Adaptive Real Time IoT Stream Processing in Microservices Architectures

DIP 2020

27 *November* 2020

Luca Bixio
Stefano Rebora
Matteo Rulli

FLAIRBIT
DATA CENTRIC SOLUTIONS

Giorgio Delzanno

Dibris

UNIVERSITÀ DEGLI STUDI
DI GENOVA

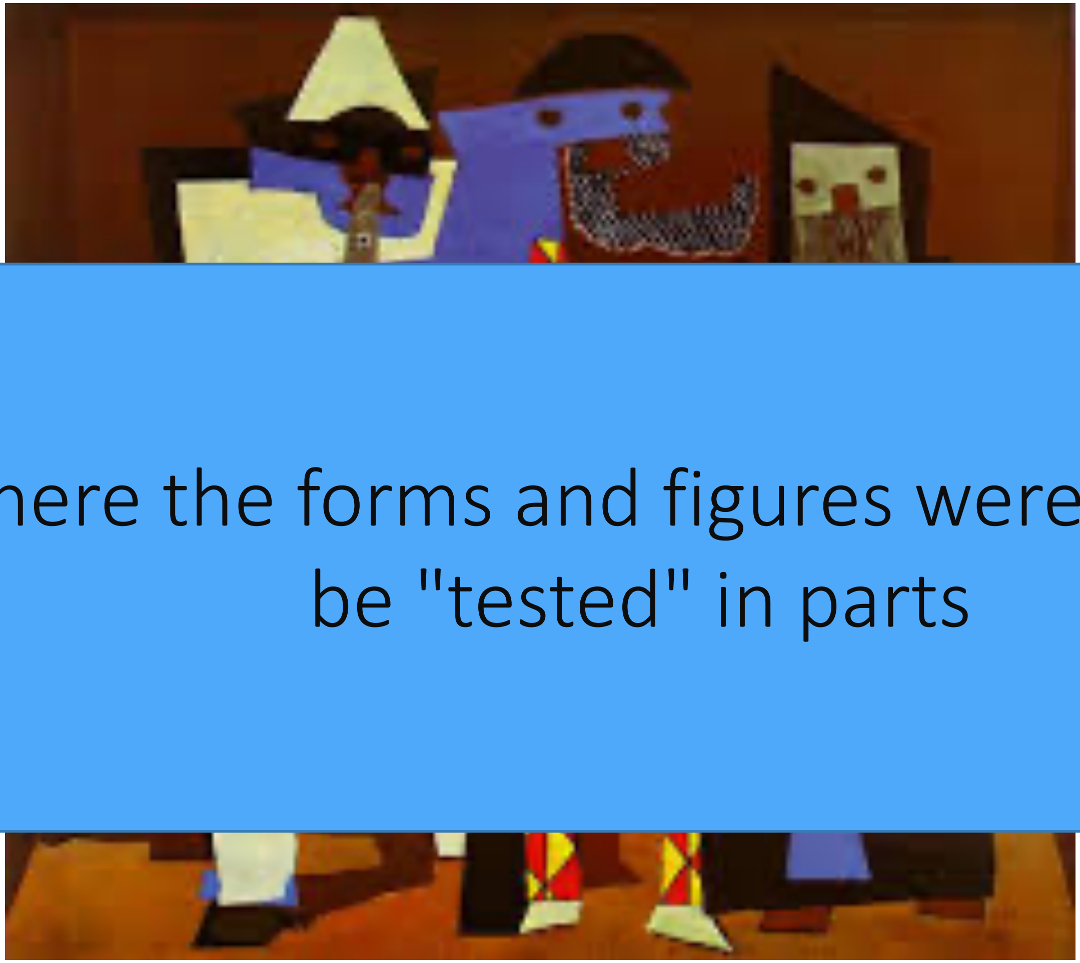
Why this talk?

An excuse for greetings from Giovanna, Davide, Elena, Alessandro, …

… but also …

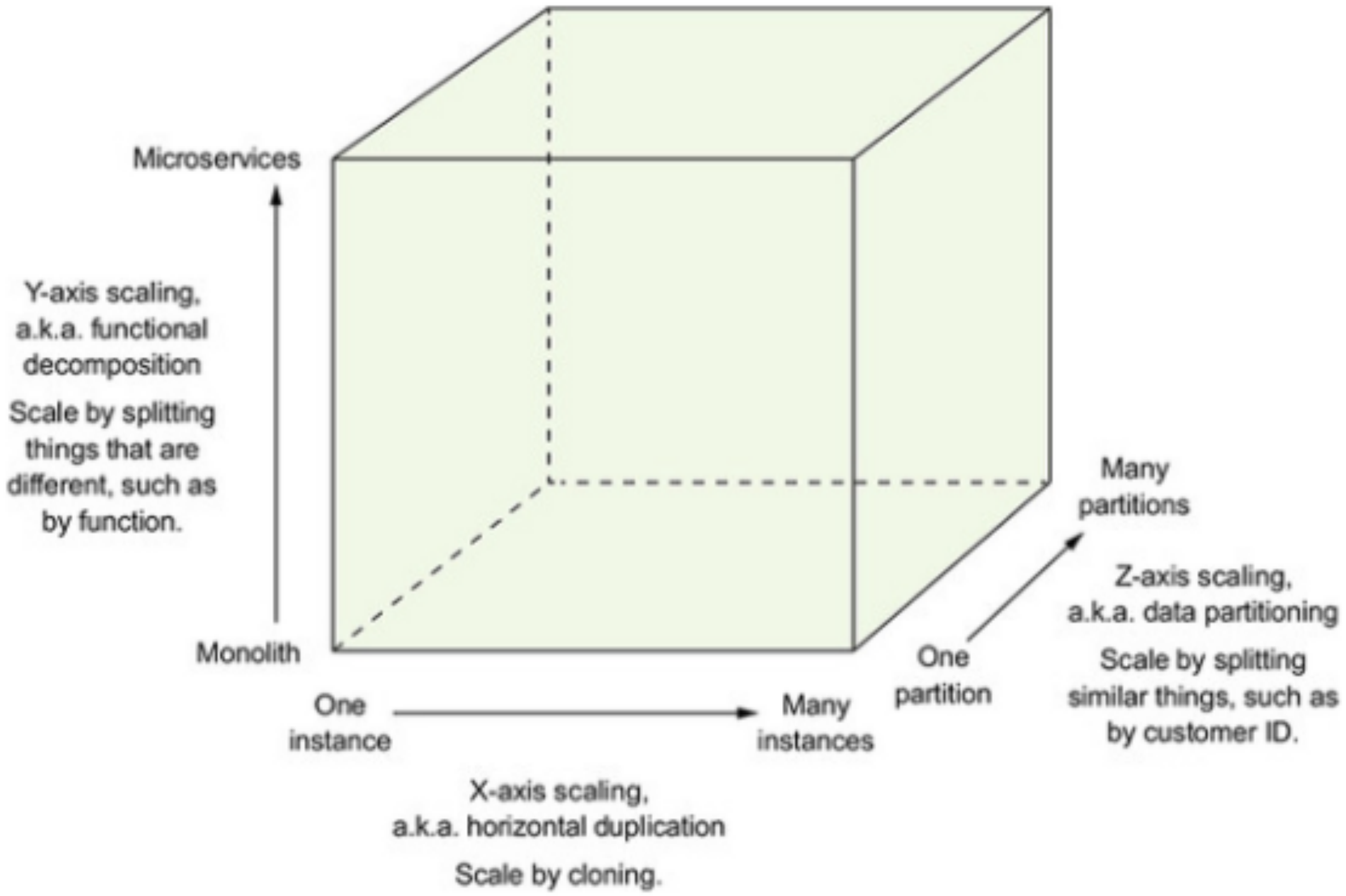
… somehow related to Maurizio's Cubist period (Service Oriented Computing, Choreographies, …)

… somehow related to Maurizio's Cubist period (Service Oriented Computing, Choreographies, …)



cubism, where the forms and figures were **decomposed** to be "tested" in parts

From Cubism to the Microservices Cube-scale Model



Background and Problem Statement

IoT and Big Data

Wide variety of smart devices

Source of large volumes of data
at an unprecedented speed

The value diminishes very fast with time

Real Time Stream processing

Continuous and potentially unbounded sequences of data elements (data streams) from which static queries (a.k.a. rules) continuously extract information in a very short time span (milliseconds or seconds)

IoT platforms + Real Time Stream processing

An IoT platform provides tools, technologies and capabilities for simplifying the development, provisioning and management of IoT applications

Real Time Stream processing in IoT application scenarios

- Anomaly and fraud detection
- Remote Monitoring
- Predictive Maintenance
- **Real-time analytics** (Sentiment analysis, Sports analytics, etc.)
- **Data quality assessment** (Data cleaning, Data profiling to discover inconsistencies and anomalies in the data, etc.)

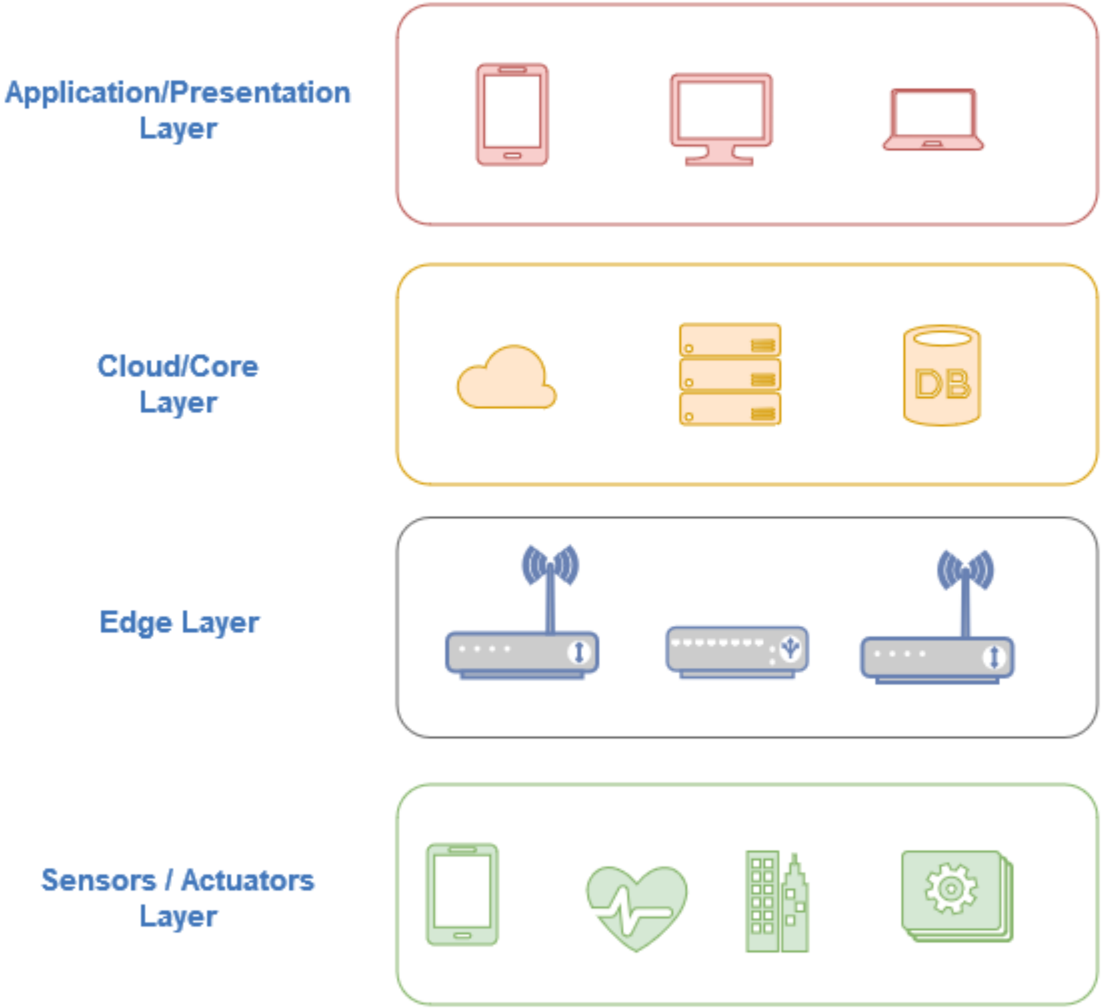
Problem Statement

When Integrating real-time stream processing capabilities in IoT platforms

I. **Twofold level of applicability**
   Edge level and cloud/core level

II. **Technological pluralism**
   Different stream processing engines to be handled

III. **Rules' dynamicity**
   Rules follow a dynamic lifecycle



Our Proposal

Microservices architecture for an IoT platform able to offer

Adaptivity and Flexibility

Applying real time stream processing both on edge- and core-level

Hybrid Cloud Approach

Dynamicity

Handling stream processing rules as dynamically allocable, composable and relocatable resources

Portable rules model

Defining rules independently from the underlying stream processing engines

Microservices based on Java OSGi

Remarks

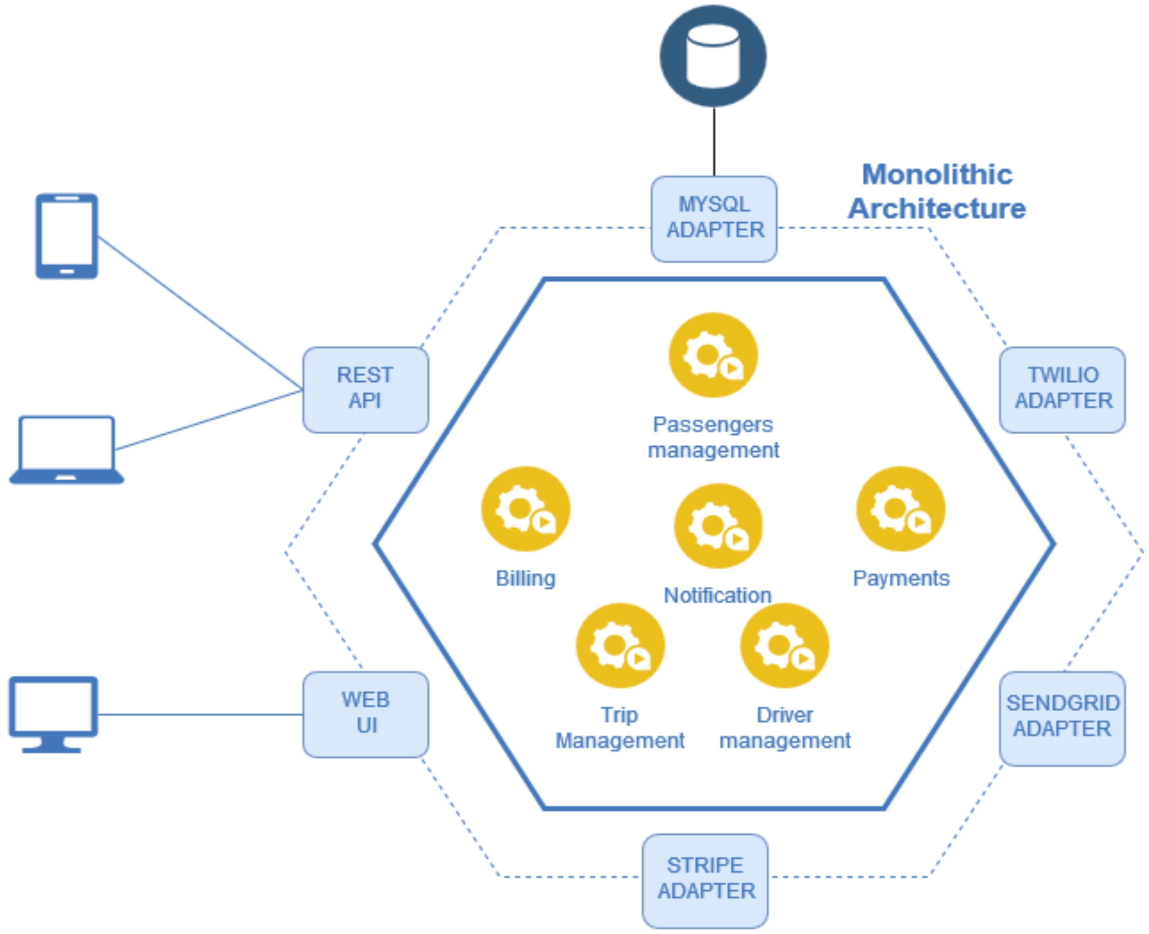
Innovative aspect

- Usually, IoT platforms offer a rich language or library for defining stream processing rules
- Our proposal restricts the query language to a predefined set of rule templates in favor of a much more flexible and dynamic deployment model

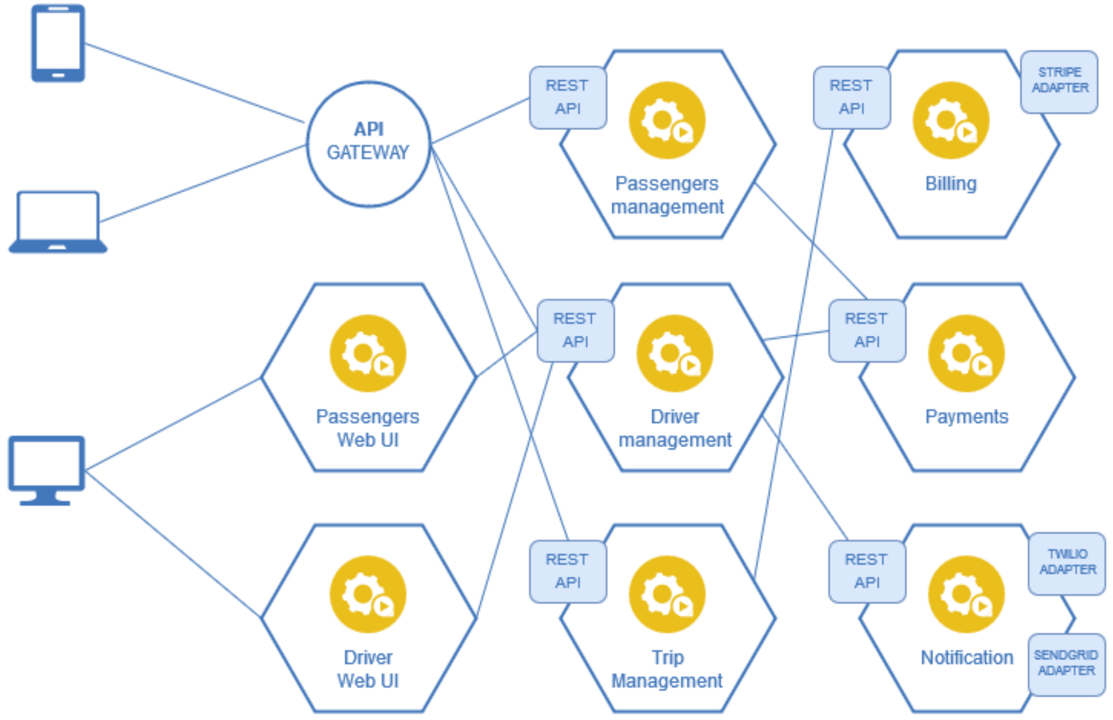
Why Microservices?

- Microservices are now the de facto standard adopted for implementing any software platform
- Senseioty platform by FlairBit
- Microservices offer an interesting level of flexibility and dynamicity

What are Microservices?



Monolithic Architecture



Microservices Architecture

Main features

Functionalities are implemented as **independent** and **autonomous services**

Services are **loosely coupled**, **replaceable** and **composable**

Services are **independently deployable and scalable**

Data management and communication mechanisms are completely **decentralized**

What is OSGi?

"OSGi technology is a set of specifications that define a dynamic component system for Java"

By OSGi Alliance

The OSGi technology is composed by two important parts

I. **The OSGi framework**

A collaborative software environment, where applications are composed of several components packaged in modules, called bundles

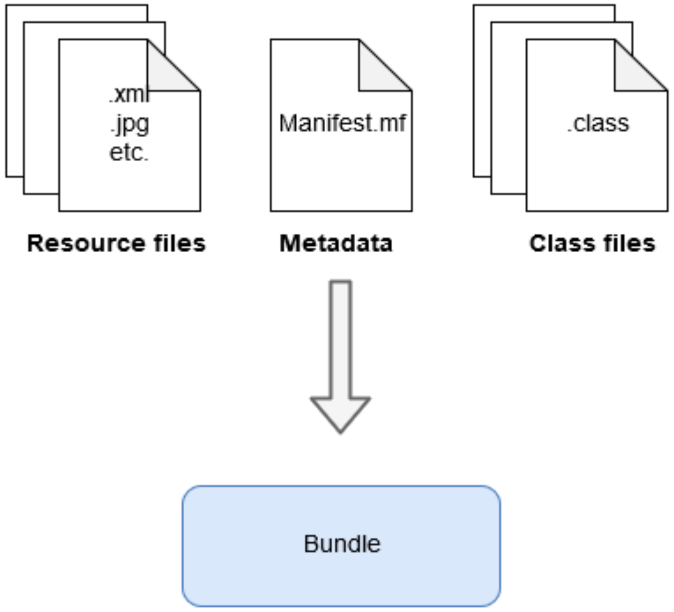
II. **The OSGI standard services**

They offers some reusable common APIs (e.g. Logging service)

Module layer

It defines the concept of bundle and how a bundle can import and export code

A **bundle** is a standard JAR file enriched by some metadata contained in a manifest

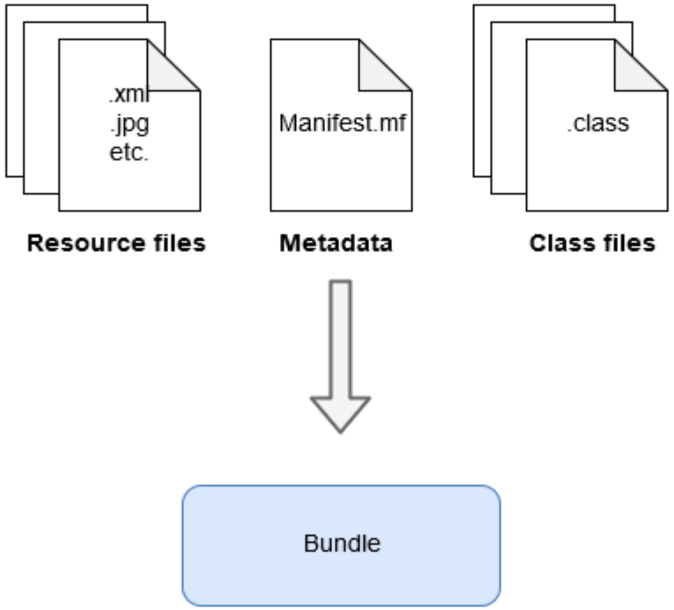



```
Manifest-Version: 1.0
Created-By: 1.2 (Oracle)
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.api
BundleVersion: 1.0.0.SNAPSHOT
Bundle-Name: Simple API
Export-Package: org.foo.api
Import-Package: javax.swing,org.foo.api
Bundle-License: http://www.example.org
Bundle-ClassPath: .,other-classes/,embedded.jar
```

Module layer

It defines the concept of bundle an

A **bundle** is a standard JAR file enriched by some metadata contained in a manifest



Code-visibility metadata:

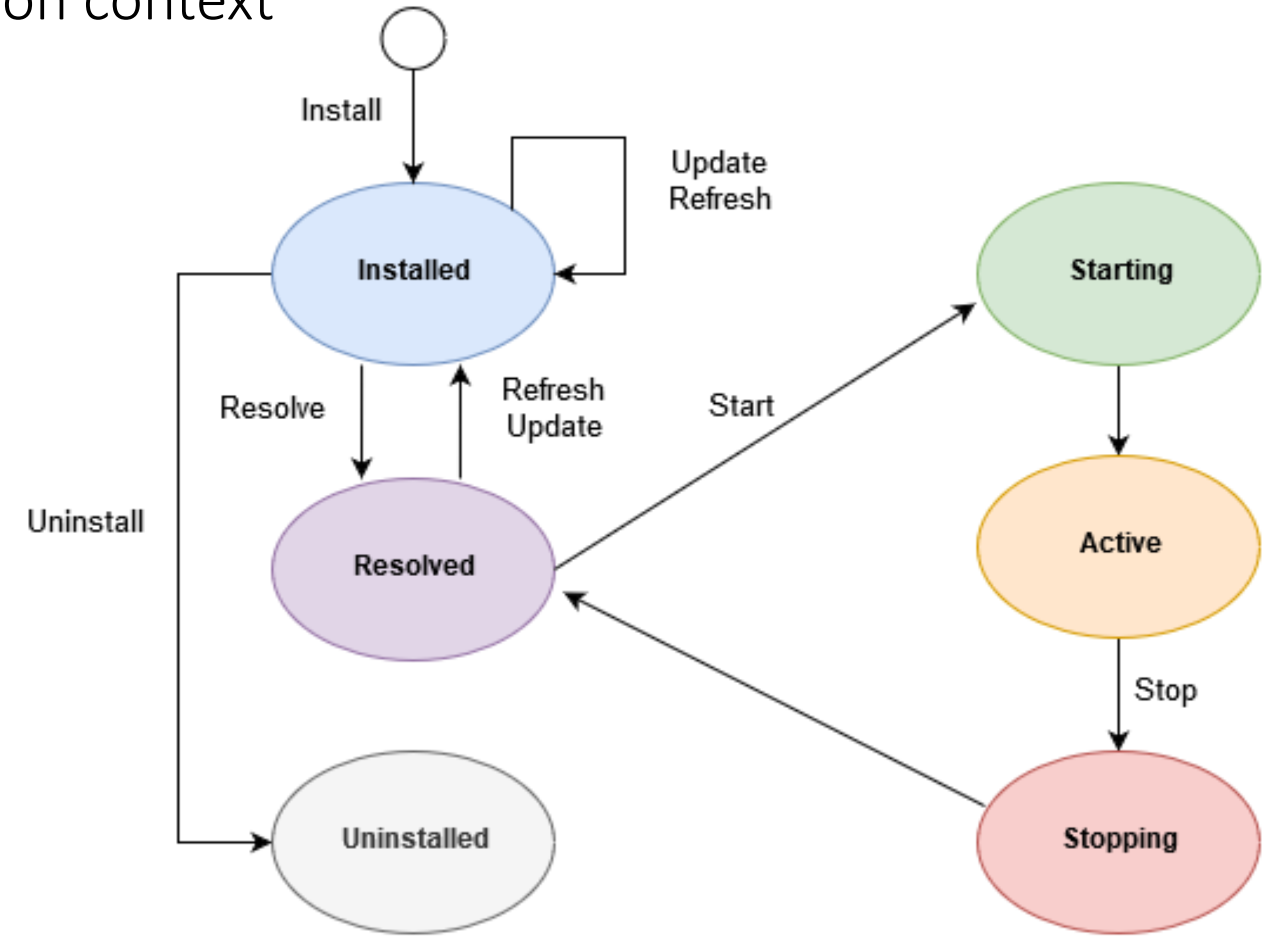
- **Internal bundle class path**: the code forming the bundle (*Bundle ClassPath* header)

- **Exported internal code**: explicitly exposed code from the bundle class path for sharing with other bundles (*Export-Package* header)

- **Imported external code**: external code on which the bundle class path code depends (*Import-Package* header)

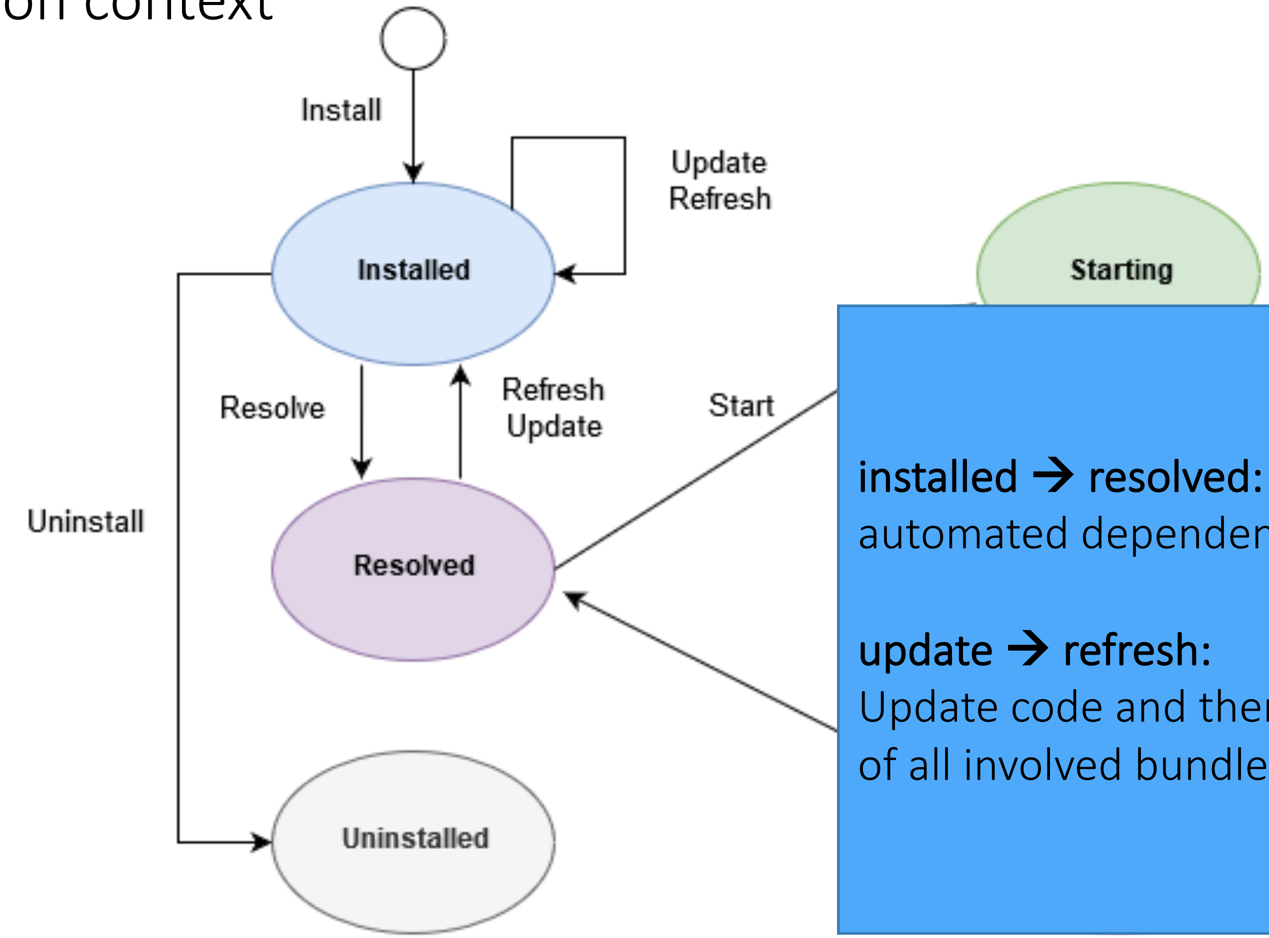
Life-cycle layer

It defines the bundle life-cycle operations and how bundles gain access to their execution context



Life-cycle layer

It defines the bundle life-cycle operations and how bundles gain access to their execution context



installed → resolved:
automated dependency resolution

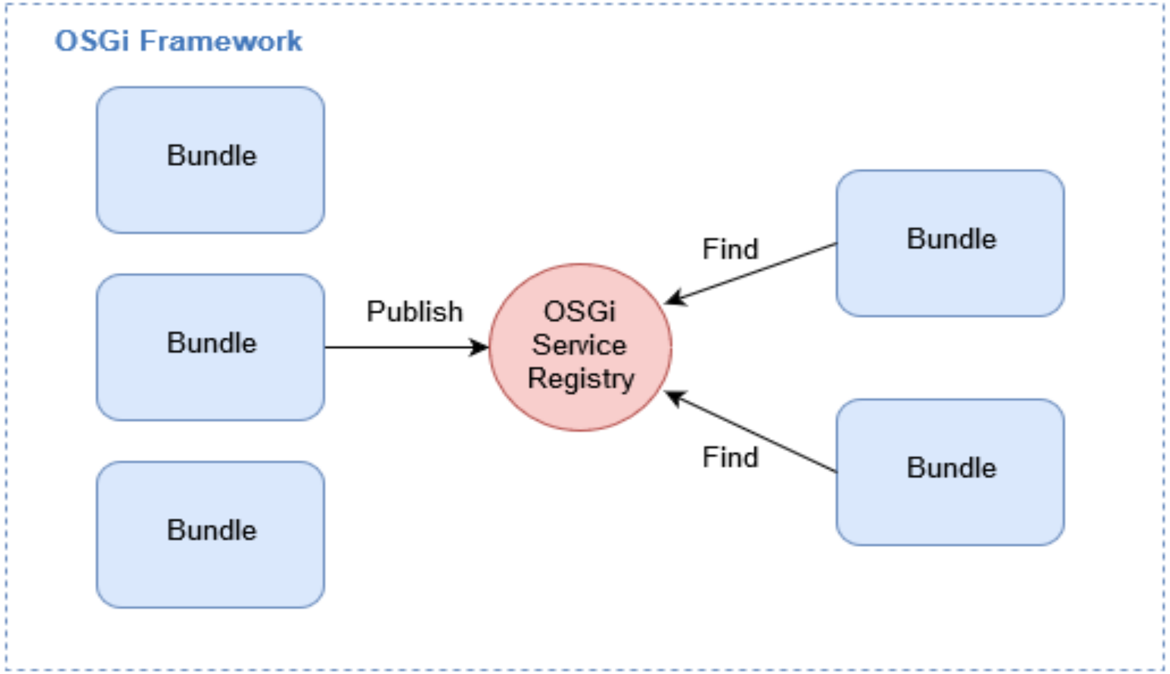
update → refresh:
Update code and then refresh dependencies of all involved bundles

Service layer

Dynamic collaborative model, where bundles communicate locally through services

A **service** consists in a Plain Old Java Objects (POJOs) that is registered under one or more Java interfaces with the OSGi service registry

- Less coupling between the provider and consumer
- Support for multiple and interchangeable implementations
- Clear highlighting of dependencies
- More emphasis on interfaces



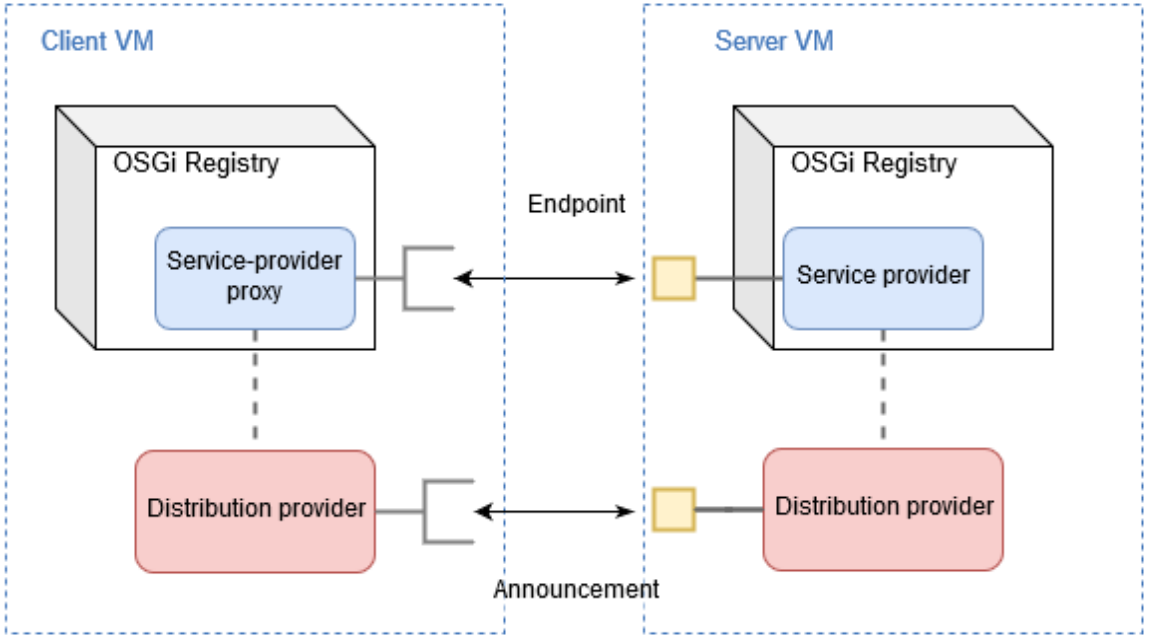
Remote Services

Set of service properties that can be attached to OSGi services in order to indicate that they should be made available remotely

- **Very flexible.** Services are exported independently from the communication protocols

- Use of intents and configurations

- The distribution provider bundles transparently manages the remote communication



OSGi and Microservices

OSGi is able to enforce and enrich some properties of the microservices architecture

Flexible granularity of service level
Combination of microservices and nanoservices

Built-in dynamic nature
Dynamicity-aware microservices

Flexibility with respect to service decomposition
OSGi Remote Services offers a flexible approach for defining the microservices boundaries

Designed for the Java Platform

OSGi is the perfect booster for those dynamic and flexible features that we were looking for

Reference Architecture

Goals

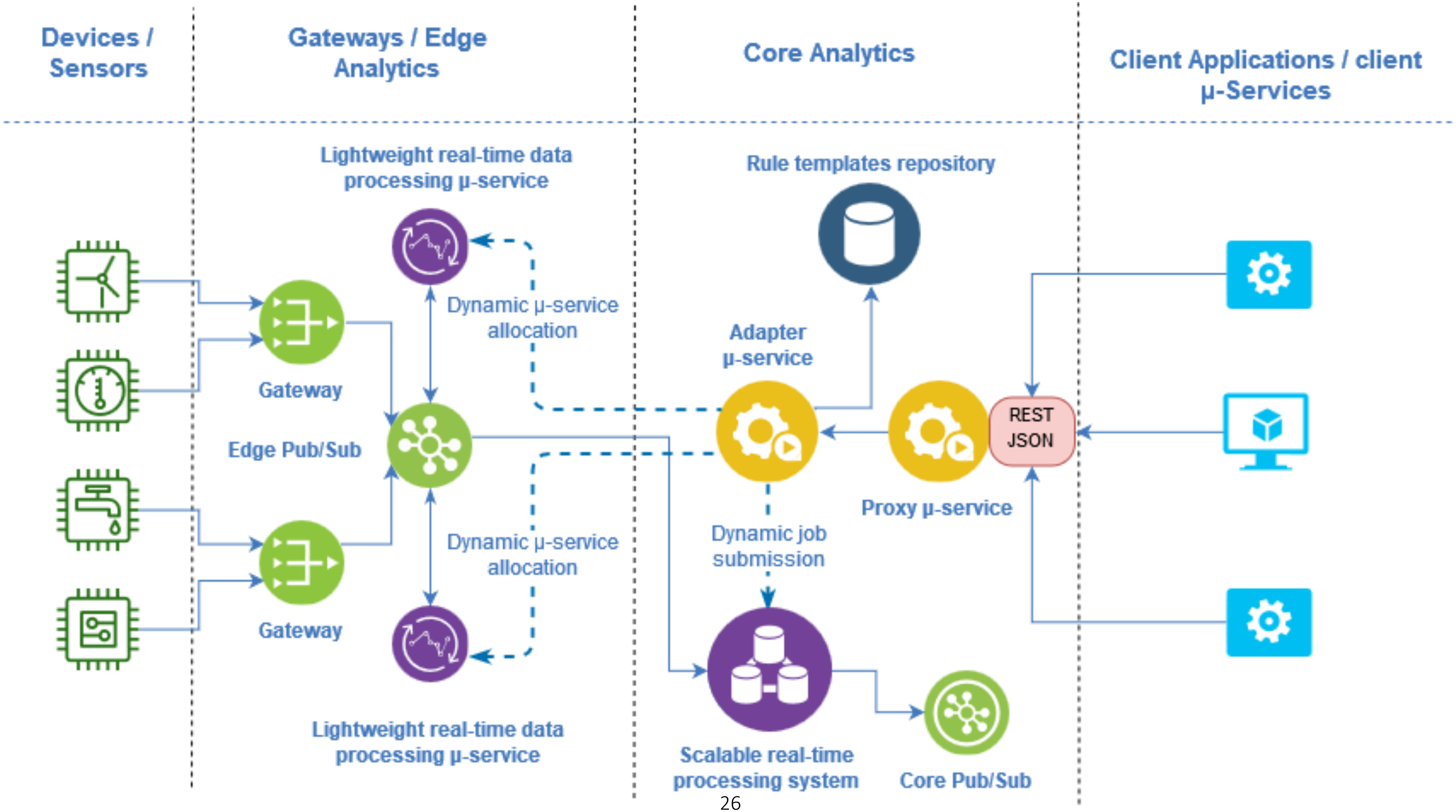
Integrating real time stream processing capabilities in an IoT platform offering

- Adaptivity and Flexibility in a Hybrid Cloud Approach
- Stream processing rules as resources
- Portable rules model

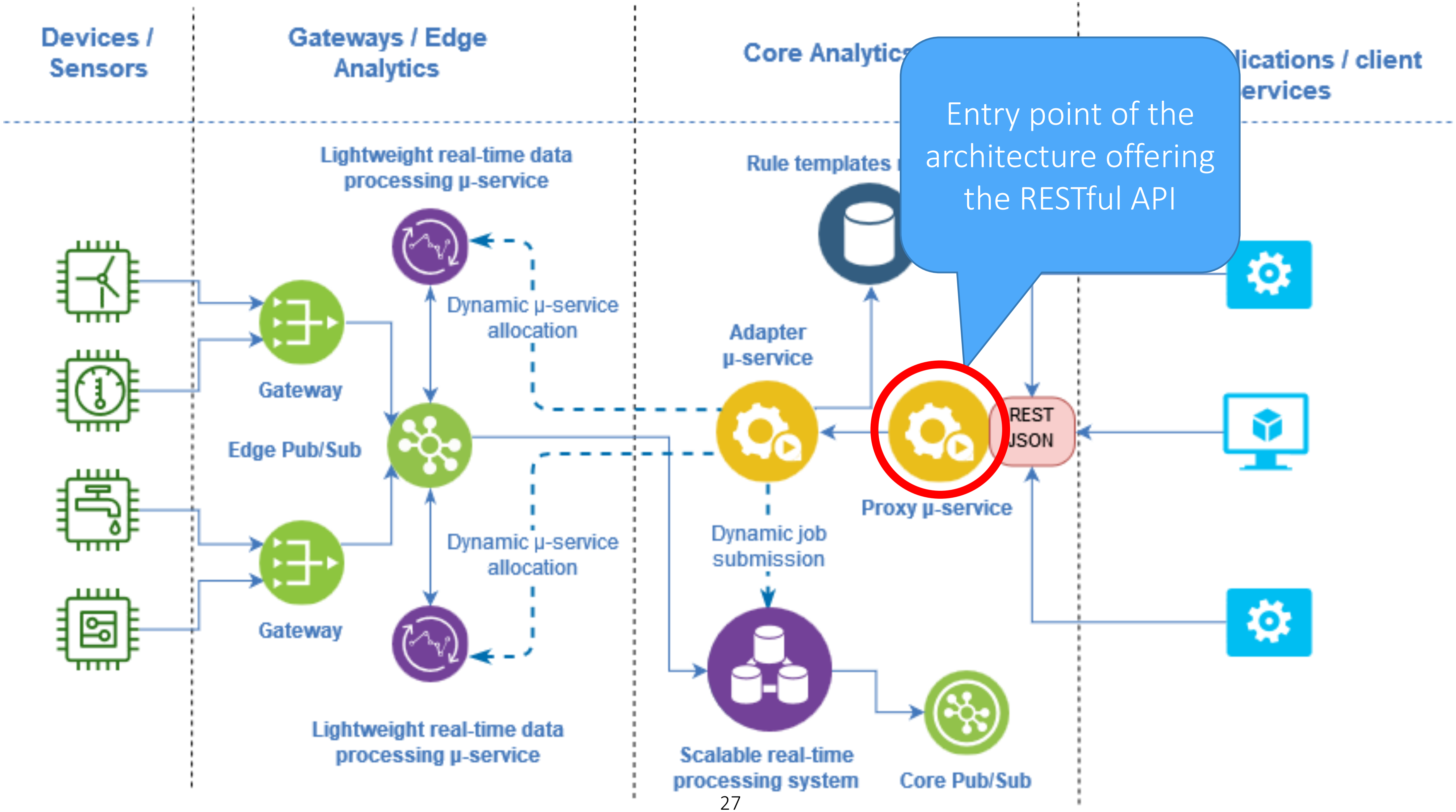
The result is a microservices architecture where these functionalities are offered as a RESTful API

- Installing and uninstalling rules
- Starting and stopping the execution of rules
- Moving the rule execution between different runtimes

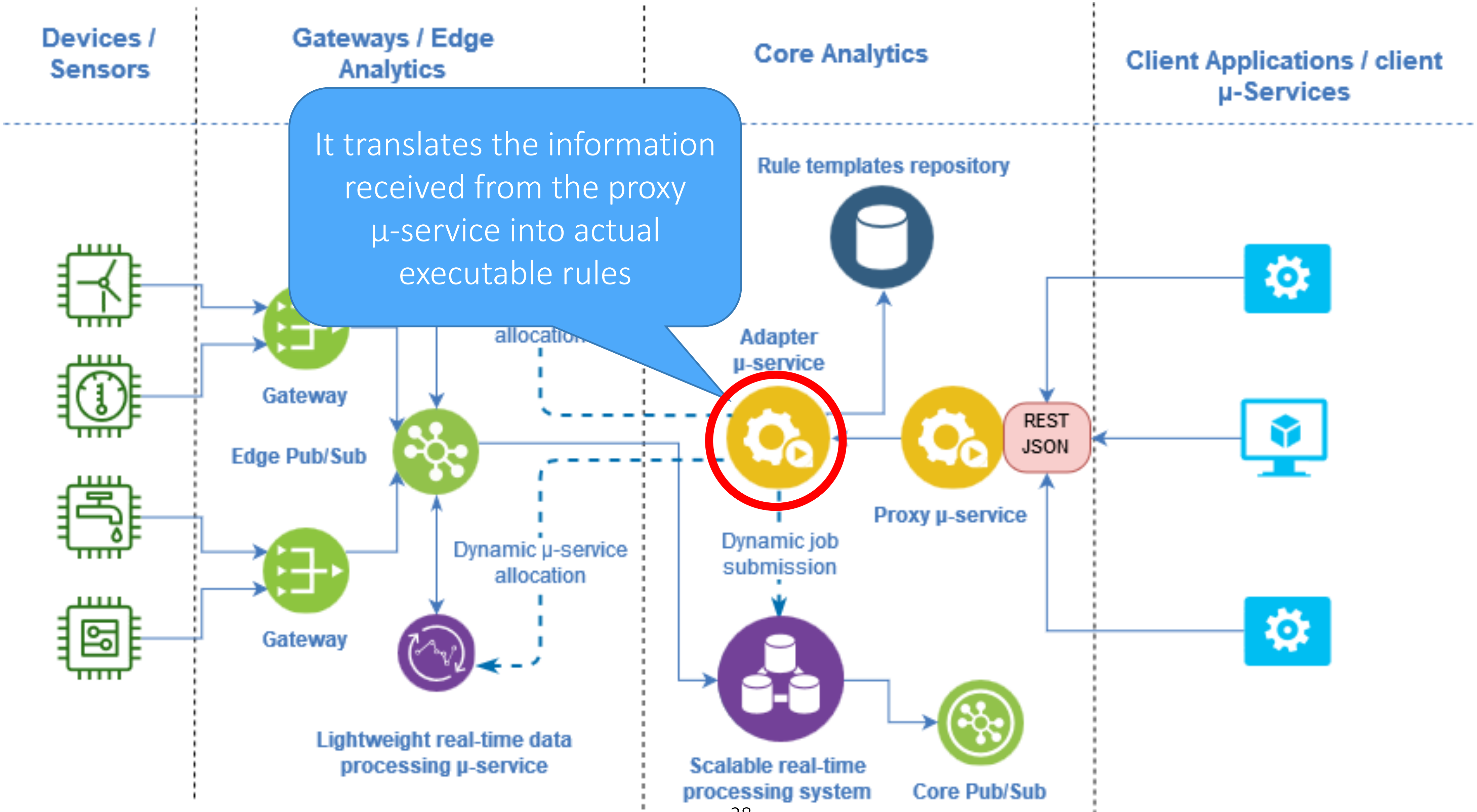
Proposed Architecture



Proposed Architecture

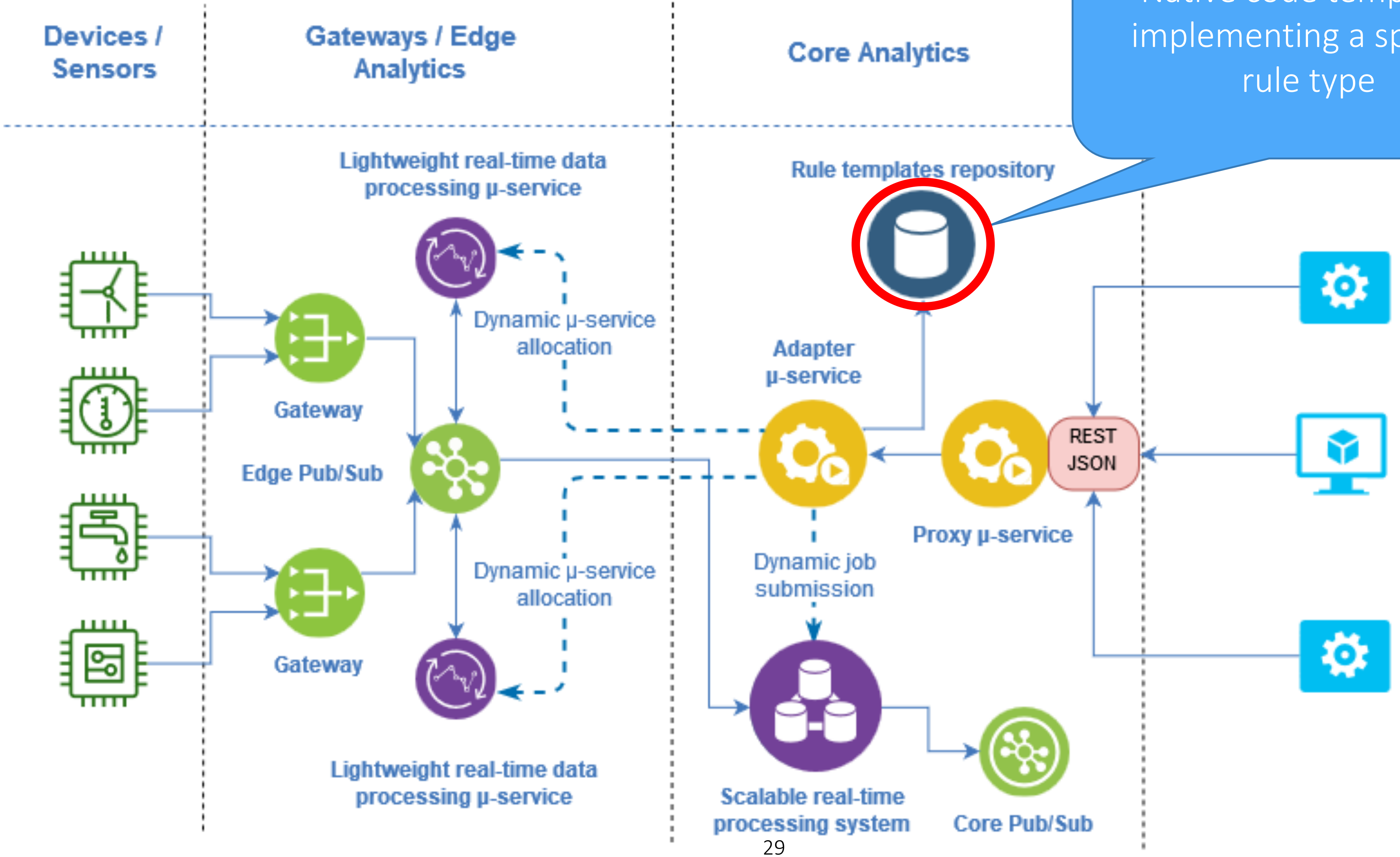


Proposed Architecture

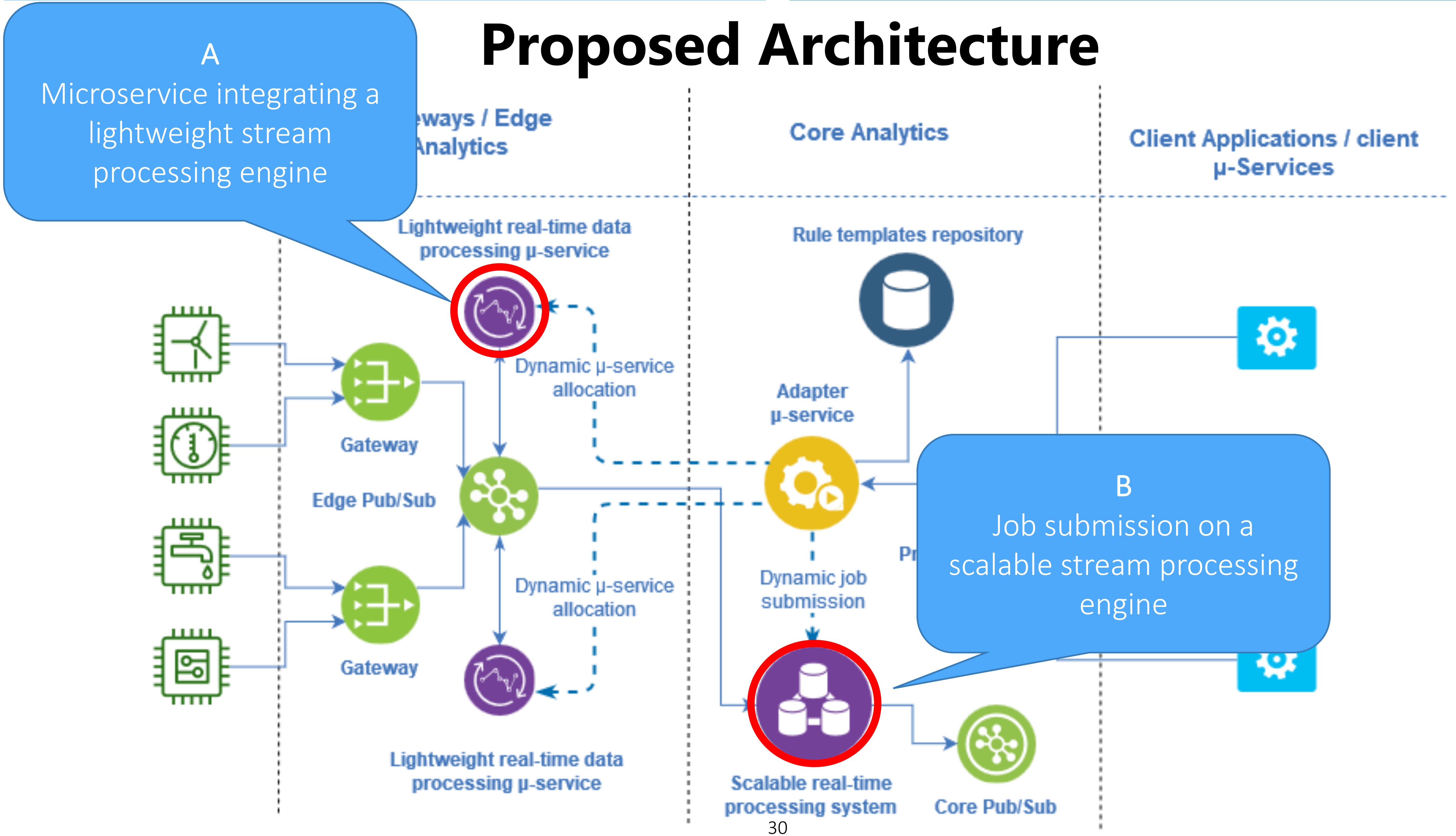


Proposed Architecture

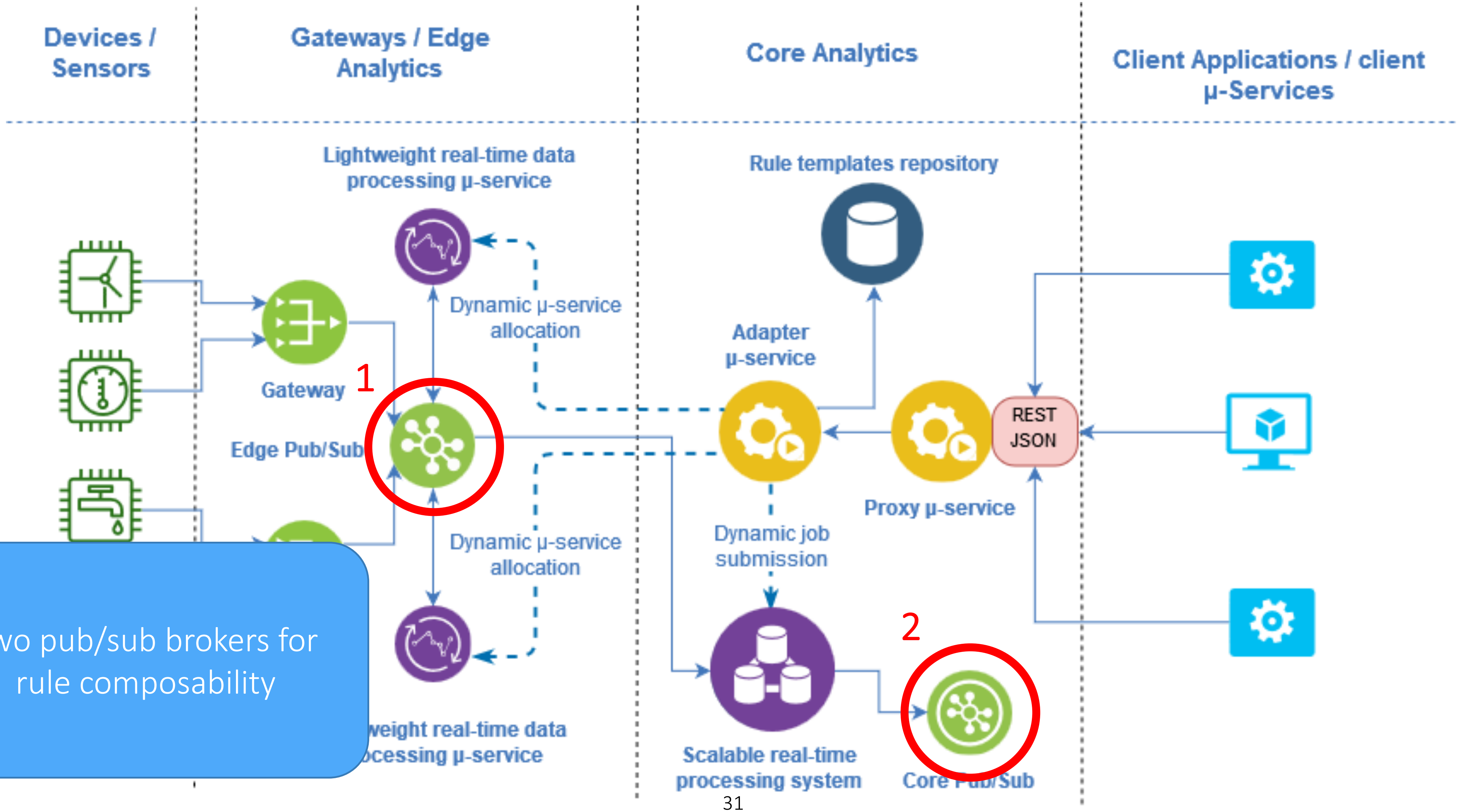
Native code templates implementing a specific rule type



Proposed Architecture



Proposed Architecture



Two pub/sub brokers for rule composability

The proxy µ-service

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

```
// 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>
}
```

```
// 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>
      }
   }
}
```

Rules' expressive power

Ideally, we would like to support any kind of rule expressible with a standard query stream language (e.g. Stanford CQL)

In practice, it is extremely complicated. It requires to implement a query compiler able to validate an arbitrary query and to compile and translate it to the model or language of the underlying stream processing engines

Our solution

Providing the expressive power for supporting the rules most commonly used in IoT scenarios

Set of predefined and composable templates

- Filtering query

`SELECT * FROM inputEvents WHERE field  > threshold`

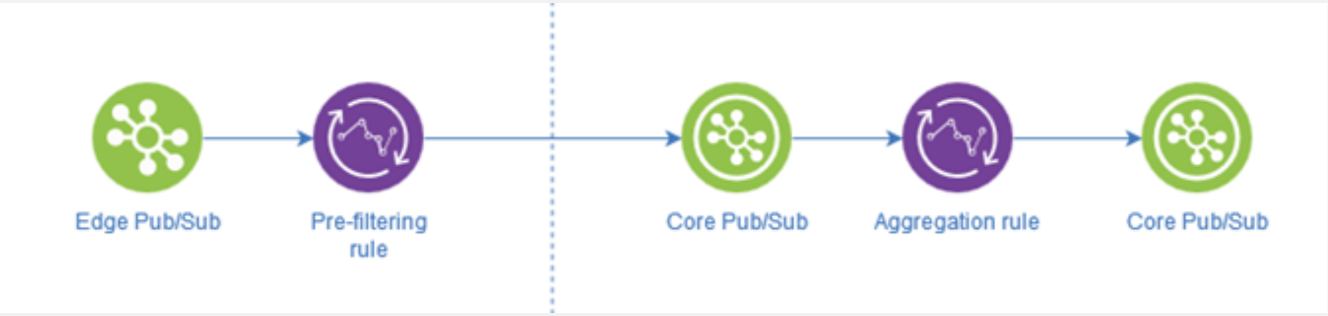
- Aggregation query over a window

`SELECT SUM(field) FROM inputEvents[5 s]`

- Joining query between two streams over windows

```
SELECT field1 field2 FROM stream1[1 m] JOIN
stream2[1 m] ON stream1.field3 = stream2.field
```

- Composability using pub-sub brokers

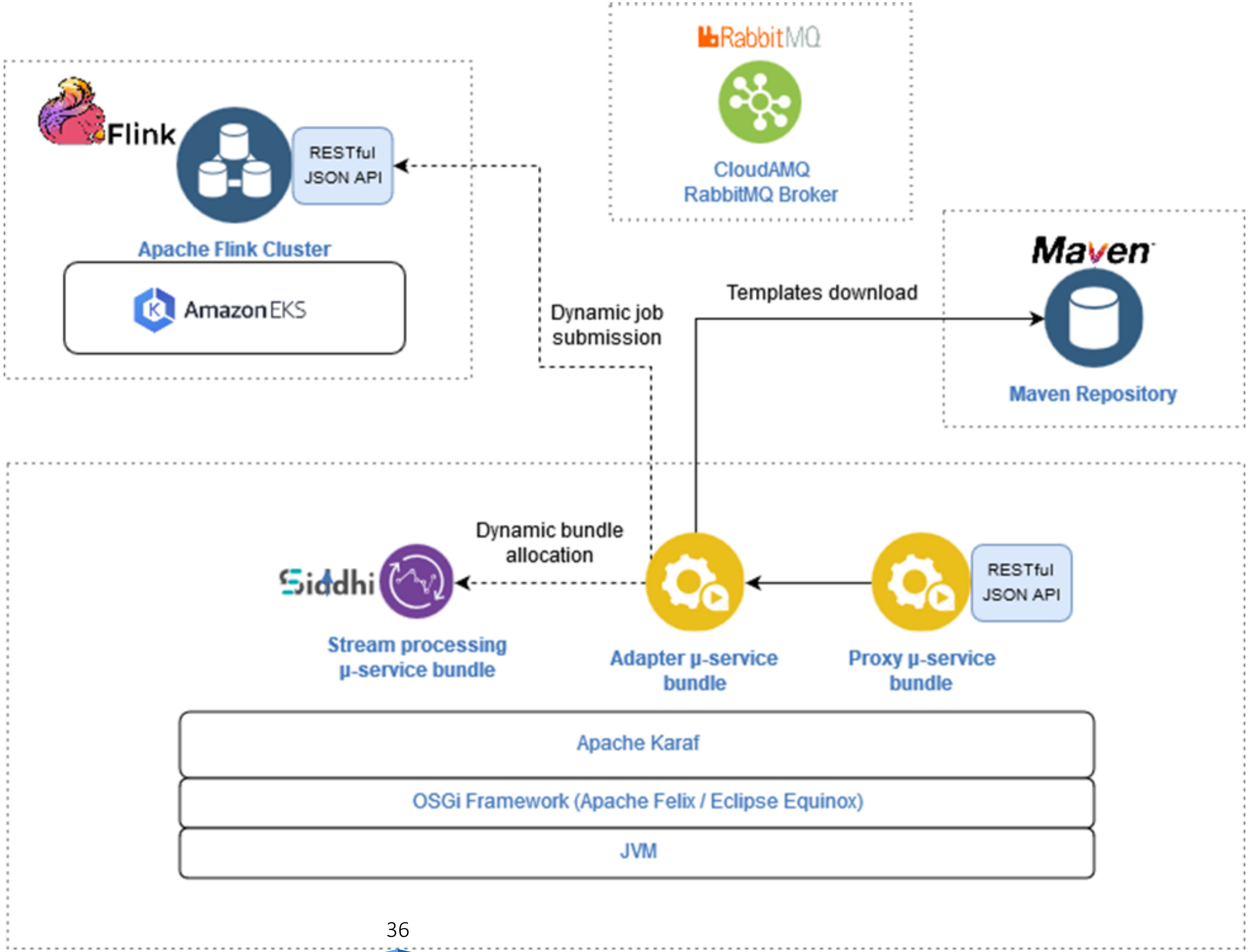


Prototype implementation

Result of an internship experience with FlairBit

- Technology exploration phase
- Technology selection phase
  Siddhi and Apache Flink
- Prototype implementation phase
  Preliminary implementation and
  PoC for extending Senseioty

Prototype overview



What about other IoT platforms?

AWS IoT Greengrass
AWS Lambda locally executed on edge devices

Azure Stream Analytics on IoT Edge
Azure Stream Analytics jobs executed on edge devices

- ✔ More expressive power
- ✖ Bound to one stream processing engine
- ✖ No dynamic allocation and relocation back and forth between the edge-level and cloud-level

Google Cloud IoT with Apache Beam SDK
Unified development model for defining and executing data processing pipelines

- ✔ Support of several engines
- ✖ No edge analytics

Future works

- Investigating possible solutions for simplifying the rules' definition
- Integrating in the architecture the monitoring μ-service introduced by the smart industry example
- Improving the prototype implementation
- Applying in the prototype the data access policies offered by Senseioty