Article

A Flexible IoT Stream Processing Architecture based on Microservices

Luca Bixio[1], Giorgio Delzanno[2], Stefano Rebora[1], Matteo Rulli[1]

[1] Flairbit s.r.l, Italy; stefano.rebora@flairbit.io
[2] DIBRIS, University of Genova, Italy, giorgio.delzanno@unige.it
* Correspondence: Giorgio Delzanno, email:giorgio.delzanno@unige.it

Version October 20, 2020 submitted to Journal Not Specified

Abstract: The Internet of Things (IoT) has created new and challenging opportunities for Data Analytics. IoT represents an infinitive source of massive and heterogeneous data, whose real-time processing is an increasingly important issue. Real-time Data Stream Processing is a natural answer for the majority of the goals of IoT platforms, but it has to deal with the highly variable and dynamic IoT environment. IoT applications usually consist of multiple technological layers connecting 'things' to a remote cloud core. These layers are generally grouped in two macro-levels: the edge-level (consisting of the devices at the boundary of the network near the devices that produce the data) and the core-level (consisting of the remote cloud components of the application). Real-time Data Stream Processing has to cope with a wide variety of technologies, devices and requirements that vary depending on the two IoT application levels. The aim of this work is to propose an adaptive microservices architecture for an IoT platform able to integrate real-time stream processing functionalities in a dynamic and flexible way, with the goal of covering the different real-time processing requirements that exist among the different levels of an IoT application. The proposal has been formulated for extending Senseioty, a proprietary IoT platform developed by FlairBit S.r.l., but it can easily be integrated in any other IoT platform. A preliminary prototype has been implemented as proof of concept of the feasibility and benefits of the proposed architecture.

Keywords: Cloud Computing; Service Oriented Computing; Internet of Things; Real-time Stream Processing; Query Languages

---

1. Introduction

Nowadays, with the rise of IoT, we have at our disposal a wide variety of smart devices able to constantly produce large volumes of data at an unprecedented speed. Sensors, smartphones and any other sort of IoT devices are able to measure an incredible range of parameters, such as temperature, position, motion, health indicators an so forth. More and more frequently, the value of these data highly depends on the moment when they are processed and the value diminishes very fast with time: processing them shortly after they are produced becomes a crucial aspect. Indeed, the aim of Real-time Stream Processing is to query continuous data streams in order to extract insights and detect particular conditions as quickly as possible, allowing a timely reaction. Possible examples are the alert generation of a medical device or the real-time monitoring of a production line. In Stream Processing, data are no more considered as static and persistent data stored in a database, but as continuous and potentially unbounded sequences of data elements (i.e. data streams) from which static queries (a.k.a. rules) continuously extract information. The systems that execute this processing phase in a very short time span (milliseconds or seconds) are defined real-time stream processing engines. The IoT world offers an infinite set of use cases where real-time stream processing functionalities can be applied, but IoT applications provide at the same time a heterogeneous environment with respect to requirements, devices and technologies. For these reasons, integrating real-time processing engines in IoT platforms becomes a challenging operation that requires special attention.

An IoT platform provides tools, technologies and capabilities for simplifying the development, provisioning and management of IoT applications. Real-time stream processing engines are an increasingly popular and relevant technology, which the majority of the platforms are integrating in order to provide all the functionalities required by modern IoT applications. Indeed, Real-Time Stream Processing plays a crucial role in different and common IoT application scenarios, for instance: Anomaly and fraud detection; Remote monitoring; Predictive Maintenance; Real-time analytics (Sentiment analysis, Sports analytics, etc.). When integrating real-time stream processing engines in IoT platforms, the main difficulties arise from the high heterogeneity and dynamicity of the requirements and technologies of common IoT applications. At high level, a general IoT application consists of the following layers: The sensors/actuators layer, which includes the IoT devices; The edge layer, which includes all the devices near the sensors/actuators-level. These edge devices usually play the role of gateways, enabling the collection and the transmission of data; The core/cloud layer, which includes all the core functionalities and services of the application; The application/presentation layer, which includes all the client applications that have access to the core functionalities and services. Integrating real-time stream processing capabilities in IoT platforms imposes to face the following three main aspects:

- Twofold level of applicability. It is required often to apply Real-Time Stream Processing at two different levels: at edge level and at core/cloud level. Both approaches offer different benefits but the great difference between the devices and resources at edge level and core level imposes also quite different requirements that affect the choice of the stream processing engines.
- Technological pluralism. Due to the previous point, a natural consequence is to introduce different stream processing engines in the IoT platform because one stream processing technology rarely covers the edge level and the cloud level requirements. Having different stream processing engines means having different processing models and languages that must be handled for implementing stream processing rules.
- Rules' dynamicity. Usually, real-time IoT stream processing rules are based on a dynamic lifecycle. In the majority of IoT use cases, the functionalities implemented by real-time stream processing rules can be temporary functionalities (that are executed on demand and then removed after a while) or long-running functionalities never modified (e.g. a remote monitoring process). Moreover, it is often required to deploy rules directly on edge devices for reducing the response latency time or applying some pre-filtering operations, but when the workload increases, a scalable approach may be more preferable. For all these reasons, rules should have the possibility to be dynamically reallocated on different stream processing engines.

Considering these aspects, the goal of this work is to propose an adaptive solution for integrating real-time stream processing functionalities into an IoT platform, Senseioty by Flairbit [21], able to satisfy the different requirements imposed by the edge level and the cloud/level. Moreover, the proposal offers a dynamic mechanism for facilitating the dynamic management and relocation of stream processing rules, hiding at the same time the complexity introduced by the presence of different and heterogeneous stream processing engines. The innovative aspect of our solution, with respect to common IoT platforms, consists in limiting the expressive power for defining stream processing rules to a predefined set of templates, in favor of a much more flexible and dynamic deployment model. The proposal architecture has been designed following a microservices architectural pattern. Microservices are a natural and widely adopted solution for implementing software platforms. The majority of IoT platforms are based on a microservices approach, even when it is not explicitly mentioned. This happens because the microservices architectural style is a chameleonic style, which can be implemented in different ways. Indeed, several technologies exist for implementing microservices, but for our purposes we have selected a particular technology able to guarantee a significant level of flexibility and dynamicity.

Plan of the paper

In Section 2 we give an overview of the main features of the microservices architectural style and of a particular Java technology named OSGi, the main technology applied in Senseioty, the proprietary IoT platform developed by FlairBit. In Section 3 and 4 we present our proposal and a prototype implemented as a possible extension of Senseioty based on Siddhi and Apache Flink. In Section 5 we address some conclusions and future work.

2. The Microservices Architectural Style and Java OSGi

The microservices architectural style, see e.g. [25], is born to address the problems of the traditional monolithic approach. When you start to design and build a new application, the easiest and most natural approach is to imagine the application as unit composed by several components. The application is logically partitioned in modules and each one represents a functionality, however it is packaged and deployed as a unit. This monolithic approach is very simple and comes naturally. Indeed, all the IDE's are necessarily designed to build a single application and the deployment of a single unit is easy and fast. Also scaling the application is trivial because it requires only running multiple instances of the single unit. This approach initially works and apparently quite well: the question is what happens when the application starts to grow. When the number of functionalities increases and the application becomes bigger and bigger, the monolithic approach shows its natural limit with respect to human capacities. In a short while, the dimensions of the application are such that a single developer is unable to fully understand it and this leads to serious problems. For example, implementing a new functionality becomes harder and time consuming and fixing bug even worse. The whole code is inevitably too complex; therefore adopting new frameworks and technologies is discouraged. In addition, the deployment and the start-up time are obviously negative affected by the huge size of the application. The main consequence is the slowdown of the entire development phase and any attempts of continuous integration and other agile practises fails. Moreover, all the components run in the same process or environment causing serious problems of reliability: a failure or a bug in a single component can compromise the entire application. In few words, the overall complexity of a huge monolith overwhelms the developers. The microservices architectural style was created specifically to address this kind of problems and to tackle the complexity. The book [24] describes a three-dimensional scale model known as scale-cube: Horizontal Scaling (running multiple instances behind a load-balancer); Functional Scaling (decomposing a monolithic application into a set of services, each one implementing a specific set of functionalities); Scaling of Data Partitioning (data are partitioned among the several instances and each copy of the application). These concepts are strongly connected to the idea of Single Responsibility Principle (SRP) [31]. The functionalities are exposed through an interface, often a REST API, and can be consumed by other services increasing the composability. The communication between microservices can be indifferently implemented by synchronous or asynchronous communication protocol and each microservice can be implemented with a different and ad-hoc technology. Moreover, each microservice has its own database rather than sharing a single database schema with other services. This makes a microservice an actual independently deployable and loosely coupled component. In this setting communication is provided via an API Gateway [28].

The API Gateway is similar to the Facade pattern from object-oriented design: it is a software component able to hide and encapsulate the internal system details and architecture, providing a tailored API to the client. It is responsible for handling the client's requests and consequently invoking different microservices using different communication protocol, finally aggregating the results. Microservices architectures often provide a service discovery mechanism typically implemented via a shared registry which is basically a database that contains the network locations of the associated service instances. Two important requirements for a service registry are to be highly available and up to date, thus it often consists in a cluster of servers that use a replication protocol to maintain consistency. One of the main principles at the heart of the microservices architecture is the decentralization of

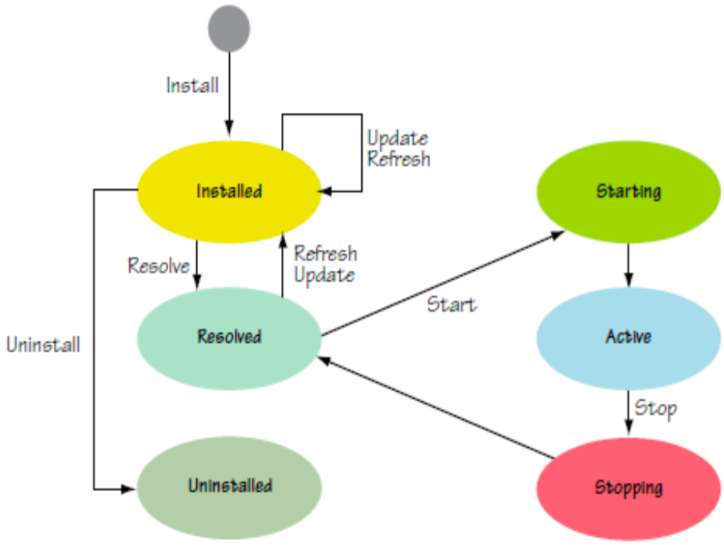

Figure 1. Bundle life cycle.

data management: each microservice encapsulates its own database and data are accessible only by its API. This approach makes microservices loosely coupled, independently deployable and able to evolve independently from each other. In addition, each microservice can adopt different database technologies depending on its specific requirements, for example for some use cases a NoSQL database may be more appropriate than a traditional SQL database or vice versa. Therefore, the resulting architecture often uses a mixture of SQL and NoSQL databases, leading to the so-called polyglot persistence architecture. In this setting, data consistency is often achieved via an event-driven architecture [29]. A message broker is introduced into the system and each microservice publishes an event whenever a business entity is modified. Other microservices subscribe to these events, update their entities and may publish other events in their turn. The event-driven architecture is also a solution for the problem of queries that have to retrieve and aggregate data from multiple microservices. Indeed, some microservices can subscribe to event channels and maintain materialized views that pre-join data owned by multiple microservices. Each time a microservice publishes a new event, the view is updated. The last key aspect of the microservices architecture is how a microservices application is actually deployed. Three main different deployment patterns exist [30]: Multiple Service Instances per Host Pattern; Service Instance per Host Pattern sub-divided in (Service Instance per Virtual Machine Pattern/Container Pattern); Serverless Deployment Pattern (e.g. AWS Lambda; Google Cloud Functions; Azure Functions).

Java OSGi OSGi [14] consists of a set of specifications established by the OSGi Alliance. The OSGi architecture [15] appears as a layered model. The bundles are the modules implemented by the developers. A bundle is basically a standard JAR file enriched by some metadata contained in a manifest [27]. The manifest and its metadata make possible to extend the standard Java access modifiers (public, private, protected, and package private). A bundle can explicitly declare on which external packages it depends and which contained packages are externally visible, meaning that the public classes inside a bundle JAR file are not necessarily externally accessible. The module, life cycle and services layer constitute the core of the OSGi framework: The module layer defines the concept of bundle and how a bundle can import and export code; The life cycle layer provides the API for the execution-time module management; The service layer provides a publish-find-bind model for plain old Java objects implementing services able to connect dynamically the bundles. Finally, the security layer is an optional layer, which provides the infrastructure to deploy and manage applications that must run in fine-grained controlled environments, and the execution environment defines the methods and classes that are available in a specific platform. A Bundle object logically represents a bundle into

OSGi framework and it defines the API to manage the bundle's lifecycle. The BundleContext represents the execution context associated to the bundle. It basically offers some methods for the deployment and lifecycle management of a bundle and other methods for enabling the bundle interaction via services. It is interesting to notice that the BundleContext interface has methods to register BundleListener and FrameworkListener objects for receiving event notifications. These methods allow to monitor and to react to execution-time changes into the framework and to take advantage of the flexible dynamism of OSGi bundles. Finally, the BundleActivator offers a hook into the lifecycle layer and the ability to customize the code that must be executed when a bundle is started or stopped. The class implementing the BundleActivator inside a bundle is specified adding the Bundle-Activator header to the bundle manifest.

As shown in Figure 1, firstly, a bundle must be installed into OSGi framework. Installing a bundle into the framework is a persistent operation that consists in providing a location of the bundle JAR file to be installed (typically a URL) and then saving a copy of the JAR file in a private area of the framework called bundle cache. Then, the transition from installed to resolved state is the transition that represents the automated dependency resolution. This transition can happen implicitly when the bundle is started or when another bundle tries to load a class from it, but it can also be explicitly triggered using specific methods of lifecycle APIs. A bundle can be started after being installed into the framework. The bundle is started through the Bundle interface and the operations executed during this phase (e.g. operations of initialization) are defined by an implementation of the BundleActivator. The transition from the starting to the active state is always implicit. A bundle is in the starting state while its BundleActivator's start() method executes. If the execution of the start() method terminates successfully, the bundle's state transitions to active, otherwise it transitions back to resolved. Similarly, an active bundle can be stopped and an installed bundle can be uninstalled. When uninstalling an active bundle, the framework automatically stops the bundle first. The bundle's state goes to resolved and then to installed state before uninstalling the bundle.

The OSGi environment is dynamic and flexible and it allows to update a bundle with a newer version even at execution-time. This kind of operation is quite simple for self-contained bundles but things get complicated when other bundles depend on the bundle being updated. The same problem exists when uninstalling a bundle, both the updating and uninstalling operations can cause a cascading disruption of all the other bundles depending on it. This happens because, in case of updating, dependent bundles have potentially loaded classes from the old version of the bundle, causing a mixture of loaded old classes and new ones. The same inconsistent situation occurs when a dependent bundle cannot load classes from a bundle that has been uninstalled. The solution for this scenario is to execute the updating and uninstalling operation as a two-step operation: the first step prepares the operation; the second one performs a refreshing. The refreshing allows to recalculate the dependencies of all the involved bundles, providing a control of the moment when the changeover to the new bundle version or removal of a bundle is triggered for updates and uninstalls. Therefore, each time an update is executed, in the first step the new version of the bundle is introduced and two versions of the bundle coexist at the same time. Similarly, for uninstalling operations, the bundle is removed from the installed list of bundles, but it is not removed from memory. In both cases, the dependent bundles continue to load classes from the older or removed bundle. Finally, a refreshing step is triggered and all the dependencies are computed and resolved again. In conclusion, the lifecycle layer provides powerful functionalities for handling, monitoring and reacting to the dynamic lifecycle of bundles. The next section presents the last but not the least layer of the OSGi framework: the service layer.

Java OSGi and Microservices OSGi allows the combination of microservices and nanoservices. Leveraging the OSGi service layer, it is possible to implement microservices internally composed by tiny nanoservices. The final resulting architecture will be composed by a set of microservices, each one running on its own OSGi runtime and communicating remotely with the other microservices. Internally, a single microservice may be implemented as a combination of multiple nanoservices that communicate locally as a simple method invocations. Secondly, OSGi offers an in-built dynamic nature. Developing microservices using OSGi means having a rich and robust set of functionalities specifically implemented for handling services with a dynamic lifecycle. Even more, it makes the microservices able to be aware of their dynamic lifecycle and react consequently to the dynamic changes. The OSGi runtime and its service layer were built upon this fluidity; therefore, the resulting microservices are intrinsically dynamicity-aware microservices. Last but not the list, OSGi makes the microservices architecture a more flexible architecture with respect to service decomposition. One drawback of microservices is the difficulty of performing changes or refactoring operations that span multiple microservices. When designing a microservices architecture, understanding exactly how all the functionalities should be decomposed into multiple small microservices is an extremely difficult task, which requires defining explicit boundaries between services and establishing once for all the communication protocols that will be adopted. If in future, the chosen service decomposition strategy turns to be no more the best choice or only a modification involving the movement of one or multiple functionalities among different microservices is required, performing this change may become extremely difficult because of the presence of already defined microservices boundaries and communication protocols. On the contrary, the OSGi Remote Services offers a flexible approach for defining the microservices boundaries. Indeed, a set of functionalities implemented by an OSGi service can be easily moved from a local runtime to a remote one without any impact on other services. Therefore, an already defined microservices decomposition strategy can be modified by reallocating the functionalities offered by services at any time. For example, one of two OSGi services previously designed for being on the same runtime (i.e. within the same microservice boundary) can be moved on another remote OSGi runtime without any difficult changes. The interaction between a distribution provider and a distribution consumer in OSGi takes place always as the two entities were on the same and local runtime: the distribution manager provided by the Remote Services specification transparently handles the remote communication. Moreover, this remote communication is completely independent from the communication protocols; therefore, any previous choice is not binding at all. In conclusion, OSGi enriches the microservices architecture with new and powerful dynamic properties and a flexible model able to support elastic and protocol-independent service boundaries. Moreover, it provides a level of service granularity highly variable allowing the combination of microservices and nanoservices. The only but very relevant drawback of OSGi with respect to microservices architectural pattern is the complete cancellation of technological freedom that characterizes microservices. OSGi is a technology exclusively designed for Java and implementing a microservices architecture based on OSGi necessary requires to adopt Java for developing the microservices. This does not mean that a microservice implemented using OSGi cannot be integrated with other services implemented with different technology; an OSGi remote service, for example, can be exposed externally also for not-OSGi service consumers, loosing however all the OSGi service layer benefits. It actually means that if Java and OSGi are not widely adopted for implementing the majority of the microservices of the architecture, the OSGi additional features lose their effectiveness. OSGi represents also a very powerful and dynamic service-oriented platform due to the several features offered by its service layer [17].

Finally, the last relevant and powerful feature of the OSGi service layer is the flexibility offered by the Remote Services Specification [20]. The OSGi framework provides a local service registry for bundles to communicate through service objects, where a service is an object that one bundle registers and another bundle gets. However, the Remote Services Specification extends this behaviour in a very powerful and flexible manner, allowing the OSGi services to be exported remotely and independently from the communication protocols. The client-side distribution provider is able to discover remote endpoints and create proxies to these services, which it injects into the local OSGi service registry. The implementation of the discovery phase depends on the chosen distribution provider implementation (e.g. The Apache CXF Distributed OSGi [3] implementation provides discovery based on Apache Hadoop Zookeeper [9]). Another additional and powerful feature of OSGi Remote Services is the ability to be independent from the underlying communication protocol adopted for the service exportation. A distribution provider may choose any number of ways to make the service available remotely. It can use various protocols (SOAP, REST, RMI, etc.), adopting a range of different security or authentication mechanisms and many different transport technologies (HTTP, JMS, P2P, etc.). The Remote Services specification offers a layer of indirection between the service provider and the distribution provider, leveraging the concepts of intents and configurations. They basically allow the service provider to specify just enough information to ensure that the service behaves as expected, then the task of the distribution provider is to optimize the communications for the environment in which they are deployed.

3. A Microservices Architecture for Adaptive Real-time IoT Stream Processing

Senseioty [21] is an IoT platform designed to accelerate the development of end-to-end solutions and verticals, revolving around the concept of insights-engineering, the seamless integration between data ingestion and distribution, data analytics and on-line data analysis. Senseioty is developed in Java as a set of highly cohesive OSGi microservices. Each Senseioty microservice can either be used together with Amazon AWS or Microsoft Azure managed services or deployed on private cloud or on-premises to accelerate and deliver full-fledged end-to-end IoT solutions for the customer. Senseioty features also an SDK to implement rapid verticalizations on top of its rich set of JSON RESTful APIs and analytics services. Senseioty automates the integration of IoT operational data with analytics workflows and provides a common programming model and semantics to ensure data quality, simplify data distribution and storage and enforce data access policies and data privacy. Senseioty is natively integrated with both Microsoft Azure IoT and Amazon AWS IoT and it can also operate on private and hybrid cloud to provide the maximum flexibility in terms of cloud deployment models Senseioty offers a wide variety of interesting and flexible functionalities that should give an idea of the flexibility and interoperability offered by a microservices architecture in the IoT context: Single-sign-on services for user and devices along with user management; Access policies microservice to protect resources and devices against unauthorized access and to guarantee data privacy; Flexible and unified programming interface to manage and provision connected devices.; Persistence of time series in Apache Cassandra clusters; Powerful and flexible way to communicate different microservices together and to implement remote services discovery based on the OSGi Remote Service specification; Senseioty microservices can be deployed at the three different layers of the hybrid-cloud stack (cloud layer, on-premises layer and edge layer); Deep-learning workflows based on neural networks and to push them on connected devices, in order to run analytics workflow on the edge. Senseioty integrates Apache Spark, a powerful Distributed Data Stream Processors engine to analyse data stream in real-time and provide on-line data analytics on the cloud, leveraging both neural network and statistical learning techniques to analyse data. Finally, Senseioty provides a rich set of IoT connectors to integrate standard and custom IoT protocols and devices.

FlairBit extensively adopts in its platform Apache Karaf [6], a powerful and enterprise ready applications runtime built on top two famous OSGi implementation (Apache Felix and Eclipse Equinox) that offers some additional and useful functionalities, such as the concept of feature.

One problem exposed by FlairBit, which is usually a problem common to the majority of IoT platform, is to have two different levels of data stream processing: The edge level; The core/cloud level. The two levels offer different benefits but impose quite different requirements. The term “edge” in IoT platforms generally means the location at the boundary of the network near the devices that produce the data. Edge devices are usually quite simple devices that play the role of gateways, enabling the collection and the transmission of data. However, modern edge devices can also offer enough computational resources to enable more complex functionalities, such as pre-processing, monitoring or pre-filtering. Moving the stream processing elaboration directly on the edge of the IoT platform takes the name of Edge analytics and the consequent benefits are quite notable: Lowest possible latency, having a stream processing unit deployed directly on an edge devices makes possible to respond quickly to events produced locally, avoiding to send data to the remote cloud/core of the platform over the network; Improved reliability, moving the stream processing rules on the edge allows the edge devices to operate even when they lose the connection with the core platform; Reduced operational costs, pre-processing and pre-filtering data directly on the edge makes possible to save bandwidth, cloud storage and computational resources consequently lowering operational costs.

On the other hand, edge analytics imposes some stringent requirements in term of computational power. The modern edge devices are becoming more and more powerful, but the computational resources offered by this kind of devices are limited. Therefore, the technologies installed on the edge must be lightweight and it is likely that they are quite different technologies from those applied on the core platform. Indeed, the stream processing units on the edge usually deals with simple filtering rules and streams of data restricted to the local sensors or devices, without the need of scaling the stream processing job across multiple machines.

On the core platform, the context is completely different. In this scenario, the stream processing engines must be able to deal with different workloads and the computational resources abound. They must be able to scale the computation across a cluster of machines in order to handle large volume of data and more intensive tasks, for example joining and aggregating different events from different streams of data. Therefore, the need of scaling capabilities overcomes the limit of the computational resources. The main goals of the proposed extension of the Senseioty architecture are as follows: 1) Providing adaptivity, meaning that the stream processing units can be indifferently allocated on the edge or on the core and moved around. This makes possible to cover the two different levels of data stream processing, the edge level and the core/cloud level, and exploiting all their different benefits. 2) Providing flexibility, allowing a punctual and on-demand deployment of the stream processing units. The user or the client application/service defines when and where allocating, starting, stopping and deallocating the stream processing rules. 3) Providing a set of portable and composable rules that can be defined in a standard way and then automatically deployed on different stream processing engines without depending on their own languages and models. The rules can be combined together, in order to apply a sort of stream processing pipeline. The rules are not only dynamically manageable, but composable and engine-independent. The reference structure of the resulting architecture is shown in

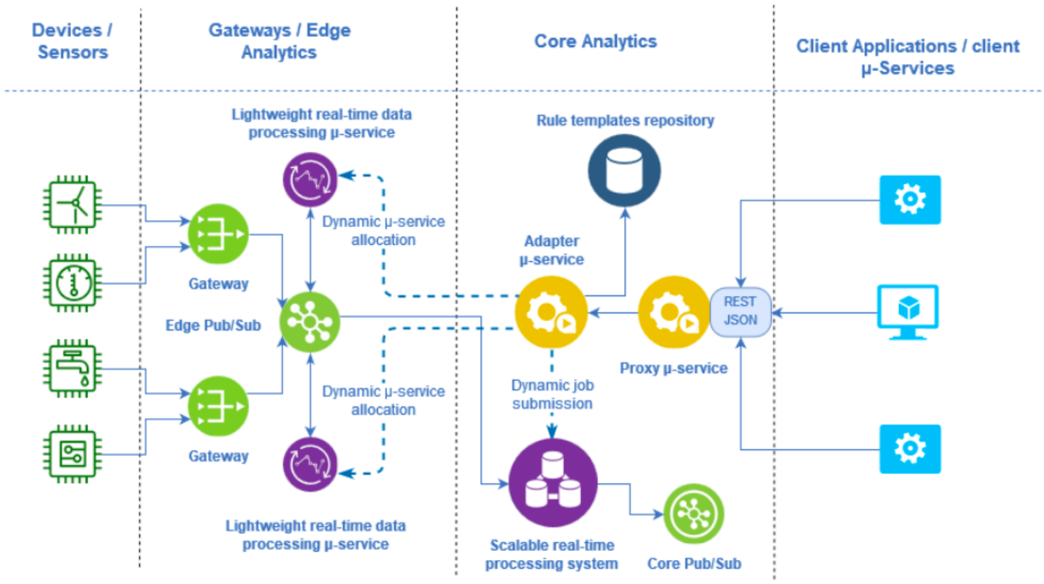

Figure 2. Reference Architecture

Fig. 2. There are two main components in our architecture:

- The proxy μ-service: the entry point of the architecture, offering a RESTful API for installing, uninstalling, starting, stopping and moving stream processing rules on demand.

- The adapter μ-service. It is responsible for physically executing the functionalities offered by the proxy μ-service, interacting with the different stream processing engines available on the edge and on the core of the architecture.

The proxy μ-service represents the entry point of the architecture. It offers a RESTful JSON interface, a standard choice in microservices architecture (and it is usually adopted in Senseioty) in order to offer a solution as much compatible and reusable as possible. The REST API offers the following functionalities:

URL Method	Request Body	Response Body
/api/install POST	JSON installation object	JSON jobinfo object
/api/uninstall POST	JSON jobinfo object	JSON jobinfo object
/api/start POST	JSON jobinfo object	JSON jobinfo object
/api/stop POST	JSON jobinfo object	JSON jobinfo object
/api/move POST	JSON relocation object	JSON jobinfo object

The method *install* installs the rule on the required resource and engine defined by the json installation object. The method *uninstall* uninstalls the rule identified by the jobinfo request object. The method *start* runs the rule identified by the jobinfo request object. The method *stop* stops the execution of the rule identified by the jobinfo request object. The method *move* moves the rule identified by the jobinfo request object to the target runtime defined by the relocation object. Interaction with the proxy μ-service is carried out through the JSON objects of the following form:

```
// JSON INSTALLATION OBJECT
{
   "headers":{
      "runtime":<ENGINE>,
      "targetResource":<URL>,
      "jobType":<JOB_TYPE>
   },
   "jobConfig":{
      "connectors":{
         "inputEndpoint":<STRING>,
         "outputEndpoint":<STRING>
      },
      "jobProps":{
         "condition":< ">" | ">=" | "=" | "<" | "<=" >,
         "threshold":< INT | FLOAT | DOUBLE | STRING >,
         "fieldName":<STRING>,
         "fieldJsonPath":<JSON_PATH>
      }
   }
}
// JSON JOBINFO OBJECT
{
   "runtime":<ENGINE>,
   "jobId":<STRING>,
   "jobType":<JOB_TYPE>,
   "jobStatus":<INSTALLED|RUNNING|STOPPED|UNINSTALLED>,
   "configFileName":<STRING>
}
// JSON RELOCATION OBJECT
{
   "target_runtime":<ENGINE>,
```

```
    "targetResource":<URL>,
    "jobInfo":<JSON_JOBINFO_OBJECT>
}
```

The JSON installation object is the object that the client must provide to the proxy in order to describe the stream processing rule to be allocated. The headers field indicates the runtime engine that will execute the rule (the <ENGINE> value depends on the engines supported by the implementation), the target resource which is the machine on which allocating the rule (the value can be an URL or a simple ID, depending on the architecture implementation) and the job type, which indicates the kind of rule that the jobProps field contains. The implementation of the architecture supports a set of predefined rule templates identified by a unique name that must be inserted in the jobType field (e.g. single-filter, sum-aggregation, avg-aggregation, single-join etc.). Ideally, we would like to have a solution able to support any kind of rule expressible with a standard query stream language (e.g. the Stanford CQL [23]), but in practice this is not achievable because each stream processing engine has its own model and language with its own level of expressiveness. Therefore, it is extremely complicated to implement a compiler able to validate an arbitrary query and to compile and translate it to the model or language of the underlying stream processing engine. Considering this scenario, we provide an architecture able to support a set of predefined rule templates. A possible subset that should be compatible with the majority of stream processing engines includes (using a SQL like syntax):

- Filtering query (e.g. SELECT * FROM inputEvents WHERE field > threshold)
- Aggregation query over a window (e.g. SELECT SUM(field) FROM inputEvents[5 s])
- Joining query between two streams over windows (e.g. SELECT field1 field2 FROM stream1[1m] JOIN stream2[1m] ON stream1.field3 = stream2.field)

This is of course only a possible subset, which must be verified and extended considering the engines selected for the implementation.

The connectors field specifies the information needed for reading and writing the events consumed by the rule from/to a pub-sub broker. Again, the format of these fields depends on the pub-sub broker adopted in the implementation, but it in general the required parameters are a simple URL or a queue or topic name. It is important to notice that the presence of two pub-sub brokers (one on the edge and one on the core) makes possible to combine and compose the rules in order to obtain stream processing pipelines. The jobProps field contains the parameters needed for allocating the stream processing rules. The format of this field depends on the rule template specified in the jobType field. The JSON jobInfo object is the object that contains all the necessary information that must be provided in order to perform all the other operations (starting, stopping, uninstalling or moving the rule) and it is created by the adapter μ-service and returned to the client by the proxy μ-service. It contains some information specified by the JSON installation object, with the addition of a jobId (a unique identifier for the installed rule instance), a jobStatus (it indicates the current execution status of the rule) and a configFileName (the name of the configuration file that represents the materialization of the jobConfigs field specified in the JSON installation object). The role of the configuration file will be clarified shortly when describing the adapter μ-service. The JSON relocation object is the object required for moving a rule from the current runtime to a target runtime. It contains the jobinfo object describing the selected rule and information regarding the target runtime (the engine and the resource URL or ID identifying the target machine).

The adapter μ-service is responsible for actually executing the functionalities offered by the proxy μ-service. It offers the following procedures: A procedure for installing a new rule; A procedure for starting/stopping/uninstalling an existing rule; A procedure for moving an existing rule from its current runtime to another one. During the installation procedure, the adapter μ-service translates the information received from the proxy μ-service into executable rules via a sort of parametrization as shown below: The adapter μ-service has access to a repository from where it can download the rule template corresponding to the jobType and runtime fields expressed in the JSON installation. The rule template is any sort of predefined executable file (for our purposes will be a JAR archive) that can be modified injecting a configuration file containing the rule parameters specified by the JSON installation object. Therefore, in case of rule installation, the adapter μ-service downloads the relative rule template, creates and injects the configuration file and then install the rule on the target runtime. If the target runtime is a distributed stream processing engine, the executable template is actually an executable job that is submitted to the cluster manger. If the target runtime is a lightweight and non-distributed stream processing engine for the edge, the rule template is actually an independent μ-service that is installed on the target machine on the edge. Finally, the adapter μ-service creates the JSON jobinfo object with the necessary information that will be returned to the client. In case of starting, stopping and uninstalling operations, the adapter μ-service acts always depending on the runtime engine associated to the rule, as shown below.

In case of distributed stream processing engine, it communicates with the cluster manager for executing the required operation. On the other hand, in case of lightweight stream processing μ-service on the edge, the implementation must provide a mechanism to interact dynamically with target runtime. It is intuitive to understand that a technology like OSGi and its bundle lifecycle naturally fits this scenario. OSGi is the main technology adopted in the prototype that will be described in the next section, but this architecture description section is intentionally lacking of technical and implementation details in order to be as much general as possible. The idea is to offer a guideline proposal that must be refined with respect to technologies selected for the implantation, which may be completely different from those selected for our prototype. Indeed, one benefits of a microservices architecture is the technological freedom.

Finally, for moving an existing rule across different runtimes the adapter μ-service acts as follows. First, it checks if the rule to be moved is running and eventually it stops its execution. Secondly, it uninstalls the current rule, it downloads the new template for the new target runtime and it injects the previous configuration file. Finally, it installs the new rule on the new target runtime, starting the execution of the new rule if it was previously running. In our architecture we introduce two pub/sub brokers. Having two event dispatcher systems in the architecture, one for edge level and another one for the cloud/core level, makes possible to implement composable stream processing rules. Indeed, the connectors field in the JSON installation object allows to specify the queue or topic names from where reading events and where writing the output events. This means that any rule can be concatenated with other rules in order to implement a stream processing pipeline. For example, two filtering rules can be combined on the same edge-device leveraging the edge pub/sub broker in order to create a two-step filter. Moreover, a pre-filtering rule on the edge can be applied on top of an aggregation rule executed at cloud/core level in order to reduce the amount of data sent over the network.

The last aspect to consider is the client application layer. As already explained, the proxy μ-service offers a simple RESTful JSON API accessible from any kind of client. For this reason, the functionalities offered by the API can be employed by other μ-services in the context of a larger platform (e.g. Senseioty) and combined with other additional functionalities (e.g. the authentication and authorization μ-services offered by Senseioty). Moreover, it is possible to provide a web interface that lets a user to interact directly with the proxy μ-service, defining and managing the rules. The API offers all the functionalities needed for implementing an adaptive monitoring rule relocation procedure. The only prerequisites are: Having access to a stream of events logging statistics about the performance and workload of the edge-devices; Having the possibility to store and update the information mapping the rules (i.e. the JSON jobInfo objects) to the IDs of the edge devices that are executing the rules.

4. Prototype Implementation

In order to implement a prototype of the proposed architecture we considered a lightweight stream processing engine for edge analytics called Siddhi [22]. Siddhi is an effecting streaming processing engine that provides an SQL-like stream language with a rich expressive power. It allows any sort of

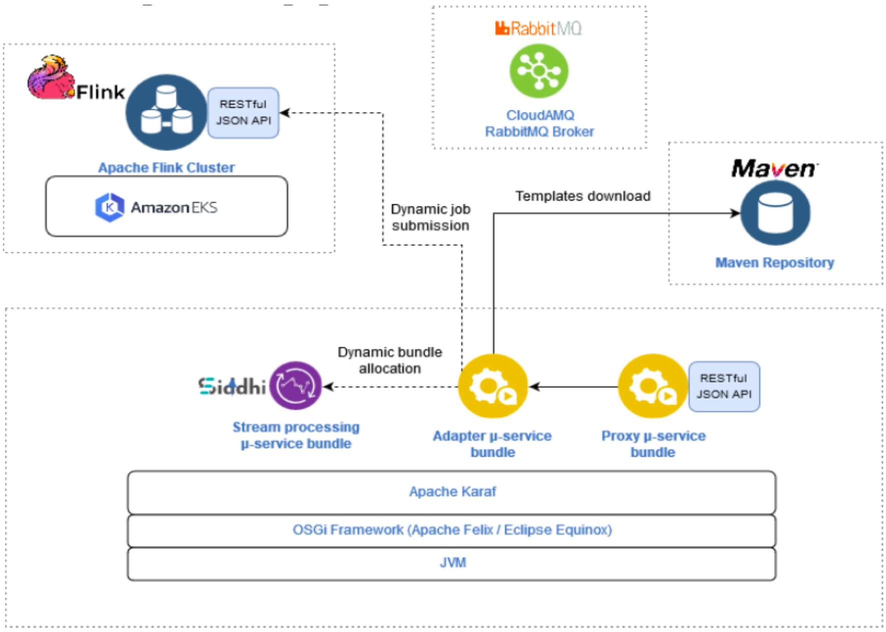

Figure 3. Prototype Structure

stateful and stateless operation, timing and counting windows, aggregation and join functions. It also supports different event formats (JSON, XML, etc.) for specifying event patterns for complex event processing (CEP). It provides a rich set of external event source integration, such as Kafka, MQTT, RabbitMQ and other brokers and provides a lightweight runtime compatible, e.g., with Android devices. The Siddhi libraries were transformed and wrapped into well-defined OSGi bundles. The Senseioty SDK provides some Java project templates explicitly configured for applying OSGi specific tools (e.g. Bnd tools [10]) able to create a JAR with OSGi meta data (i.e. a bundle) based on instructions and the information in the class files. A feature for the Karaf runtime, collecting all the bundles needed by Siddhi as dependencies, was created. The Senseioty SDK offers some functionalities able to discover all the dependencies and transitive dependencies required by a bundle and then to materialize them in the form of a Karaf feature. Therefore, the provisioning phase of a Siddhi application on a Karaf runtime (by provisioning application, it means install all modules, configuration, and transitive applications) requires now only a simple and automatic feature installation. Although this step required a lot of technical passages, a detailed description is beyond the scope of the paper. Once obtained a fully OSGi-compliant stream processing engine for edge analytics purposes, the second step consisted in exploring and selecting another engine able to scale across a cluster of machines for core/cloud analytics purposes. During this phase, an analysis and some implementation of spike test programs were performed for the following stream processing technologies: Ignite [5]; Samza [7]; Flink [4]; Storm [8]; Streams [11]. Apache Flink turned out to be the most flexible available solution. It has a rich and complete API that follows a declarative model very similar to the Spark Streaming one and it has also a powerful additional library for complex event processing for specifying patterns of events. Moreover, it has a rich set of out-of-the-box external source connectors, a flexible resource allocation model based on slots independent from the number of CPU cores, and it is very easy to deploy a Flink cluster on Kubernetes. Based on all the above considerations, the structure of the implemented prototype is shown in Fig. 3. The proxy and the adapter μ-services are implemented as OSGi-bundles deployed on a Karaf runtime. The proxy μ-service offers a RESTful JSON API with the following functionalities. First, it provides an installation functionality for installing a filtering rule for events in JSON format. The rule can be indifferently instantiated as an independent Siddhi μ-service (implemented in the form of an OSGi bundle) or deployed as a distributed job on a Flink cluster. It also provides a starting, stopping and uninstalling functionalities for removing or handling the rule execution, and a moving functionality for relocating a rule from a Siddhi runtime to a Flink runtime or vice versa. The RESTful API was implemented using Apache CXF [2], an open-source and fully featured Web services framework. In this preliminary implementation, the runtime supported are Siddhi and Apache Flink and only one rule type is available: a threshold filter for events in JSON format. The client can specify a filtering rule defining the following jobProps in the JSON installation object:

```
// JSON INSTALLATION OBJECT

{ ...
      "jobProps":{
         "condition":< ">" | ">=" | "=" | "<" | "<=" >,
         "threshold":< INT | FLOAT | DOUBLE | STRING >,
         "fieldName":<STRING>,
         "fieldJsonPath":<JSON_PATH>
      }
   }
}
```

The prototype supports one rule type: a threshold filter for events in JSON format. The parameters specified by this rule type will be injected into two different rule templates that are implemented using the model and libraries provided by Siddhi and Flink. In practice, the rule parameters can be instantiated in two different rule templates:

```
// PROTOTYPE INSTALLATION ADAPTER SERVICE PROCEDURE
private bundleContext;

Install_rule (JsonInstallationObj req) {
   // Get and configure the right template
   mavenUrl = get_template_maven_url (req.headers.runtime, req.headers.jobType)
   ruleTemplate = download_from_maven_repo(mavenUrl)
   configurationFile = create_config_file(req.headers, req.jobConfig)
   configFileName = save(configurationFile)
   deployableRule = inject_config_file(ruleTemplate, configurationFile)

   // Install the rule as an independent  Siddhi service
   if (req.headers.runtime == SIDDHI)
      job_id = install_OSGi_bundle ( bundleContext, deployableRule)

   // Submit the rule to the remote Flink Cluster
   if (req.headers.runtime == FLINK)
      job_id = submit_to_cluster_manager (deployableRule)

   jobInfo = new JobInfo(runtime, jobId, jobType,
    status.INSTALLED, configurationFileName )
   return jobInfo
 }
```

In the form of an OSGi bundle (i.e. a μ-service) encapsulating a Siddhi runtime executing the filtering rule; In the form of a Flink job, which can be submitted to a Flink cluster. In this preliminary version of the prototype, the Siddhi bundles are installed and executed on the same OSGi runtime of the proxy and adapter μ-service. The remote installation on an edge-device can be easily integrated in future. Instead, the Flink runtime is installed on a remote Kubernetes cluster on the Amazon EKS service. The two rule templates previously cited are implemented in the form of a JAR file. Both templates are stored as Maven artifact into a Maven repository. Maven [11] is a tool used for building and managing any Java-based project and a Maven repository is basically a local or remote directory where Maven artifacts are stored. A Maven artifact is something that is either produced or used by a project (e.g. JARs, source, binary distributions, WARs etc.). In this case, both templates are implemented as JAR files. In order to download a Maven archetype from a Maven repository, an OSGi bundle (i.e. the adapter μ-service) needs only to specify a Maven URL identifying the artifact. Then the URL resolution and the JAR download is handled by Pax URL [12], a set of URL handlers targeting the OSGi URL Handler Service. This mechanism is applied by the adapter μ-service for downloading the rule template for installing the rule on the required stream processing engine. The template to be download (and its relative Maven URL) depends on the jobType and runtime fields specified in the JSON installation object. Therefore, the adapter μ-service must have some predefined information that bind a Maven URL to a specific jobType and runtime. In this preliminary implementation, the above mentioned information are stored in memory into a simple hashTable, but for real purposes a simple database is required. The adapter μ-service provides the implementation of the procedures for installing, starting, stopping, uninstalling and moving the rules and it is responsible for injecting the rule parameters into the two different templates previously cited. When the adapter μ-service has to install a new rule, considering the jobType (in this case there is only one jobType: a filter) and the runtime (Siddhi or Flink) specified by the JSON installation object, it downloads the corresponding JAR file template from the Maven repository. Once obtained, the adapter μ-service translates the jobProps in a configuration file that is injected into the JAR template file. At this point, depending on the runtime chosen, the template rule is installed in two different ways. In case of a Flink job, the JAR template is sent to the Flink cluster manager using a REST API offered directly by Flink. On the other hand, in case of an OSGi bundle implementing the Siddhi filtering application, the bundle is installed on the Karaf runtime using the OSGi methods offered by the lifecycle layer. In this preliminary prototype, for the sake of simplicity, the OSGi bundle is installed on the same runtime of the proxy and adapter μ-service, but actually it should be installed on a remote runtime (i.e. a gateway device) on the edge of the IoT platform. Once the required rule is correctly installed on the target runtime, the adapter μ-service creates a JSON jobInfo object collecting all the relevant information about the just installed rule. In particular, it keeps trace of a jobId (corresponding to a bundle id for a Siddhi rule and to a jobId for Flink rule) and a configFileName (corresponding to a unique name of the generated configuration file, useful for reusing the file when moving the rule for one runtime to another). For all the other operations (starting, stopping, uninstalling and moving), the adapter μ-service uses the information provided by the jobInfo object and the methods offered by the OSGi lifecycle layer or the Flink REST API. The Siddhi OSGi bundles are installed on the same runtime of the proxy and adapter μ-service, but actually they should be installed on a remote runtime (i.e. a gateway device) on the edge of the IoT platform. This behaviour has been successfully implemented in Senseioty by FlairBit, which has extended the OSGi functionalities for communicating with remote runtime and it can be easily integrated in this prototype implementation in future. In practice, a remote OSGi runtime is connected to the core platform through two communication channels. A bidirectional channel used for communicating configurations options and statements. In this scenario, the adapter μ-service would use this channel to notify the target runtime about downloading the required bundle rule: it requires only a symbolic ID or URL to identify the target runtime. Possible communication protocols adopted for this channel are MQTT or TCP. A one-directional channel used by the remote OSGi runtime for download a remote resource, in this case the bundle rule notified by the adapter μ-service. A possible example of communication protocol adopted for this channel is FTP. This communication mechanism can be used by the adapter μ-service for executing all the required interactions with a remote OSGi runtime (installing, starting, stopping and uninstalling a Siddhi bundle). Another relevant feature implemented by this prototype is the rule composability, meaning that multiple filtering rules can be concatenated in order to obtain a multiple-step filtering pipeline. Indeed, the currently supported filtering rules are easily composable because they read and write events from a RabbitMQ broker. RabbitMQ [19] is an open source message broker supporting multiple messaging protocols and it was chosen for this prototype implementation because both Siddhi and Flink provide out-of-the box connectors for consuming and writing event from a RabbitMQ broker. More specifically, RabbitMQ is adopted in this prototype for handling streams of events in JSON format using the AMQP protocol [1]. The role of an AMQP messaging broker is to receive events from a publisher (event producer) and to route them to a consumer (an application that processes the event). The AMQP messaging broker model relies on two main components:

- *Exchanges*, which are components of the broker responsible for distributing message copies to queues using rules called bindings. There are different exchange types, depending on the binding rules that they apply. This prototype uses only exchanges of type direct, which delivers messages to queues based on a message routing key included when publishing an event.
- *Queues*, which are the component that collect the messages coming from exchanges. A consumer reads the events from a queue in order to process the messages.

Therefore, when specifying the jobConfigs field in the JSON installation object, a client must provide in the connectors field the information needed for reading and writing events from/to an AMPQ queue. More specifically, is necessary to specify the parameters in the connectors field: For specifying the input source for the event, the following information are needed:

- *inputEndPoint*: the URL for connecting to the RabbitMQ broker (it might be different from outputEndPoint).
- *inputExchange*: the name of the exchange from which the input queue will read the messages. If the exchange does not already exist, it is created automatically.
- *inputQueue*: the name of the queue that will be bind to the inputExchange. If the queue does not already exist, it is created automatically.
- *inputRoutingKey*: the routing key that is used for binding the inputExchange to the InputQueue.

On the other hand, for specifying the output source of events, these information are required:

- *outputEndPoint*: the URL for connecting to the RabbitMQ broker.
- *outputExchange*: the name of the exchange where to publishing the events. If the exchange does not already exist, it is created automatically.
- *outputRoutingKey*: the routing key that is included to the event when publishing it.

Leveraging these features, the prototype allows to create stream processing pipelines of arbitrary complex. For example, multiple Siddhi filters can be concatenated with other filters executed on Flink. In practice, there is the need of two message brokers: one for the edge level and one for the cloud/core level. This aspect makes possible to concatenate multiple edge rules without the need of sending events to a remote broker in the core of the platform, avoiding to introduce unnecessary latency. RabbitMQ may be a reasonable choice for the cloud/core level scenario, but for the edge level, the choice must be carefully evaluated for not overloading the edge/gateway devices. For FlairBit and Senseioty purposes, considering that the edge/gateway devices are provided with an OSGi runtime, it may be a reasonable choice to take advantage of the OSGi Event Admin Service [16]: an inter-bundle communication mechanism based on an event publish and subscribe model. This sort of OSGi message broker can be easily paired with the Remote Service functionalities in order to connect multiple OSGi runtimes. This solution makes possible to obtain a message broker at edge level, without the need of adding an external and additional technology. The drawback is that we have to develop a customized connector implementation for each stream processing engine, in order to consume events from the OSGi broker

5. Related Work and Conclusions

In this paper we have proposed an adaptive solution for satisfying the dynamic and heterogeneous requirements that IoT platforms are inevitably facing. During the research and development path that led to our proposal, we investigated all the features of the microservices architectural pattern, with the aim of deeply understanding the level of flexibility and dynamicity that this approach is able to offer. OSGi turned out to be the perfect booster for those dynamic and flexible features that we were looking for. Then, on the basis of the industrial experience of FlairBit, we formulated a proposal architecture accompanied by a preliminary prototype implementation. Our solution meets the need to introduce different real-time stream processing technologies in IoT platforms, in order to offer streaming analytic functionalities on the different architectural levels of IoT applications. The innovative aspect resides in a limitation of the expressiveness power for defining stream processing rules, in favour of a much more flexible and dynamic deployment model. Streaming rules are restricted to a predefined and manageable set of templates, which allows to handle rules as resources dynamically allocable, composable and engine independent. These resources can be indifferently deployed at edge-level or core-level and moved around at any time.

Comparing our proposal with similar real-time streaming functionalities offered by the IoT platforms of Amazon, Azure and Google, the dynamic features of our solution can be potentially promising and innovative. Amazon offers AWS IoT Greengrass [16] as a solution for moving analytical functionalities directly on edge devices. It is basically a software that once installed on an edge device enables the device to run AWS Lambda functions locally. AWS Lambda enables to run code without provisioning or managing servers. They offer a great level of expressivity with respect to our proposal because they support function implemented with all the most common programming languages. However, AWS IoT Greengrass does not provide any functionality for dynamically moving the Lambda computation back and forth between the edge-level and cloud-level and it is bound to the Lambda execution model. It does not offer any integration with external stream processing engines, which on the other hand can be integrated in our solution as pluggable components as long as template implementations of the supported rule types are provided. Microsoft Azure offers similar functionalities with Azure Stream Analytics on IoT Edge [13]. It empowers developers to deploy near-real-time analytical intelligence, developed using Azure Stream Analytics, to IoT devices. The principle is the same of AWS IoT Greengrass: installing the Azure IoT Edge software we enable the edge devices to locally execute Azure Stream Analytics rules. Azure Stream Analytics is a real-time analytics and complex event-processing engine where streaming rules and jobs are defined using a simple SQL-based query language. Again, the power of expressiveness is much wider with respect to our proposal, but the resulting solution is inevitably bound to the only Azure Stream Analytics engine and no mechanisms for the dynamic relocation of rules between edge and cloud are provided. Finally, Google Cloud IoT [18] integrates the Apache Beam SDK [21], which provides a rich set of windowing and session analysis primitives. It offers a unified development model for defining and executing data processing pipelines across different stream processing engines, including Apache Flink, Apache Samza, Apache Spark and other engines. However, Apache Beam supports only scalable engines suitable for the core-cloud level and it is not designed for supporting edge analytics.

Concerning future research directions, one possibility is to improve the architecture by investigating possible solutions for simplifying the rules' definition. Our API requires to define a JSON object containing the rules' parameters, but for example a web interface or an SDK similar to Apache Beam may offer a higher level approach. In this case, it is required to identify the right trade-off between the level of expressiveness offered by a possible unified model or language and the limits imposed by the presence of predefined rule types and templates. Another interesting point consists in integrating a monitoring μ-service in the application. This monitoring functionality, which is presented as possible application scenario of our proposal, can be formalized in more details in order to become an integral part of our solution. Providing an out-of-the-box monitoring behaviour can be a powerful additional feature useful in many IoT use cases.

21. Senseioty. http://senseioty.com/.
25. Fowler, M.; Lewis, J. Microservices - A definition of this new architectural term. https://martinfowler.com/articles/microservices.html, 2014.
24. Abbott, M.; Fisher, M. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*; Addison-Wesley Professional, 2015.
31. Robert, M. *Agile Software Development: Principles Patterns And Practices*; Pearson, 2003.
28. Richardson, C. Building Microservices: Using an API Gateway. https://www.nginx.com/blog/introduction-to-microservices/, 2015.
29. Richardson, C. Event-Driven Data Management for Microservices. https://www.nginx.com/blog/event-driven-data-management-microservices/, 2015.
30. Richardson, C. Choosing a Microservices Deployment Strategy. https://www.nginx.com/blog/deploying-microservices, 2016.
14. OSGi Alliance. https://www.osgi.org.
15. OSGi Architecture. https://www.osgi.org/developer/architecture/.
27. Hall, R.; Pauls, K.; McCulloch, S.; Savage, D. *OSGi in Action, Creating Modular Applications in Java*; Manning, 2011.
17. OSGi Service Layer. https://osgi.org/specification/osgi.core/7.0.0/framework.service.html.
20. Remote Services Specification. https://osgi.org/specification/osgi.cmpn/7.0.0/service.remoteservices.html.
3. Apache CXF Distributed OSGi. https://cxf.apache.org/distributed-osgi.html.
9. Apache Zookeeper. https://zookeeper.apache.org/.
6. Apache Karaf. https://karaf.apache.org/.
23. A. Arvind, B.S.; Jennifer, W. The CQL Continuous Query Language: Semantic Foundations and Query Execution. http://ilpubs.stanford.edu:8090/758/1/2003-67.pdf, 2003.
22. Siddhi Streaming and Complex Event Processing System. https://siddhi.io/.
10. Bnd tools. https://bnd.bndtools.org/.
5. Apache Ignite. https://ignite.apache.org/.
7. Apache Samza. http://samza.apache.org/.
4. Apache Flink. https://flink.apache.org/.
8. Apache Storm. https://storm.apache.org/.
11. Kafka Streams. https://kafka.apache.org/documentation/streams/.
2. Apache CXF. http://cxf.apache.org/.
12. Kubernetes. https://kubernetes.io/.
19. RabbitMQ. https://www.rabbitmq.com/.
1. AMQP Protocol. https://www.amqp.org/.
16. OSGi Event Admin Service. https://osgi.org/specification/osgi.cmpn/7.0.0/service.event.html.
13. Maven. https://maven.apache.org/.
18. Pax URL. https://ops4j1.jira.com/wiki/spaces/paxurl/overview.