure and can be accessed with a REST-oriented application program interface (API) or a variety of client APIs developed and supported by Google.
- **Google Firewall** -The Google Cloud Platform provides a firewall service [SRR18] that lets you deny or allow traffic from VM instances or Kubernetes containers based on a JSON configuration that you specify. GCP firewall rules are set on a virtual networking level, so they work as a physical firewall would.
- **Cloud MQTT** - Cloud MQTT [84c18b] is provided by 84codes - a Swedish tech company that supplies cloud infrastructure for developers. Cloud MQTT is a "Messaging as a Service" product that provisions managed instances of RabbitMQ brokers that use the MQTT. Cloud MQTT also provides a management API available through REST.

In the case of third party providers, we do not need to implement any resource or provider code. We need only interface the provider to our framework with a Provider Adaptor implementation.

We implemented all the Adaptors for the resources and providers in Section 6.1.1. The implementations were in Node.js using the same structure in order to save development time. Each of our adaptor implementations takes a configuration as input that includes:

- **ADAPTOR NAME** - This serves as the ID of the Adaptor that is used by the Local Management Service.
- **BROKER URI** - This configuration is the connection string of the AMQP broker, that includes and username and password that might be required.
- **LOCAL EXCHANGE** - The routing key of the Local Management Service, this is exchange that the Local Management Service uses to receive Adaptor messages.

Our Adaptor implementations use additional configuration that are necessary to interface with their assigned providers, for example, an endpoint HTTP URL of the provider's API. However, these additional configuration values can change depending on each Adaptor implementation.

Table 6.1 gives an overview of the resources, providers and adaptors that we use in our test system. From the table it is clear which services and components that we implemented by providing the GitHub link to the source code. Resources that have been implemented and reused by external parties have been marked as such. Likewise, resources that are provided by cloud providers have also been marked.

Resource	**Resource Implemtation**	**Provider Implementation**	**Adaptor Implementation**
Generic Sensor	`/rdsea/IoTCloudSamples/tree/master/IoTCloudUnits/simplesensor`	`/rdsea/IoTCloudSamples/tree/master/IoTProviders/bts-sensor`	`/SINCConcept/HINC/tree/master/ext-plugin/bts-sensor-plugin`
Analytics Client	`/rdsea/IoTCloudSamples/tree/master/IoTCloudUnits/ingestionClient`	`/rdsea/IoTCloudSamples/tree/master/IoTProviders/bts-ingestion-provider`	`/SINCConcept/HINC/tree/master/ext-plugin/bts-ingestion-plugin`
Mosquitto MQTT	External	`/rdsea/IoTCloudSamples/tree/master/IoTProviders/mosquitt-mqtt-provider`	`/SINCConcept/HINC/tree/master/ext-plugin/mqtt-plugin`
Node-RED	External	`/rdsea/IoTCloudSamples/tree/master/InterOpProviders/nodered-datatransformer-provider`	`/SINCConcept/HINC/tree/master/ext-plugin/nodred-plugin`
Google BigQuery	External	Cloud Provider	`/SINCConcept/HINC/tree/master/ext-plugin/bigquery-plugin`
Google Firewall	External	`/rdsea/IoTCloudSamples/tree/master/IoTProviders/kubernetesFirewallProvider`	`/SINCConcept/HINC/tree/master/ext-plugin/kubefirewall`
Cloud MQTT	External	Cloud Provider	`/SINCConcept/HINC/tree/master/ext-plugin/cloudmqtt`

Table 6.1: Resource, provider and adaptor implementations used in our test system. Append *https://www.github.com* to URL paths

6.1.2 Deployment Configuration

The test system we use can be seen in Figure 6.1. We utilize a simple deployment that consists of one Slice Management Client, Global Management Service and Local Management Service. We use six different resource providers along with one adaptor attributed to each provider. The choice of providers fulfills the different kinds of resources that we need for our scenario shown in Figure 3.1.

We use the minimum cardinalities of the Global and Local Management Services. This means that we deploy one instance each. For the Local Management System, we deploy one of each resource provider that we implemented for the test system. Then, for each resource provider we deploy one instance of a Provider Adaptor to register the provider with the Local Management Service. For the functional evaluation, the low cardinalities of the deployed services in our system are of no consequence. Moreover, since we only have a limited number of resource providers, it does not make sense to create several Local Management Services during the experiments just to double the same resources that we deploy. We are developing a novel framework, with no real related tool which can provide a similar feature base. For this reason, we would like to evaluate the performance of the tools of our framework and not the performance of deployment with solutions that mitigate the effect of a large load. Therefore we do not deploy multiple Global Management Services that are load balanced behind an API gateway. We also do not deploy multiple Adaptors for a resource provider to increase message consumption/handling. Since our implementation is only a prototype and not a production ready product, we should test with the most basic deployment configuration of services to evaluate the implementations of the services. Using deployment techniques to mitigate performance stress would be counterproductive to our evaluation.

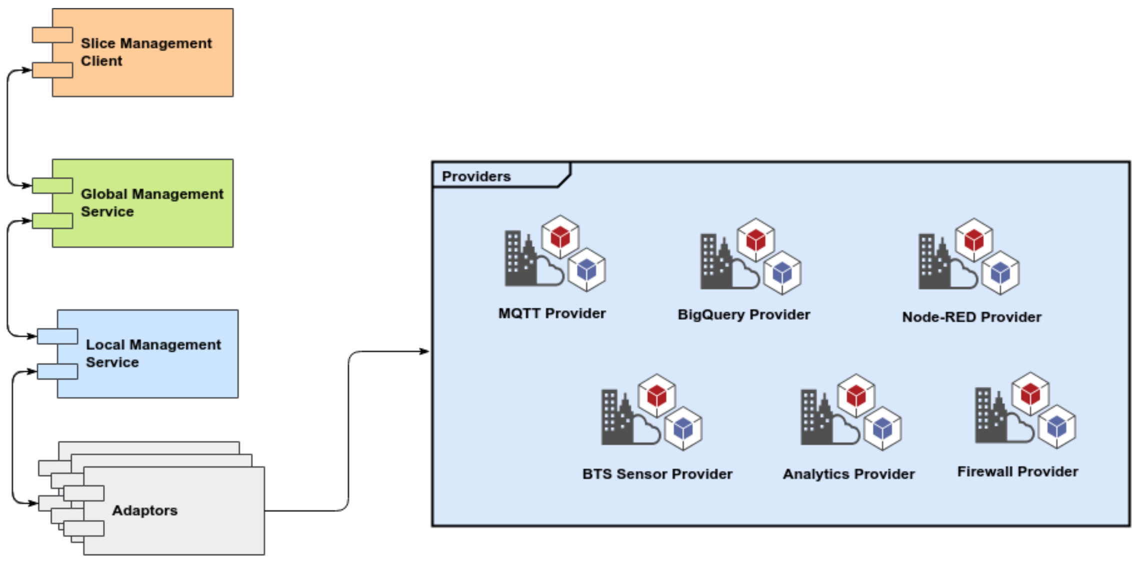


Figure 6.1: An overview of our test system

6.2 Functional Evaluation

6.2.1 Experimental Setting

In order to show that we have attained our use cases set out in Chapter 3. We conduct the following four experiments that fulfill selected use cases in Figure 1.2. Since thoroughly evaluating every use case is time consuming, we choose these use cases to evaluate, since they cover the maximum amount of functional requirements in Table 3.21.

The experiments can all be found on GitHub `https://github.com/SINCConcept/HINC/tree/master/scenarios/btssensors`, along with the scripts and configuration files required to run them. The GitHub repository also includes a set of deployment scripts which help to deploy a running system. We deploy our proposed test system in Figure 6.1 following the cloud-only model as seen in Section 5.4.1.

The deployment takes place mostly on the Google Cloud Platform Kubernetes Engine [Pla14]. We use Kubernetes to take advantage of the internal DNS which avoids any manual IP configurations of our services. Table 6.2 gives and overview of the test system deployment.

Most of the components of our test system are deployed using containers. Containers deployment is automated and easier to configure since we don not have to worry about configuring IP addresses thanks to the Google Container Engine [Pla14]. However, the specifications that we declare for each of our containers (CPU, memory) do have a margin of error due to the scheduling process of Kubernetes [SRR18].We create and manage the resource ensembles in our experiments by using *Pizza* on a Dell XPS laptop through the command line terminal. We save all deployment configurations in JSON file templates which can be reused for future experiments. We also use a MongoDB Atlas cluster and a CloudAMQP cluster as our external services.

6.2.2 Criteria

We evaluate the usage of our framework against a set of criteria that measures certain non-functional qualities regarding user experience. The evaluation of these non-functional qualities below, is essential in determining the effectiveness of our framework in deploying resource slices. To this end, we run experiments with our framework to implement the use cases stated in Chapter 3.

- **Learnability** - How quickly a new user can begin efficient and error-free interaction with the framework.
- **Operability** - The level of effort required for a user to interact with the framework, we also use the vernacular term *Simplicity* to avoid ambiguity with specific operations that we effect in our experiments.
- **Attractiveness** - To what degree would a user use our framework compared to an alternative solution

- **Flexibility** - How easy the framework facilitates complex operations for an experienced user.

These first two criteria of Learnability and Operability can be found in the software quality requirements from the ISO/25010 system and software quality models standard [ISO10]. The second set of criteria of Attractiveness and Flexibility are also from the standard ISO/20150 [ISO10] but are a part of the Systems and software Quality Requirements and Evaluation (SQuaRE) – Systems and software quality models. We chose the criteria that would provide the most value to our work. Unfortunately the whole suite of quality requirements and models cannot be evaluated in the scope of this thesis.

The evaluation of our framework with respect to the above criteria is carried out by us. We conduct the experiments through role play, by using our framework as we would expect the end user to. This technique is often used in formal requirements review [rol][DS16], which is appropriate for our evaluation since we aim to evaluate the features of our framework.

6.2.3 Experiments

We base our functional evaluation on four experiments that are operations based on the richest use cases presented in Chapter 3. We explain our procedure of these experiments and then follow up with an evaluation of our framework's key features based on these experiments.

6.2.3.1 UC01 Add Data Consumer

Use Case 3.1. We begin with a simple slice description in JSON that consists of a MQTT broker, BigQuery dataset and an analytics client (example in Figure 6.2). We deploy this resource slice simulating the Service Operator declaring an empty pipeline for a new client. The client can then run a JSON query through *Pizza* specifically for IoT resources that are sensors (example of the query in Figure 6.6 and the result in Figure 6.7). The chosen sensor descriptions can then be added to the JSON slice description followed by an update command (Figure 6.5) to *Pizza* for the new slice description. The result is that the BigQuery dataset is being filled up with data points from the newly provisioned sensors that pass through the analytics client.

6.2.3.2 UC03 Protect Data Consumer

Use Case 3.3. This experiment begins by deploying a slice that is fully functional. We assume that in this case a client already possesses a functional data pipeline. The slice description can be submitted to *Pizza* to obtain monitoring information on the different resources in the slice. We assume that the Data Consumer has found a reason in this monitoring information to apply a firewall to the analytics client. A script is provided to ease the configuration of JSON firewall resource which can be quite tedious. The firewall must be accurately configured to avoid breaking the pipeline functionality of the slice. Once the firewall has been configured, it can be observed that the row count in the BigQuery database does not increase. A call to *Pizza* to retrieve the logs for the analytics client will show that new messages are still being consumed from the broker, but the inserts to the BigQuery database fail due to connection errors.

6.2.3.3 UC04 Custom Analysis Logic

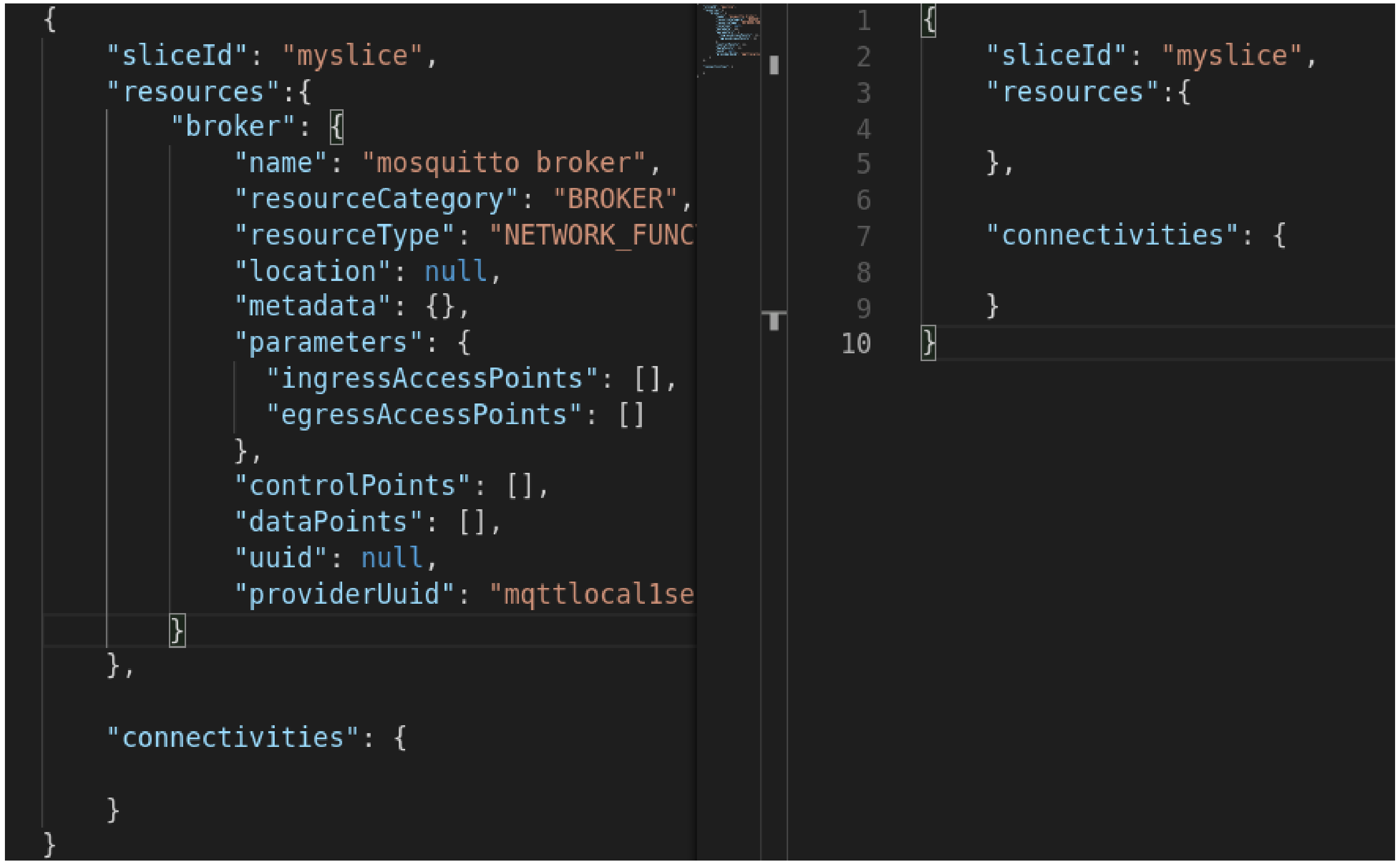
Use Case 3.4. This use case tries to increase the flexibility of a data pipeline by allowing custom logic to enter the pipeline. We begin by deploying a functional slice. We then deploy a Node-RED instance by adding it to the slice description and then submitting an update through *Pizza*. The URL endpoint for the Node-RED instance can be found in the updated slice description once *Pizza* has finished updating the slice. To simplify the process of creating a data flow in Node-RED, we have included an exported data flow that can be imported with one click into Node-RED so that the custom logic can be applied. In this case the custom logic is a function that adds an offset of 100 to the values received from the data sources. These changes can be visualized on the debug screen in Node-RED or in the BigQuery dataset. In our use case, the Node-RED data flow could have been prepared by a separate Data Engineer and handed to the client in a packaged form that can be imported, like in Node-RED.

6.2.3.4 UC09 Remove Data Pipeline

Use Case 3.5. This experiment simply involves provisioning a data pipeline slice and then deleting it and releasing its resources. This use case is important to show the on-demand aspect of resource slices. Once there is no more need for a resource slice, it can be deleted. A new slice can equally be provisioned afterwards.

Name	**Deployment**	**Type**	**Specs**
Global	Google Cloud[SRR18]	Container	2x2.0GHz CPU, 2048Mi RAM
Local	Google Cloud	Container	1x2.0GHz CPU, 1024Mi RAM
MongoDB	MongoDB Atlas [Inc18b]	Amazon EC2	512MB Storage, Shared RAM
RabbitMQ	CloudAMQP [84c18a]	Amazon EC2	Share CPU & RAM
Pizza	N/A	Dell XPS	8GB RAM, 2x1.8GHz, Intel Core i5,
Mosquitto Adaptor	Google Cloud	Container	0.5x2.0GHz CPU, 256Mi RAM
Analytics Adaptor	Google Cloud	Container	0.5x2.0GHz CPU, 256Mi RAM
Sensor Adaptor	Google Cloud	Container	0.5x2.0 GHzCPU, 256Mi RAM
BigQuery Adaptor	Google Cloud	Container	0.5x2.0 GHzCPU, 256Mi RAM
Node-RED Adaptor	Google Cloud	Container	0.5x2.0 GHzCPU, 256Mi RAM
Firewall Adaptor	Google Cloud	Container	0.5x2.0 GHz CPU, 256Mi RAM
BigQuery Provider	Google BigQuery	N/A	N/A
Mosquitto Provider	Google Cloud	Container	0.5x2.0 GHz CPU, 256Mi RAM
Analytics Provider	Google Cloud	Container	0.5x2.0 GHz CPU, 256Mi RAM
Sensor Provider	Google Cloud	Container	0.5x2.0 GHz CPU, 256Mi RAM
Node-RED Provider	Google Cloud	Container	0.5x2.0 GHz CPU, 256Mi RAM
Firewall Provider	Google Cloud	Container	0.5x2.0 GHz CPU, 256Mi RAM
BTS Sensor	Google Cloud	Container	0.1x2.0 Ghz CPU, 100Mi RAM
Mosquitto Broker	Google Cloud	Container	0.1x2.0 Ghz CPU, 100Mi RAM
Analytics Client	Google Cloud	Container	0.5x2.0 Ghz CPU, 256Mi RAM
Node-RED Instance	Google Cloud	Container	0.5x2.0 Ghz CPU, 256Mi RAM
BigQuery Dataset	Google BigQuery [Goo18b]	N/A	N/A
Google Firewall	Google VPC [SRR18]	Container	0.5x2.0 GHz CPU, 256Mi RAM

Table 6.2: Experimental setting for functional evaluations



```
{
    "sliceId": "myslice",
    "resources":{
        "broker": {
            "name": "mosquitto broker",
            "resourceCategory": "BROKER",
            "resourceType": "NETWORK_FUNC
            "location": null,
            "metadata": {},
            "parameters": {
              "ingressAccessPoints": [],
              "egressAccessPoints": []
            },
            "controlPoints": [],
            "dataPoints": [],
            "uuid": null,
            "providerUuid": "mqttlocal1se
        }
    },

    "connectivities": {

    }
}
```

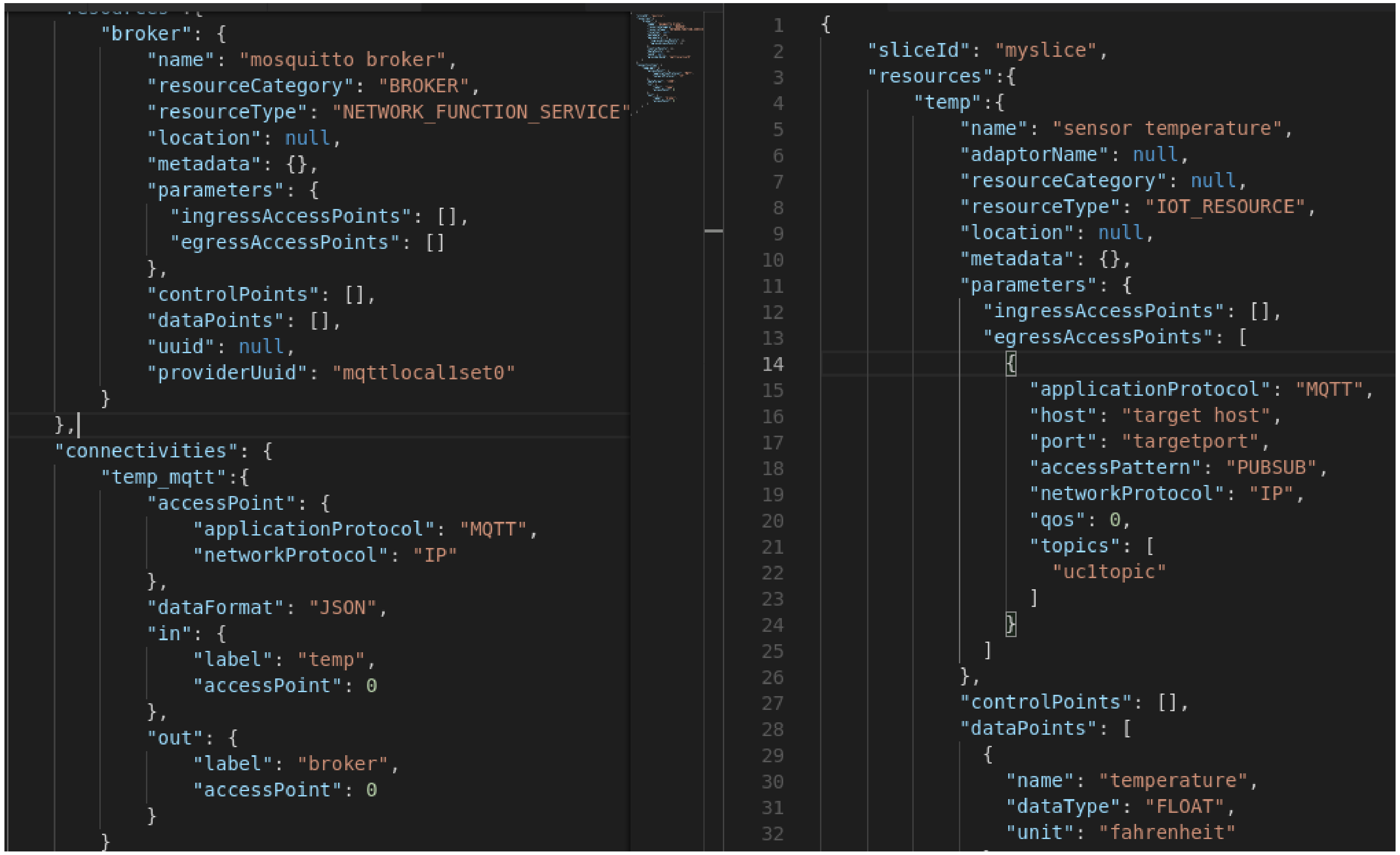
```
{
    "sliceId": "myslice",
    "resources":{

    },

    "connectivities": {

    }
}
```

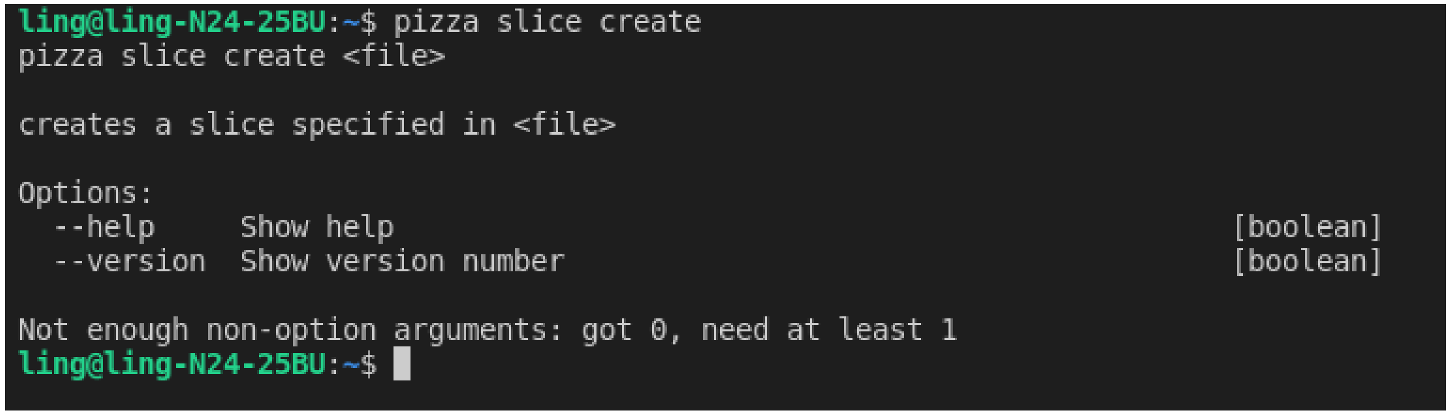
Figure 6.2: Creating a slice description in JSON and adding an MQTT broker resource



```
        "broker": {
            "name": "mosquitto broker",
            "resourceCategory": "BROKER",
            "resourceType": "NETWORK_FUNCTION_SERVICE",
            "location": null,
            "metadata": {},
            "parameters": {
              "ingressAccessPoints": [],
              "egressAccessPoints": []
            },
            "controlPoints": [],
            "dataPoints": [],
            "uuid": null,
            "providerUuid": "mqttlocal1set0"
        }
    },
    "connectivities": {
        "temp_mqtt":{
            "accessPoint": {
                "applicationProtocol": "MQTT",
                "networkProtocol": "IP"
            },
            "dataFormat": "JSON",
            "in": {
                "label": "temp",
                "accessPoint": 0
            },
            "out": {
                "label": "broker",
                "accessPoint": 0
            }
        }
```

```
{
    "sliceId": "myslice",
    "resources":{
        "temp":{
            "name": "sensor temperature",
            "adaptorName": null,
            "resourceCategory": null,
            "resourceType": "IOT_RESOURCE",
            "location": null,
            "metadata": {},
            "parameters": {
              "ingressAccessPoints": [],
              "egressAccessPoints": [
                {
                  "applicationProtocol": "MQTT",
                  "host": "target host",
                  "port": "targetport",
                  "accessPattern": "PUBSUB",
                  "networkProtocol": "IP",
                  "qos": 0,
                  "topics": [
                    "uc1topic"
                  ]
                }
              ]
            },
            "controlPoints": [],
            "dataPoints": [
              {
                "name": "temperature",
                "dataType": "FLOAT",
                "unit": "fahrenheit"
```

Figure 6.3: A sensor and broker in a slice, along with their connectivity



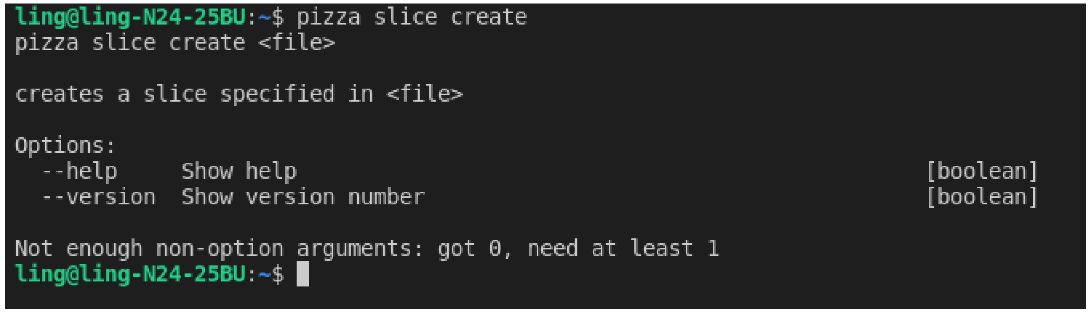
```
ling@ling-N24-25BU:~$ pizza slice create
pizza slice create <file>

creates a slice specified in <file>

Options:
  --help     Show help                                                  [boolean]
  --version  Show version number                                        [boolean]

Not enough non-option arguments: got 0, need at least 1
ling@ling-N24-25BU:~$ 
```

Figure 6.4: Using *Pizza* to provision a slice from a JSON description file



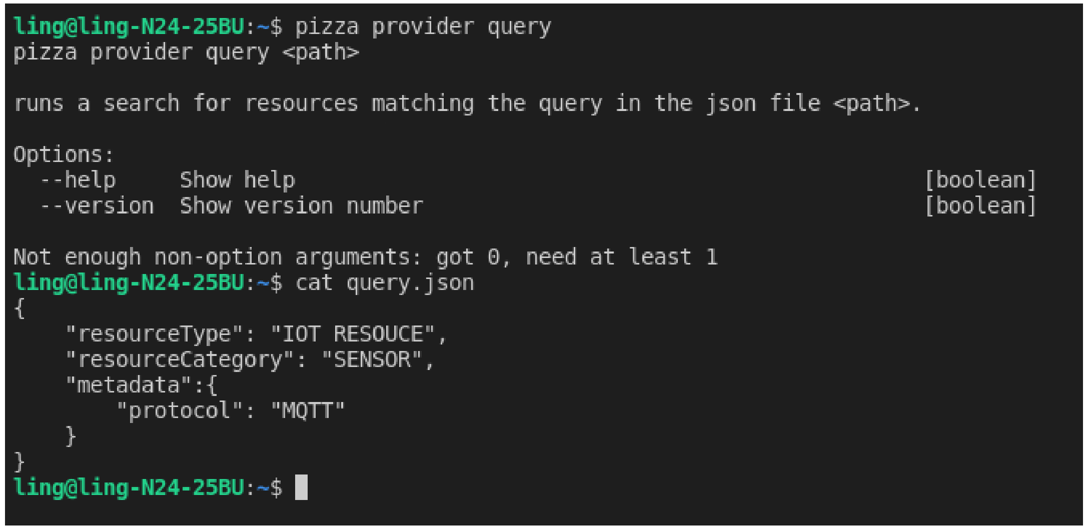
```
ling@ling-N24-25BU:~$ pizza slice create
pizza slice create <file>

creates a slice specified in <file>

Options:
  --help     Show help                                                        [boolean]
  --version  Show version number                                              [boolean]

Not enough non-option arguments: got 0, need at least 1
ling@ling-N24-25BU:~$
```

Figure 6.5: Using *Pizza* to update a slice from a JSON file



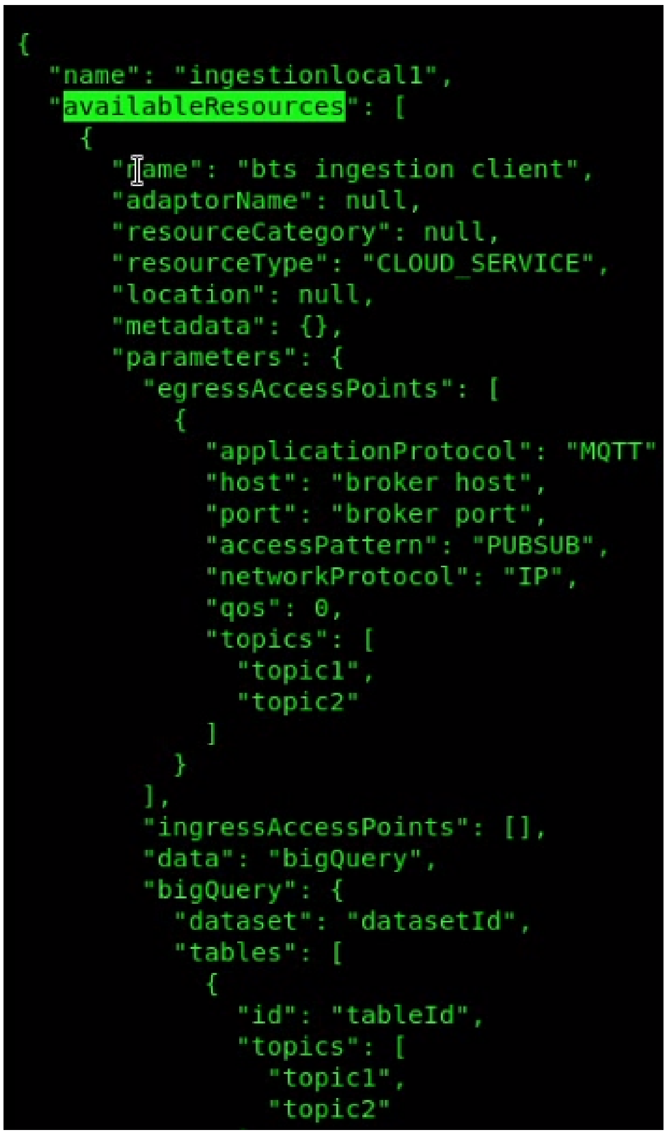
```
ling@ling-N24-25BU:~$ pizza provider query
pizza provider query <path>

runs a search for resources matching the query in the json file <path>.

Options:
  --help     Show help                                                  [boolean]
  --version  Show version number                                        [boolean]

Not enough non-option arguments: got 0, need at least 1
ling@ling-N24-25BU:~$ cat query.json
{
    "resourceType": "IOT RESOUCE",
    "resourceCategory": "SENSOR",
    "metadata":{
        "protocol": "MQTT"
    }
}
ling@ling-N24-25BU:~$ 
```

Figure 6.6: Using *Pizza* to query providers for available resources



```
{
  "name": "ingestionlocal1",
  "availableResources": [
    {
      "name": "bts ingestion client",
      "adaptorName": null,
      "resourceCategory": null,
      "resourceType": "CLOUD_SERVICE",
      "location": null,
      "metadata": {},
      "parameters": {
        "egressAccessPoints": [
          {
            "applicationProtocol": "MQTT"
            "host": "broker host",
            "port": "broker port",
            "accessPattern": "PUBSUB",
            "networkProtocol": "IP",
            "qos": 0,
            "topics": [
              "topic1",
              "topic2"
            ]
          }
        ],
        "ingressAccessPoints": [],
        "data": "bigQuery",
        "bigQuery": {
          "dataset": "datasetId",
          "tables": [
            {
              "id": "tableId",
              "topics": [
                "topic1",
                "topic2"
```

Figure 6.7: Result of the query (Figure 6.6) for available resources

6.2.4 Evaluation

6.2.4.1 Dynamic Resource Provisioning

Learnability As our experiments have shown, it is possible to provision new resources in a slice at any point during its lifecycle. The learnability of this feature is difficult. The concept of long and verbose JSON descriptions is not a new idea in industry. This practice can be seen in all of the major cloud providers: GCP[SRR18], and AWS[Ser17]. Moreover, some of more complicated RESTful APIs that exist such as Facebook's V2 graph based API [Fac18] also rely on heavy JSON payloads. Although these JSON formats are generally documented, the majority of support comes from the community through the form of posted questions of blog post tutorials. The lack of clear documentation hurts the learnability factor of our framework.

Operability The process of dynamically provisioning a resource is relatively easy. As we mentioned in Chapter 4, all configurable resource parameters are in the *parameters* section of the resource model. In practice the description can simply be copy and pasted from a resource discovery query into a slice description, brokers for example have no necessary parameters to be provisioned. Although the resource and slice descriptions can be long and verbose, the structured JSON representation means that simple operations can easily be supported by a variety of tools or scripts. In our experiments we make plenty of use of *shortcut.js* scripts to parse and fill our different resource parameters based on our existing slice description, at the same time always leaving room for a manual inspection if necessary.

Attractiveness The framework has certain attractive features for dynamic provisioning that would draw users from existing solutions. The main value of using our framework is the use of a unified information model and API. This prevents the need to learn specialized models and APIs of different providers. For example, MongoDB Atlas[Inc18b] and MLab[mla18] both offer cloud hosted MongoDB databases. Although both products offered have the same functionality, the methods of provisioning them differ.

Flexibility There is a lot of flexibility involved in the dynamic provisioning feature of our framework. As seen in the first experiment, we can choose on-demand the number and types of data sources through a simple query and copy/paste operation. Additionally, in the third experiment we can see that we can entirely rewire and change the nature of our data flow by provisioning a new resource and configuring old ones.

6.2.4.2 Dynamic Resource Configuration

Learnability The learnability of dynamic resource configuration is on par with dynamic resource provisioning. The same knowledge of the information model is a prerequisite to do it effectively. Adopting some sort of documentation for the resource model would facilitate a smaller learning curve.

Operability A dynamic resource configuration is simply changing the required parameters in a resource model. This can be done through automated scripts and other tools.

However, a knowledge of the resource model is essential for this task to be trivial.

Attractiveness The dynamic resource configuration makes the framework very attractive compared to any related tools that we discussed in Chapter 2. Most dynamic configuration tools do not actually handle any configuration but are very powerful configuration stores. Certain implementation libraries will actually effect any reconfiguration by watching the changes to these configuration stores. Our solution however, aims to actually reconfigure a resource. The adaptor design chains[Ins13] a series of more complicated operations behind a reconfiguration. Additionally, The Service Mesh avoids reconfiguration of TCP/IP communication through abstraction.

Flexibility The dynamic configuration feature can provide flexibility in the form of resource re-usability. In some cases an existing resource can simply be reused to suit changing requirements with a dynamic configuration, there might be no need to statically delete the resource and recreate a new one. Dynamic resource configuration also provides a margin of error in deployments, since and configuration errors can be fixed on-demand.

6.2.4.3 Resource Query

Learnability The resource query feature is probably the most easy to learn feature of our framework. The strong resemblance to the type of query used by the popular document based MongoDB[Inc17] will provide a smooth transition for new users. Learnability is therefore a very strong trait of this feature.

Operability Since JSON is one of the most popular data interchange formats in the Web among developers, it not only adds to the learnability but also makes and query very simple. By looking at a resource model (also in JSON) it is very simple to write a query into a JSON file and call the appropriate command to use the JSON query.

Attractiveness The Resource query might not sound as important as the two features discussed above but can be quite attractive. The document based query is a very powerful tool that can explore different metadata that might be of relevance to a user. Although we do not make much use of metadata in the scope of our work, this could prove very interesting in the scope of resource interoperability.

Flexibility The flexibility of JSON translates pretty well to this feature.

6.2.4.4 Third Party Interfacing

Learnability Interfacing with a third party can be a steep learning curve. There exist no implementation guidelines for an Adaptor that interfaces with a provider. We made the conscious choice of not adhering to strict implementation guidelines like the authors of [LNT16]. However, we have a very well defined set of information models and communication protocols that should be adhered to by an Adaptor for communicating with the framework. To be able to implement Provider Adaptors, the developer must first learn the prerequisites models and protocols.

Operability Once the learning curve has been overcome, the actual implementation of provider adaptors becomes almost formula-like. Additionally the fact the Provider Adaptors are stand alone pieces of software, the developer can pick the technologies of his/her choice as long as the models and protocols are adhered to by the Adaptor.

Attractiveness Since this feature of our framework that allows the dynamic provisioning and configuration of various types of resources, this is a very attractive feature of our framework. Thanks to the provider adaptor design pattern we can apply our abstraction to a wide range of third party providers through the unified API and information model.

Flexibility The flexibility of the Provider Adaptor is one of the main reasons we chose to use it in the framework. A provider could be an organization that provides resources through a well defined API. However a provider can also be a GitHub repository with a useful software artifact. In this case the provider adaptor is responsible to deploying this software artifact and exposing its capabilities through our information model.

6.3 Performance Evaluation

6.3.1 Experimental Setting

During the performance evaluation, our test system is deployed on dedicated VMs. The VMs give us the advantage of better monitoring since we are available to install monitoring agents on the machines to gather metrics such as CPU and memory usage through Google Stackdriver. Additionally, when a service is deployed on a VM we can be sure of its specifications as long as it is the only service deployed on that machine. During a containerized deployment however, a container is scheduled among the different nodes the container cluster and therefore it is hard for use to guarantee a certain set of specifications for the service (even when we declare usage limits). The specifcations of the VMs stated in Table 6.3 can be found on the Google Cloud Platform Compute Engine website at `https://cloud.google.com/compute/docs/machine-types`.

Since our Slice Management Client is implemented as a command line interface, we need separate environment for each user. We cannot concurrently execute the commands of the client in the same operating system since the local data store used in its implementation is not designed to be threadsafe. In order simulate users, we package the Slice Management Client into a docker image. During the experiment we will deploy groups of Slice Management Clients through the Google Cloud Platform Kubernetes Engine. Each container will generate slices specifications to provision. The response time for completed operations will be measured In order to collect test metrics, we deploy a master service which records receive results through HTTP from the simulated user clients. Each container executes one slice creation script and exits. We use the restart policy of Kubernetes to keep providing more users into the system while maintaining a maximum user limit for the current run. We will process the results and gather metrics with a separate analysis script.

In early performance tests, we used MongoDB and RabbitMQ through their respective cloud providers. However, we can only select a few options due to the lack of funding for these platforms. MongoDB Atlas [Inc18b] only allows a maximum of 512MB of storage and while the RAM is not specified, we started receiving connection errors in the middle of testing. CloudAMQP [84c18a] restricts the number of connections and RAM available to free-tier brokers.

Name	**Deployment**	**Type**	**Specs**
Global	Google Cloud	Compute	n1-standard-2
Local	Google Cloud	Compute	n1-standard-2
MongoDB	GCP	Compute	n1-standard-2
RabbitMQ	GCP	Compute	n1-standard-2
Pizza	N/A	Container	0.1x2.0 Ghz CPU, 100Mi RAM
Mosquitto Adaptor	Google Cloud	Compute	n1-standard-1
Analytics Adaptor	Google Cloud	Compute	n1-standard-1
Sensor Adaptor	Google Cloud	Compute	0.n1-standard-1
BigQuery Adaptor	Google Cloud	Compute	n1-standard-1
BigQuery Provider	Google BigQuery	N/A	N/A
Mosquitto Provider	Google Cloud	Compute	n1-standard-1
Analytics Provider	Google Cloud	Compute	n1-standard-1
Sensor Provider	Google Cloud	Compute	n1-standard-1
BTS Sensor	Google Cloud	Container	0.1x2.0 Ghz CPU, 100Mi RAM
Mosquitto Broker	Google Cloud	Container	0.1x2.0 Ghz CPU, 100Mi RAM
Analytics Client	Google Cloud	Container	0.5x2.0 Ghz CPU, 256Mi RAM
BigQuery Dataset	Google BigQuery	N/A	N/A

Table 6.3: Experimental setting for performance evaluations

Each user effects a set number of *test runs*. A test run consists of a set of operations that a users executes with our deployment framework. We run our experiments with an increasing number of users, we gather the set of metrics that we describe in Table 6.4

Metric Name	**Description**
Average Response Time	The average time *in seconds* that passed between requests and responses with our framework. If a test run involves multiple requests to our framework, we measure the average of all those requests. This metric answers the question "How fast does our framework respond?"
Max Response Time	The longest time *in seconds* that occurred between a request and response with our framework. This metric answers the question "What is the slowest time that our framework can respond?"
Min Response Time	The shortest time *in seconds* that occurred between a request and response with our framework. This metric answers the question "What is the fastest time that our framework can respond?"
Deviation	This is the standard deviation of the all the response times from the requests made to our framework. This metric answers the question "How much does the response time of our framework vary?"
Success Rate	The percentage of successful responses between that are executed in all of our tests runs. This metric answers the question "How many requests are answered?"
Users	The number of users constantly using the framework, conducting test runs.
Runs	The number of test runs that each user executes.

Table 6.4: Metrics used for performance experiments

6.3.2 Experiment 1 - Stress Test

This experiment tests the ability of our framework to handle performance at unexpected high loads. We simulate these high loads by using concurrent users that send continuous requests for slice creation to our framework. By continuously creating extra slices, (which also means extra resources) we burden the framework on purpose in order to find the breaking point of our services.

However, the idea of a user that continuously creates resource slices is not expected standard behavior since we expect a user to create at most a few slices and use them. In an ideal situation a good user will also delete his/her slices and release the resources from our framework. The result of the initial set of tests can be seen in Table 6.5.

One test run consists of a slice creation operation. The simulated users create one slice per test run of specification: 2 BTS sensors, 1 MQTT Broker, 1 BigQuery dataset and 1 analytics client. An illustration of the slice we use in this experiment is presented in Figure 6.8. We calculate our metrics by running 25 runs per simulated user.

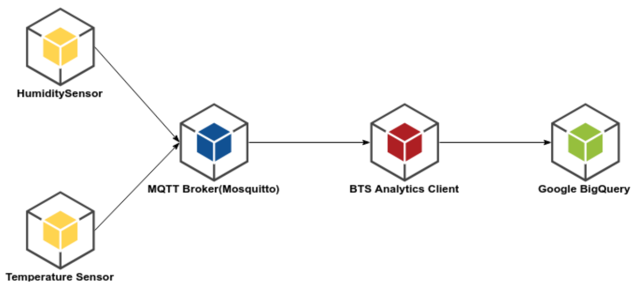


Figure 6.8: An illustration of the resource slice deployed during experiment 1

users	avg rT	deviation	max rT	min rT	success
1	8.04	1.2	16.26	7.19	100
3	15.48	5.84	25.14	7.26	86.71
5	25.07	91.76	604.87	7.59	62.80
10	114.25	215.51	605.91	8.04	59.67
15	115.22	204.04	606.84	5.65	54.81
30	613.11	2.99	619.88	607.52	26.25

Table 6.5: Results of stress test, first attempt
users: Total number of concurrent users
avg rT: Average response time (seconds)
deviation: Standard deviation (seconds) of response times
max rT: Maximum response time (seconds)
min rT: Minimum response time (seconds)
success: % of successful requests

users	avg rT	deviation	max rT	min rT	success
1	6.8	1	13.26	5.86	100
3	12.65	4.72	20.6	5.55	96.87
5	11.33	2.07	24.55	9.18	97.64
10	25.15	8.39	43.14	8.69	86.82
15	39.71	15.2	84.86	12.63	89.78
20	111.07	29.27	175.8	33.5	88.69
30	185.11	58.29	345.28	68.11	53.57

Table 6.6: Results of stress test, second round
users: Total number of concurrent users
avg rT: Average response time (seconds)
deviation: Standard deviation (seconds) of response times
max rT: Maximum response time (seconds)
min rT: Minimum response time (seconds)
success: % of successful requests

We can observe that these results of the first round (Table 6.5) seem under-performing. While our implemented solution might not be very performant, this kind of performance is scarcely seen even in the worst of implementations. After an investigation we noticed a problem with the message consumption between the Local Management Service and its provider adaptors. The Spring AMQP framework that we used in the Local Management Service implementation provides and easy to use abstraction of the RabbitMQ API. However, this abstraction hides certain details in the operations. One such operation *convertSendAndReceive* is responsible for sending a message and waiting for a correct reply. The destination is declared by the programmer, but the receiving is implicitly handled by Spring AMQP. Spring AMQP declares a single reply queue that is create in the default exchange where all messages sent using this method is received. Each message is tagged with a *correlationId* which allows the consumers receiving the reply to identify which reply is destined for which request.

It must first stated that this is the reccommended approach by RabbitMQ for receiving message replies in a request/response manner [Piv12]. Spring AMQP consumes the replies using a pool of consumers that are multi-threaded. This method saves memory for the broker as only one queue needs to be declared throughout the lifecycle of our framework. This method is detrimental to our framework since we would like to use as many consumers as possible to handle replies. We found that a Spring AMQP can be configured to create callback message queues instead of using a single reply queue. This means that for each request a separate reply queue would be created as well as a dedicated consumer to consume from that queue. RabbitMQ discourages this technique in their own documentation. However, of the developers of RabbitMQ even suggested this workaround when questions were posted on RabbitMQ's development forums concerning this performance issue `https://groups.google.com/forum/#!topic/rabbitmq-users/CY6lshL1plA`. After another set of testing, we found the results to be greatly superior. The second set of tests can be seen in Table 6.6.

We can see from Figure 6.9 that the performance of our core services: the Global and Local Management Services do quite well in terms of resource consumption during the high load test runs. One significant problem is very apparent from the results from Table 6.6, the high rate of request failure at 30 users. From the performance of our core services, we see no immediate cause for the high request failure of our framework.

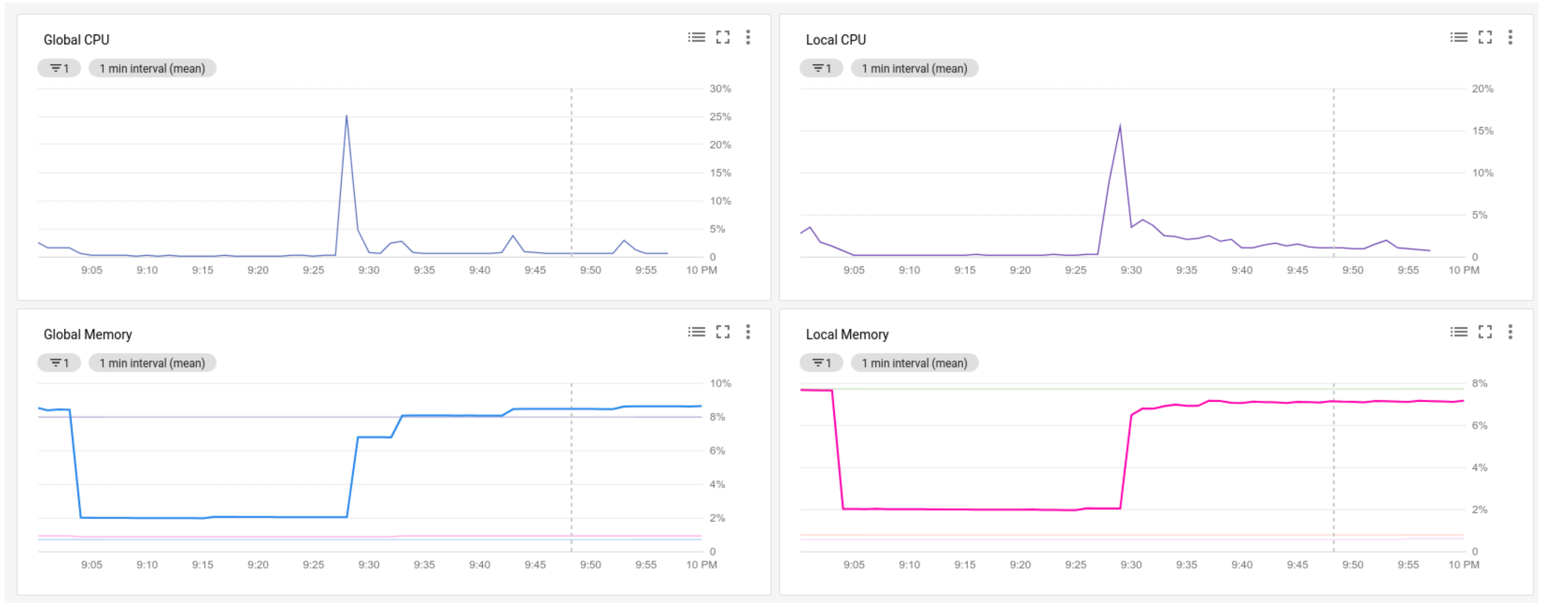


Figure 6.9: Resource consumption of Global and Local Management Services with 30 users at stressed usage patterns

6.3.3 Experiment 2 - Sustained Load

This experiments evaluates the performance of our framework under normal expected usage. Using our set of simulated user clients, one test run consists of a simple series of operations that emulate a correct usage of our framework:

- Create a resource slice
- Wait 10 seconds
- Update a resource slice
- Wait 10 seconds
- Delete a resource slice

The initial slice is identical to the slice used in the previous experiment: 2 BTS sensors, 1 MQTT Broker, 1 BigQuery dataset and 1 analytics client, illustrated in Figure 6.8. The slice is then updated to change the sensors for other sensors of a different data type. The slice is finally deleted from our framework. We calculate our metrics by running 25 runs per simulated user.

We run these tests on increasing numbers of concurrent users. The results can be seen in Table 6.7

users	**avg rT**	**deviation**	**max rT**	**min rT**	**success**
1	8.04	1.2	16.26	7.19	100
3	9.22	1.31	18.07	8.15	100
5	9.61	1.57	19.8	8.04	100
10	16.13	5.15	26.86	5.99	100
15	21.78	7.51	42.49	6.7	100
20	54.02	14.16	75.97	14.78	96
30	71.51	11.13	91.88	40.4	63.09

Table 6.7: Results of sustained load test
users: Total number of concurrent users
avg rT: Average response time (seconds)
deviation: Standard deviation (seconds) of response times
max rT: Maximum response time (seconds)
min rT: Minimum response time (seconds)
success: % of successful requests

We can see that the success rate of operations is far greater compared to our previous experiment. In general we can see that until 20-30 users the framework is quite reliable. Similar to the previous experiment, we see a very high rate of failure when we reach 30 concurrent users. However, Figure 6.9 shows that the resource usage during the 30 user test had the highest rate of failure that both Global and Local Management Services are performing quite well. This was similar to the previous experiment.

The most probable cause of the request failure is a timeout associated with a resource creation request. When creating a slice we treat the resource creation atomically. When one resource fails to be provisioned, then the operation for the entire slice fails too. Since our the performance of our framework is not strained, we analyze our resource providers in the next subsection for possible causes of the high request failure rate.

6.3.4 Resource Provider Analysis

According to our performance tests, we reach a high level of request failure with increasing concurrent users. Other related tools such as Consul [Has18], Zookeeper [Apa18b] and [Cor18] (Chapter 2) can operate at greater than 200 concurrent users according to a benchmark run by CoreOS in 2017 [Lee17]. However, these tools only partly share our feature base and they are also production ready tools. We use their benchmarks as eventual goals, but it is infeasible to try to achieve the goals of production ready products developed by large teams of developers. We indicated in Sections 6.3.2 and 6.3.3 during our experiments that we suspected resource providers as the bottleneck.

To find the problem, we investigated the log files of the different resource providers that we used in our experiment. Since we have treated the resource providers as external to our system, we did not include any kind of instrumentation in their implementations. We added additional instrumentation to resource providers to record the time it takes to create a containerized deployment on GCE [Pla14]. The additional instrumentation was applied to the BTS sensor provider and the MQTT provider, which are two providers implemented by us. We also instrumented the Provider Adaptors of the resource providers. In the Provider Adaptors we measure the time that it takes for the resource to complete all pending configuration after provisioning. Pending configuration in case of the MQTT resource means the assignment of its public IP through the provider (and subsequently GCE) and the update of its resource model (for example, setting unique id, adding metadata). However pending configuration of the sensor resource is the update of its resource model. We ran the scenario of Experiment 1 (Section 6.3.2) with 30 users with the two instrumented resource providers and Provider Adaptors. The measurements that we recorded during the test are presented in Tables 6.8 and 6.9.

Resource Adaptor	**average**	**max**	**min**	**deviation**
MQTT broker	0.8	0.83	0.69	0.13
Sensor	0.75	3.42	0.63	0.34

Table 6.8: Time to provision containerized deployments on GCE, in seconds

Resource Adaptor	**average**	**max**	**min**	**deviation**
MQTT broker	187.58	311	44	90.21
Sensor	0.75	3.42	0.63	0.34

Table 6.9: Time for resource to complete pending configuration, in seconds

We identified a problem associated with the MQTT Provider. Our implementation of the MQTT provider uses GCE Kubernetes [Pla14] to deploy containerized versions of the Eclipse Mosquitto MQTT broker [Fou12]. In order to obtain an MQTT broker as a fully functional resource, it must have a publicly exposed IP address so that clients can connect to it. Using the GCE deployment in order to obtain a public IP for a containerized deployment, it is necessary to expose it with a *LoadBalancer* service [Pla18a]. Creating this service is not a fast operation due to the latency in the Compute Engine's APIs for creating the components of a load balanced service. The latency caused by these tasks are our of our control since we cannot have any measure of control over Google's infrastructure. Therefore there is no way to avoid the wait for the assignment of a public IP to a broker.

We notice that creation of the containerized deployment between the resources is similar and fast. However there is a significant increase between the time the BTS sensor resource and MQTT broker resource to complete their pending configurations. This increase in time is due to the assignment of public IP to the MQTT resource. We find the problem to be the creation of *LoadBalancer* components in the GCE cluster. In scenarios where we use less concurrent users, the creation of *LoadBalancer* components is not an issue. However at 30 users the number *LoadBalancer* components that are created is larger than the number of machines that make up the cluster. It is important to note that in all of GCE's related documentation [Pla18a][SRR18], it is reccommended to use only as many *LoadBalancers* as needed to expose a running. However the documentation considers using GCE to deploy production systems. In our experiments we use GCE as a testbed to deploy resources for many resource ensembles.

We run same test again with 30 concurrent users. However we use CloudMQTT [84c18b] as the provider of MQTT broker resources. Since we do not handle to deployment of CloudMQTT brokers, we only instrument the Provider Adaptor to measure the time for the resource to enter our framework. The results are presented in Tables 6.10 and 6.11

Resource Adaptor	**average**	**max**	**min**	**deviation**
MQTT broker	187.58	311	44	90.21
CloudMQTT broker	1.57	1.94	1.1	0.21

Table 6.10: Time for resource to enter the framework, in seconds

users	**avg rT**	**deviation**	**max rT**	**min rT**	**success**
30	70.85	11.56	90.88	38.4	98.09

Table 6.11: Experiment 1, with only 30 concurrent users. CloudMQTT was used as a resource provider

users: Total number of concurrent users
avg rT: Average response time (seconds)
deviation: Standard deviation (seconds) of response times
max rT: Maximum response time (seconds)
min rT: Minimum response time (seconds)
success: % of successful requests

We notice by replacing our MQTT provider with CloudMQTT we achieve a success rate of 98%. We discovered that the Spring AMQP [Piv17] library that we use in our implementation of the Global and Local Management Services sets a default timeout of 180 seconds (3 minutes). Therefore, by using our MQTT provider that deploys to GCE, we very consistently go over this timeout limit. CloudMQTT as an organization who specialize in the deployment of MQTT brokers manage to provision functional brokers withing 1-2 seconds. Therefore, by switching to the more reliable resource provider we increase the performance of our framework. Therefore, we should treat the resource providers as a source of uncertainty. In a production context, we could either increase the timeout of our requests to resource providers, or impose a threshold on the response time of resource providers.

users	**avg rT**	**deviation**	**max rT**	**min rT**	**success**
30	288.94	93.7	449.82	107.57	96.93

Table 6.12: Experiment 1, with only 30 concurrent users. We use our MQTT provider and set the timeout of provider requests to 600 seconds (10 minutes)

users: Total number of concurrent users
avg rT: Average response time (seconds)
deviation: Standard deviation (seconds) of response times
max rT: Maximum response time (seconds)
min rT: Minimum response time (seconds)
success: % of successful requests

We modified our framework and set the timeout to 600 seconds (10 minutes). Then we ran experiment 1 with 30 users, using our MQTT provider. The results are presented in 6.12. We can see that we now achieve a success rate of $\approx 97\%$. However have an average response time more than four times higher than using the Cloud AMQP provider. Figure 6.11 gives an illustration of the comparison of the request success rates between the MQTT provider (with and with the long timeout) and the CloudMQTT provider. We also provide Figure 6.12 as an illustration of the comparison of the average response times.

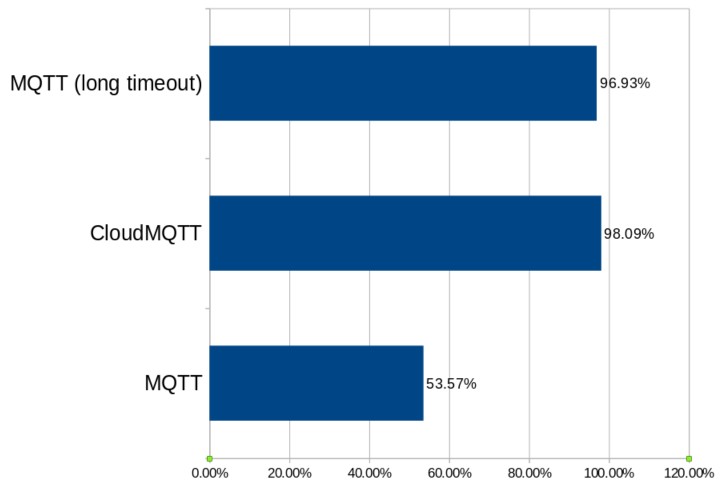


Figure 6.11: A comparison of request success rates (%) for the MQTT and CloudMQTT provider for 30 concurrent users under Experiment 1

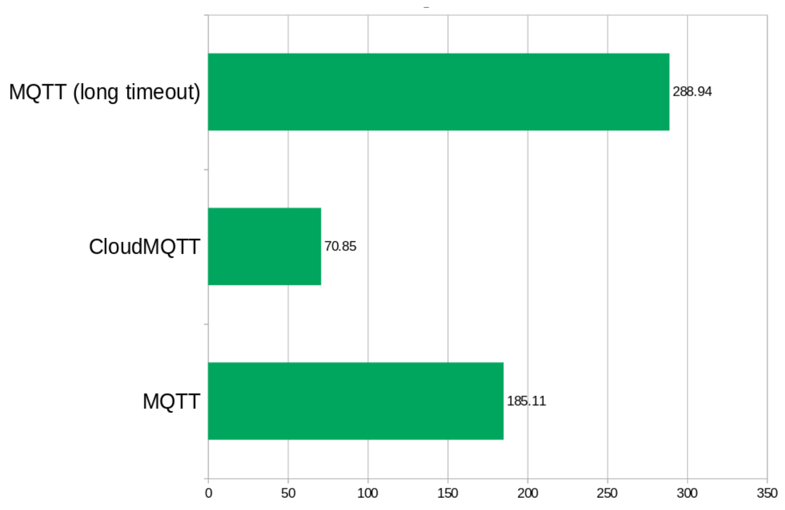


Figure 6.12: A comparison of average response time (in seconds) for the MQTT and CloudMQTT provider for 30 concurrent users Experiment 1

We can see from the graphs that the performance of a resource provider does impact the performance of our framework. In a production scenario, we need to consider a longer response time from our framework if we have underperforming resource providers, or a higher request failure rate.

6.4 Lessons Learned

In this section we give a resumé of important lessons that were learned during and after the course of the evaluation of our framework. The lessons that we learn can be avenues for improvement for our framework. Additionally the lessons learned aim to share useful experience that we gained through the use of different tools and technologies over the course of the work of this thesis.

Our framework has a significant learning curve. During the functional evaluation we found that the long an verbose JSON descriptions of resources and resource slices can a difficult read for a new user to our framework. In our evaluation we use resource slice descriptions of around 200 lines long. Moreover, the documentation of the semantics of our resource information models can be improved. However documentation is also an issue shared among other industry organizations. In our work we used primarily Google Cloud Platform [SRR18] as infrastructure for our deployments and evaluations. We learned that the learnability issue of our framework is shared by the Google Container Engine [Pla14]. GCE models all its types of objects in a JSON format. Similar to the way that we deploy resource slices, to deploy multiple objects through GCE include all the JSON descriptions in one file. The different fields in the JSON descriptions of GCE are not thoroughly documented in either the GCE official documentation [Pla18b] or the official Kubernetes (the engine which runs the GCE) documentation [Fou18b]. The knowledge of using Kubernetes on GCP was essential to our work since, our resource providers and framework deployments all take place on GCE. In practice most of our knowledge was gained from the practical tutorials that were posted in the documentation pages for [Pla18b] and [Fou18b]. Features that go further than the basics in these tutorials are simply not documented. For example, during our implementation of the firewall provider, [Fou18b] under the section "Network Policies" gives an example of a firewall rule that denies all traffic to a selected service. However, there is no documentation on allowing or denying selective traffic, or making a firewall rule apply to a group of services. We learned the two features in the previous sentence simply through experimentation with GCE by creating simply dummy services and sending pings among them. Questions from Stack Over flow such as `https://stackoverflow.com/questions/50960779/kubernetes-ingress-network-policy-from-other-pod` provide examples of selective firewall policies that the official documentation does not even mention. Therefore, with this experience we can conclude that although our JSON information models can provide a barrier for new users, our work is certainly not the only tool that suffers from this issue.

Although in our initial problem statement we consider the resource providers to be external, they can still bring significant consequences. During the performance evaluations of our framework, we noticed the impact of underperforming resource providers that interact with our system. During the evaluations, we did not know the cause of the high rate of error experienced by our framework. In Subsection 6.3.4, we analyzed our resource providers and identified an example of an underperforming provider. The MQTT provider that we implemented, deploys instances of brokers on the Google Container

Engine. However, the time to assign a public IP address to a broker service increased the provisioning time of the resource above the default three minute timeout of our framework. The underlying issue with the long delay in the assignment of public IP addresses lies in the internal processes of Google Container Engine to create the different components necessary for exposing a public IP. We found two solutions to improve the level of service of our framework. The first solution involved using a different provider for the same resource type, in this case we used CloudMQTT [84c18b] as a resource provider. We found by using an alternative provider that is more performant, we could increase the rate of success of requests from $\approx 50\%$ to $\approx 97\%$. The second solution involved increasing the request timeout of our framework to 10 minutes. By increasing the timeout we achieved a similar success ratio our first solution, but the average response time of our framework increased to ≈ 3 minutes. We have learned after this experience that the performance of a resource provider can have consequences to the performance of our framework. We could imagine that in a production context, we would face the choice of either using a small, but efficient, set of resource providers or using a large pool of resource providers but passing operational delays to our end users for underperforming resource providers.

We can see above, that we mentioned Google Cloud Platform and Google Container Engine several times to reference issues or features. This indicates that there is strong reliance on Google Cloud Platform in our work. This is true, since all our framework deployments and most of our resource providers use the infrastructure of GCP.This work has benefited from the Google Cloud Platform Education Grant (TU Wien, Advanced Services Engineering) for access to resources such as VMs and container clusters. Our work should benefit from using infrastructures from other cloud providers such as Amazon Web Services [Ser17]. However, due to the lack of funding and resources, this was simply not possible. We did initially try to use MongoDB Atlas [Inc18b] and CloudAMQP [84c18a] free tier services for the deployments in our evaluations. These free tier services were simply not powerful enough for our requirements, for example both MongoDB Atlas and CloudAMQP apply connection limits to their resources, after which the service stop functioning. We subsequently deployed these resources on powerful VMs in GCP. Any further work on our framework could benefit from using other kinds of infrastructure.

In the worst case scenario that involves underperforming resource providers and a high number of concurrent users, a user could wait several minutes for the creation of a resource ensemble. This magnitude of response time would not suit users who use our framework to create short-lived resource ensembles. The time it takes for our framework to respond might not be worth the reward of using he resource ensemble for a short a period of time. However, for long-lived resource ensembles, the performance of our tool can still be adequate. We the maximum response time from our performance evaluation in Table 6.12, which is 450 seconds ($\approx$ 8minutes). If the user intends to use the resource ensemble for only 10 minutes, we can see that the usage period could potentially take as long as the time it takes to provision the ensemble. However with increasing lifespan of the resource ensemble the upper bound of our framework's response time is amortized.

A 30 minute usage of an ensemble means the response time only accounts for 25% of the usage period, for one hour the proportion decreases to 12.5%, for two hours this proportion is 6.25% and so on.

The performance impact of the resource providers could lead to further work on our framework. Currently the framework enables users to interface with diverse resource providers. However, the user cannot know from our framework the performance of a resource provider from our framework. Using our example from Subsection 6.3.4, we assume a case where the MQTT and CloudMQTT provider are deployed together with our framework. The user depending on his/her choice, could either deploy his resource ensemble within two minutes or 10 minutes according to Tables 6.11 and 6.12. Future work could focus on a reccomendation service for resource providers. The example provide 6.3.4 shows that even between two providers that provide the same resource type, it is possible to distinguish attributes that make one provider more attractive than the other. In this case the CloudMQTT provider would be more desirable than our MQTT provider since the time to provision the resource is shorter. This concept could be extended to different attributes that do not only focus on performance but on other attributes of the resource provider. The correct provider reccomendation to an end user could improve the quality of our framework's experience.

6.5 Summary

In this section we evaluated our proposed framework on two levels: Functional and Performance. We found that functionally we achieved the feature base required to achieve the use cases of our motivating scenario specified in Chapter 3. We evaluated the features of our framework on four different criteria: Learnability, Operability, Attractiveness and Flexibility. We concluded that the features of our framework provides flexibility and operability in the management of resource ensembles compared to other related works and tools. However, the current command line based Slice Management Client combined with a verbose JSON specification means that our framework is not friendly for new users even though the JSON format simplifies more complex operations by facilitating automation.

Our performance evaluation revealed that the performance of resource providers can have a significant impact on the performance of our framework. We identified an example where using an alternative resource provider decreased the average response time from $\approx$ 180s to $\approx$ 70s. The reason behind this improvement was linked to our cloud infrastructure provider, Google Cloud Platform [SRR18]. This nevertheless, shows that our external resource providers are a source of uncertainty. This uncertainty can lead to future work on our framework by trying to reduce this uncertainty by either introducing a benchmarks for resource providers or a recommendation service to users of our framework.

We conclude the chapter with a section that notes several lessons learned from the evaluation of our framework. We discuss issues such as those related to Google Cloud

Platform, which provides the infrastructure for our work, or the performance impact of resource providers on our framework.

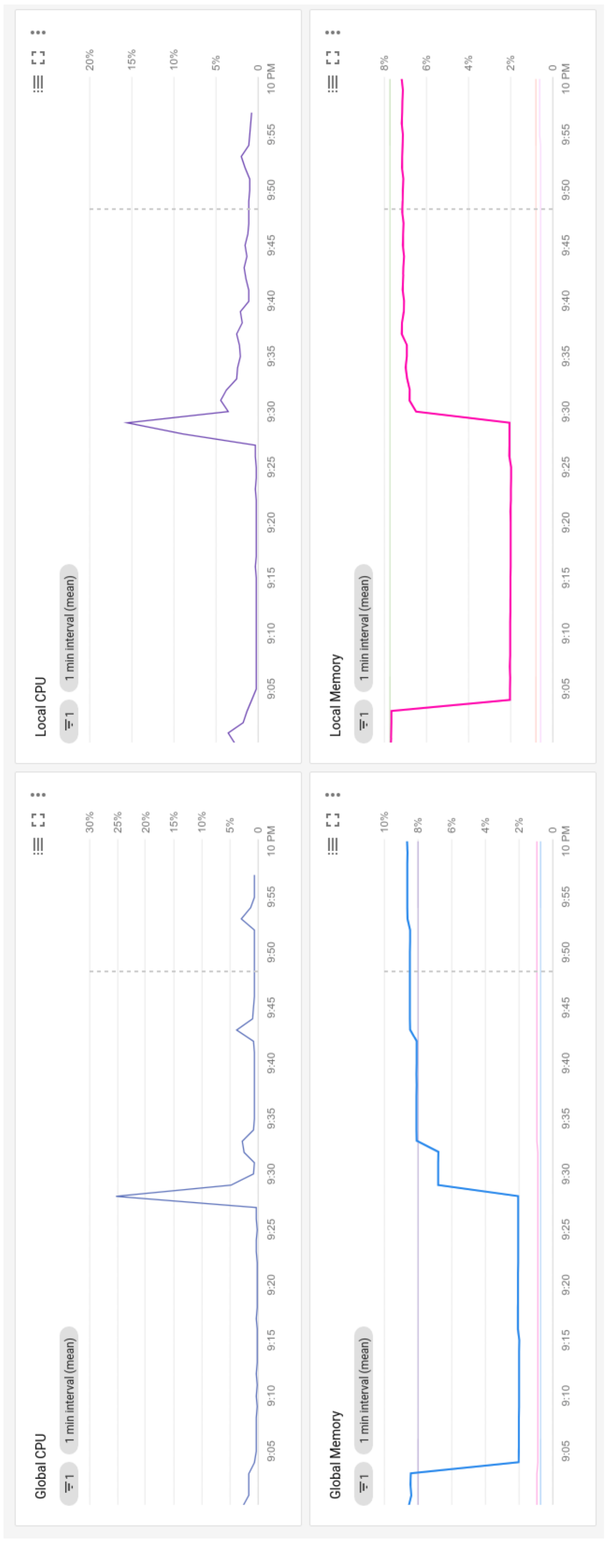


Figure 6.10: Resource consumption of Global and Local Management Services with 30 users at expected usage patterns

CHAPTER 7

Conclusion and Future Work

7.1 Conclusion

In this thesis we introduced a novel framework that is able to provision and configure resource ensembles of IoT resources, network function services and cloud services. We had difficulty finding any related works or tools that were a solution to our problem statement, although there are other solutions that handle different subsets of our problem statement. To this end, we have leveraged exiting work to create a distributed information model that harmonizes data between a large heterogeneous pool of resources and resource providers. Our harmonized information models clear the way for an architectural design that exposes a single unified API which can query and control resources and providers across different platforms and infrastructures. By offering a single unified API for resource based operations, we reduce the knowledgebase that is required to provision and configure a resource ensemble that forms an IoT cloud system. Furthermore, we use the unified API that we provide in order to develop features that allow for the provisioning and configuration of end-to-end resource slices on-demand at runtime. However, we have designed our framework in such a way that the management of resource slices do not depend solely on our implementation. In order to facilitate interoperability we have exposed our resource level operations through a HTTP REST API that can be used by third parties.

Regarding the evaluation of our prototype implementation of our proposed framework, we contribute evaluations on the functional and performance levels. The design of our framework was driven by a scenario that is based on a real world problem. In the functional evaluation we use our framework to achieve the use cases of our stated scenario. Moreover, we evaluate the features of our prototype based on a set of criteria: Learnability, Simplicity, Attractiveness and Flexibility. The evaluation against these criteria take into account the processes and methods of related tools. Regarding the performance evaluation, we tested the framework with a small amount of concurrent users, so our results are adequate but not extensive. We managed to run experiments that evaluated the stress and load that our prototype can handle. Running a basic deployment and interfacing with seven different resource providers, our framework can handle the stress of continuous resource slice creation requests for up to 20 users with an operation success rate of over 88%. However, 20 users is simply too small of a metric for performance testing to give reliable results. Furthermore, we evaluated that our prototype can handle a sustained load of reasonable usage for up to 20 users with a operation success rate over 96%. We also discovered that the performance of resource providers, which are external to our framework, can have significant impacts on the performance of our framework. We found an example where using bad and good quality resource providers with our framework resulted in a difference of 40% in the average response time. The prototype implementation of our framework is available open source on GitHub at `https://github.com/SINCConcept/HINC`. Additionally, all the IoT resource, network function service and cloud service examples that we have implemented to test and evaluate our work is also available open source on GitHub a `https://github.com/rdsea/IoTCloudSamples` and can be used by developers who need example software for testing. Some of our work is being integrated with the INTER-IoT project, and currently our tools are being used to develop interoperability support for resource ensembles. We also contributed a paper published by the European Conference on Software Architectures (ECSA 2018) in the program of Posters, Tools and Demos Track [HT18].

7.2 Future Work

Due to the scope of work and time we could only evaluate our prototype framework with one scenario that could benefit from the resource ensemble approach. With this limit in mind, we aimed to thoroughly evaluate all the functional aspects of our prototype through our chosen scenario. Therefore in future works our proposed framework could be applied to different scenarios that involve IoT and cloud resources working together to gain more knowledge into the feasibility of the resource ensemble based approach. We also learned form our evaluation that the command line terminal based approach can be verbose for constructing and reading the description of resource slices. However, with the availability of our resource REST API, we welcome any future work on a more graphical oriented user friendly solution for the Slice Management Client which could involve a native desktop or mobile client which could repackage some library code in our slice management client into a more user friendly experience. This could be somewhat like the approach taken by Node-RED[Fou18a] which allows the import and export of data flows through JSON in its graphical interface. We also believe that the performance of our prototype can be improved through further iterations of development although our performance results were satisfactory. We have separated clearly our architectural design and prototype implementation so that future researchers or students can either reuse and extend our implementation or seek a better implementation while keeping to our conceptual architectural design.

Bibliography

[84c18a] 84codes. Cloud amqp product overview. `https://www.cloudamqp.com/docs/product_overview.html`, 2018.

[84c18b] 84codes. Cloud mqtt. `https://www.cloudmqtt.com/`, 2018.

[ACIM15] M. Amadeo, C. Campolo, A. Iera, and A. Molinaro. Information centric networking in iot scenarios: The case of a smart home. In *2015 IEEE International Conference on Communications (ICC)*, 2015.

[Ala18] Alarmtab. Bts digital temperature sensor. `https://www.alarmtab.de/BTS-digital-temperature-sensor`, 2018.

[Ama18] Amazon. Amazon rds user guide. `https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html`, 2018.

[Apa18a] Apache. Cassandra nosql database. `http://cassandra.apache.org/`, 2018.

[Apa18b] Apache. Zookeeper. `https://zookeeper.apache.org/`, 2018.

[AS17] IBM Animesh Singh. What is a service mesh and how istio fits in. `https://developer.ibm.com/code/2017/07/21/service-mesh-architecture-and-istio/`, 2017.

[Bud18] Buddy. Buddy works. `https://buddy.works/`, 2018.

[Buo18] Buoyant.io. Linkerd service mesh proxy. `https://linkerd.io/`, 2018.

[Che18] Chef. Chef. `https://www.chef.io/chef/`, 2018.

[Cor18] CoreOS. etcd. `https://coreos.com/etcd/`, 2018.

[CZC11] Marco Crasso, Alejandro Zunino, and Marcelo Campo. A survey of approaches to web service discovery in service-oriented architectures. *J. Database Manage.*, 2011.

[Dat18] Treasure Data. fluentbit. `https://fluentbit.io/`, 2018.

[DCB15] S. K. Datta, R. P. F. Da Costa, and C. Bonnet. Resource discovery in internet of things: Current trends and future standardization aspects. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, 2015.

[DCLT+14] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, Jakub Zwolakowski, Antoine Eiche, and Alexis Agahi. Automated synthesis and deployment of cloud applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, 2014.

[dD18] databroker DAO. databroker dao global market. https://databrokerdao.com/, 2018.

[DS16] Pablo Delatorre and Alberto Salguero. Training to capture software requirements by role playing. In *Proceedings of the Fourth International Conference on Technological Ecosystems for Enhancing Multiculturality*, 2016.

[Fac18] Facebook. Facebook v2 graph api. https://developers.facebook.com/docs/graph-api/overview, 2018.

[FEH+14] Kaniz Fatema, Vincent C. Emeakaroha, Philip D. Healy, John P. Morrison, and Theo Lynn. A survey of cloud monitoring tools: Taxonomy, capabilities and objectives. *Journal of Parallel and Distributed Computing*, 2014.

[Fou12] Eclipse Foundation. Eclipse mosquitto™ an open source mqtt broker. https://mosquitto.org, 2012.

[Fou18a] JS Foundation. Node-red 0.18 release. https://nodered.org/blog/2018/01/31/version-0-18-released, 2018.

[Fou18b] Linux Foundation. Kubernetes reference documentation. https://kubernetes.io/docs/reference/, 2018.

[FS18] NGINX Floyd Smith. What is a service mesh? https://www.nginx.com/blog/what-is-a-service-mesh/, 2018.

[GGG+16] Maurizio Gabbrielli, Saverio Giallorenzo, Claudio Guidi, Jacopo Mauro, and Fabrizio Montesi. Self-Reconfiguring Microservices. In *Theory and Practice of Formal Methods*. 2016.

[Git18a] Github. Statsd. https://github.com/etsy/statsd, 2018.

[Git18b] GitHub. yargs the modern, pirate-themed successor to optimist. https://github.com/yargs/yargs, 2018.

[Goo18a] Google. Bigtable nosql database. https://cloud.google.com/bigtable/, 2018.

[Goo18b] Google. What is bigquery? https://cloud.google.com/bigquery/what-is-bigquery, 2018.

[Has18] HashiCorp. Consul. https://www.consul.io/, 2018.

[HT18] Lingfan Gao H.L. Truong, Michael Hammerer. Service architectures and dynamic solutions for interoperability of iot, network functions and cloud resources. In *In Proceedings of European Conference on Software Architecture (ECSA2018)*, 2018.

[II18a] INTER-IoT. Inter-iot eu project. http://www.inter-iot-project.eu/, 2018.

[II18b] INTER-IoT. Inter-iot eu project open calls. http://www.inter-iot-project.eu/open-call, 2018.

[Inc17] MongoDB Inc. Mongodb. https://www.mongodb.com/, 2017.

[Inc18a] Kong Inc. Kong api gateway. https://konghq.com, 2018.

[Inc18b] MongoDB Inc. Mongodb atlas documentation. https://docs.atlas.mongodb.com/, 2018.

[Inf18] InfluxData. Telegraf. https://www.influxdata.com/products/, 2018.

[Ins13] The European Telecommunications Standards Institute. Network functions virtualisation update white pap er," https://portal. https://portal.etsi.org/NFV/NFVWhitePaper2.pdf, 2013.

[ISO10] ISO/IEC. Iso/iec 25010 system and software quality models. Technical report, 2010.

[JAAA16] M. A. Jamil, M. Arif, N. S. A. Abubakar, and A. Ahmad. Software testing techniques: A literature review. In *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, 2016.

[Jen18] Andrew Jenkins. Service mesh architectures. https://aspenmesh.io/2018/03/service-mesh-architectures/, 2018.

[Kle17] Matt Klein. Service mesh data plane vs. control plane. https://blog.envoyproxy.io/service-mesh-data-plane-vs-control-plane-2774e720f7fc, 2017.

[KR15] Andreas Kliem and Thomas Renner. *Towards On-Demand Resource Provisioning for IoT Environments.* 2015.

[Lee17] Gyu-Ho Lee. Exploring performance of etcd, zookeeper and consul. https://coreos.com/blog/performance-of-etcd.html, 2017.

[LNT16] D. H. Le, N. Narendra, and H. L. Truong. Hinc - harmonizing diverse resource information across iot, network functions, and clouds. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*, 2016.

[MC12] Debajyoti Mukhopadhyay and Archana Chougule. A survey on web service discovery approaches. In David C. Wyld, Jan Zizka, and Dhinaharan Nagamalai, editors, *Advances in Computer Science, Engineering & Applications*, 2012.

[MCTD13] D. Moldovan, G. Copil, H. Truong, and S. Dustdar. Mela: Monitoring and analyzing elasticity of cloud services. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, 2013.

[mla18] mlab. Mlab mongodb hosting. `https://mlab.com/`, 2018.

[MS05] Jose Luis Mate and Andres Silva. *Requirements Engineering for Sociotechnical Systems*. Information Resources Press, 2005.

[NVI$^+$15] S. Nastic, M. Vögler, C. Inzinger, H. Truong, and S. Dustdar. rtgovops: A runtime framework for governance in large-scale software-defined iot cloud systems. In *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, 2015.

[OSS18a] OSS. Jenkins. `https://jenkins.io/`, 2018.

[OSS18b] Netflix OSS. Spinnaker concepts. `https://www.spinnaker.io/concepts/`, 2018.

[Per17] Charith Perera. Sensing as a service (s2aas): Buying and selling iot data. 2017.

[Piv12] Pivotal. Remote procedure call (rpc) with rabbitmq. `https://www.rabbitmq.com/tutorials/tutorial-six-spring-amqp.html`, 2012.

[Piv17] Pivotal. Spring boot framework. `https://spring.io/projects/spring-boot`, 2017.

[Piv18] Pivotal. Rabbitmq message broker. `https://www.rabbitmq.com/`, 2018.

[Pla14] Google Cloud Platform. Google container engine. `https://cloud.google.com/kubernetes-engine/`, 2014.

[Pla17a] Google Cloud Platform. Google cloud pub sub. `https://cloud.google.com/pubsub/`, 2017.

[Pla17b] Google Cloud Platform. Google stackdriver. https://cloud.google.com/stackdriver/, 2017.

[Pla18a] Google Cloud Platfor. Exposing applications using services. https://cloud.google.com/kubernetes-engine/docs/how-to/exposing-apps, 2018.

[Pla18b] Google Cloud Platform. Google container engine documentation. https://cloud.google.com/kubernetes-engine/docs/, 2018.

[PZCG13] Charith Perera, Arkady B. Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Sensing as a service model for smart cities supported by internet of things. 2013.

[rol] Role-play and use case cards for requirements review.

[See18] Seed. Sensors for grove. https://www.seeedstudio.com/category/Sensor-for-Grove-c-24.html, 2018.

[Ser17] Amazon Web Services. Amazon web services virtual private cloud. https://aws.amazon.com/vpc/, 2017.

[Sim03] Ludwig Simone. Comparison of centralized and decentralized service discovery in a grid environment. file:///home/ling/Downloads/Comparison_of_centralized_and_decentralized_servic.pdf, 2003.

[Sma17] Smartbear. Swagger api documentation. https://swagger.io/, 2017.

[SRR18] V. Srinivasan, J. Ravi, and J. Raj. *Google Cloud Platform for Architects: Design and manage powerful cloud solutions.* 2018.

[TB17] Hong-Linh Truong and Luca Berardinelli. Testing uncertainty of cyber-physical systems in iot cloud infrastructures: Combining model-driven engineering and elastic execution. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Testing Embedded and Cyber-Physical Systems*, 2017.

[TCD$^+$15] H. Truong, G. Copil, S. Dustdar, D. Le, D. Moldovan, and S. Nastic. icomot – a toolset for managing iot cloud systems. In *2015 16th IEEE International Conference on Mobile Data Management*, 2015.

[Tho15] Sonali D. Thosar. Cloud computing and software-based internet of things. *International Journal of Advanced Research in Computer Science and Software Engineering*, 2015.

[TN16] Hong Linh Truong and Nanjangud C. Narendra. SINC - an information-centric approach for end-to-end iot cloud resource provisioning. In *International Conference on Cloud Computing Research and Innovations, ICCCRI 2016, Singapore, Singapore, May 4-5, 2016*, 2016.

[Tru18] H. L. Truong. Towards a resource slice interoperability hub for iot. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, 2018.

[VC17] C.P. Vandana and A. A. Chikkamannur. Study of resource discovery trends in internet of things (iot). *Advanced Networking and Applications*, 2017.

[WM17] buoyant.io William Morgan. What's a service mesh? and why do i need one? `https://blog.buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/`, 2017.

[YZ11] Microsoft Yu Zheng. T-drive trajectory data sample. `https://www.microsoft.com/en-us/research/publication/t-drive-trajectory-data-sample/`, 2011.

[ZT13] Bernd Zwattendorfer and Arne Tauber. The public cloud for e-government. *Int. J. Distrib. Syst. Technol.*, 2013.