TOWARDS SMART CITY CONCEPT: EDGE-CLOUD IOT SOLUTIONS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL,

ELECTRONIC AND TELECOMMUNICATIONS ENGINEERING AND

NAVAL ARCHITECTURE

AND THE COMMITTEE ON GRADUATE STUDIES

OF THE UNIVERSITY OF GENOA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Ahmad Hassan Kobeissi

February 2020

This page is left empty.

Abstract

Since the term was coined by Kevin Ashton in 1999, the Internet of Things (IoT) did not gain considerable popularity until 2010 where it became a strategic priority for governments, companies, and research centers. Despite this large-scale interest, IoT only reached mass markets in 2014 in the form of wearable devices and fitness trackers, home automation, industrial asset monitoring, and smart energy meters. The ‘things’ refer to sensors and other smart devices with the ability to monitor an object’s state, or even control it using actuators. Ashton envisaged that when such sensors and smart devices were on a ubiquitous network – the Internet – they would have far more value. Trending data-centric technologies in the IoT involve security and data governance, infrastructure (edge & cloud analytics), data processing, advanced analytics, and data integrating and messaging. These technologies are supported by cloud computing service models that include three major layers – Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). Of the three, IaaS is the foundation while SaaS is the top layer functioning off both PaaS and IaaS. Interestingly enough, although SaaS is normally represented in graphics as the smallest layer of Cloud infrastructure, it is anything but. The IaaS layer of Cloud Computing is comprised of all the hardware needed to make Cloud Computing possible. The PaaS layer of the Cloud is a framework for developers that they can build upon and use to create customized applications. Built on top of both IaaS and PaaS, Software as a Service provides applications, programs, software, and web tools to the public for free or for a price.

By the year 2020, trillions of gigabytes of data will be generated through the Internet of Things. This is no doubt difficult to comprehend easily. However, with the growing number of connected devices it is not surprising that by 2020, more than ten billion sensors and devices will be connected to the internet. Furthermore, all of these devices will gather, analyze, share, and transmit data in real-time. Hence, without the data, IoT devices would not hold the functionalities and capabilities which have made them achieve so much worldwide attention. If organizations are not in a position to somehow ingest, process and analyze these data, then it becomes worthless, and the IoT project will be considered a failure. Unlike a traditional IT system, IoT systems are cyber-physical systems involving both humans and machines as end-users. Their interaction forms a complex web of M2M (Machine to Machine) and H2M (Human to Machine) transactions. Right from device firmware, to network interfaces, extending all the way to business logic defined in cloud application and user app, software remains the most critical driver in IoT. Similarly, Edge computing presents great opportunities to achieve ubiquitous computation in the Internet ecosystem. It is proposed to overcome the intrinsic challenges of computing on the cloud side. Edge computing offers to gather more sensory data, reducing the response time, freeing up network bandwidth, and ultimately reducing the workload on the cloud.

In the effort to elevate support for technologies that are directed toward IoT in smart cities concept, support for developers and service providers is critical especially regarding fast and feasible deployment of IoT solutions and assets. To that end, I focused during my research on ways and methods to exploit generic IoT solutions; Application Programming Interfaces (APIs) and edge engines. In this book, I present Atmosphere, a novel edge-to-cloud solution for supporting development and deployment by IoT developers and service providers. Atmosphere cloud is a SaaS deployment-ready model, while Atmosphere edge is a lightweight edge engine for IoT device management.

Needless to say, testing the various software components is essential to ensure a safe and reliable IoT system. The solutions I contributed to were tested in multiple projects of varying volumes and challenges. In some projects, using the generic concept was straight forward, while in others, where the structure of the IoT data was complicated and restrictions were established by the partners, the integration was challenging.

Acknowledgments

First and foremost, I want to thank my supervisor Prof. Riccardo Berta. It has been an honor to be his Ph.D. student. He taught me, both consciously and unconsciously, how good research in computer engineering is done. I appreciate all his contributions of time, ideas, and funding to make my Ph.D. experience productive and stimulating. The joy and enthusiasm that he possesses for his research were contagious and motivational for me, even during tough times in the Ph.D. pursuit. I am also thankful for the excellent example he provided as a successful professor.

The members of the Elios lab have contributed immensely to my personal and professional time at the University of Genova. The group has been a source of friendships as well as good advice and collaboration. I am especially grateful for professors Alessandro De Gloria and Francesco Bellotti for their invaluable mentorship and support through the years. I would like to acknowledge my former Ph.D. colleague Nikesh Bajaj for his friendship and substantial assistance to my work. I very much appreciated his willingness to assist me to navigate through programming situations. Other past and present lab members that I have had the pleasure to work with or alongside are Ivan Carmosino, Rana Massoud, Alessio Sedotti, Giacomo Lanza, and the numerous students who have come through the lab. A special thanks to Hassan Mokalled for the brotherly colleagueship during my three years in the Ph.D.

In regards to the Lebanese University, I thank professors Hussein Chibli, Yasser Mohanna, and Ali Ibrahim. They gave me helpful advice on the research process especially with electronics early on. Ali Ibrahim shared with me a lot of tips he picked up during his research at both the University of Genova and Lebanese University. I would also like to acknowledge the Mecrl lab and other facilities at the Lebanese University in Beirut, Lebanon where I spent a couple of months every year in research.

In my work on Health at Home project and the subsequent updates to the API, I thank Loreto Pescosolido (at IIT) for his collaboration and enthusiasm on novel additions to the resource model of Atmosphere’s API. In my field tests on Fabric project in Susa valley, I thank the following people for helpful discussions and cooperation: Vincenzo Cirimele, Michela Diana, Nadim El Sayed, Mojtaba Khalilian, Jacopo Colussi, Paolo Guglielmi, and the staff at the MotorOasi Piemonte facility. For the Atmosphere deployment in L3Pilot, I thank Eduardo Arnold and Sajjad Mozaffari at the University of Warwick for their outstanding cooperation in adjusting the user interface along with API modifications and requirements.

I gratefully acknowledge the funding sources that made my Ph.D. work possible. I was funded by the University of Genova for a 3-year Ph.D. program. My work was also supported by the Lebanese University through minor conference funding contributions.

My time at the University of Genova was enriched by the oversight of my Ph.D. coordinator Prof. Mario Marchese whom I thank very much for his honest guidance. I extend my appreciation to the staff at the department of electrical, electronic and telecommunications engineering and naval architecture (DITEN).

Finally, I would like to thank my family for all their love and encouragement. For the presence of my brother Mahmoud and his valuable bureaucratic assistance to me. For my cousin Tania and her family for their support abroad. And most of all for my supportive, encouraging, and patient parents Hassan and Ibtissam whose faithful support during my entire life is so appreciated. Thank you.

Contents

List of Tables

List of Figures

List of Scripts

Nomenclature

Abbreviations

IoT – Internet of Things

API – Application Programmable Interface

AWS – Amazon Web Services

DB – Database

REST – REpresentational State Transfer

IP – Internet Protocol

IaaS – Infrastructure as a Service

PaaS – Platform as a Service

SaaS – Software as a Service

TDD – Test-Driven Development

BDD – Behaviour-Driven Development

k-NN – k-Nearest Neighbor

GAAS – Grid Alignment Assistant System

FABRIC – Fe**a**si**b**ility analysis and Development of on-**r**oad charg**i**ng solutions for future electric vehi**c**les

DWC – Dynamic Wireless Charging

EV – Electric Vehicle

E-Mobility – Electro Mobility

OBU – On-Board Unit

VMU – Vehicle Management Unit

4G-LTE – Fourth Generation Long Term Evolution mobile network technology

CU – Control Unit

L3Pilot – Piloting Automated Driving on European Roads

SI – Scenario Instance

DSI – Driving SI

PI – Performance Indicator

AD – Automated Driving

SAE – Society of Automotive Engineers

FESTA – Field opErational teST support Action

OS – Operating System

CPU – Central Processing Unit

RAM – Random Access Memory

SRAM – Static RAM

GPIO – General-Purpose Input/Output

PIR – Passive Infra-Red

H@H – Heath at Home

NodeJs – JavaScript runtime environment for server-side scripting

ExpressJs – Web application framework for Node.js

NPM – Node Package Manager

JSON – JavaScript Object Notation

BSON – Binary JSON

JWT – JSON Web Token

SSH – Secure Shell

HTTP – Hypertext Transport Protocol

HTTPS – HTTP Secure

CA – Certificate Authority

GUID – Globally Unique Identifier

HMAC – Hash-based Message Authentication Code

RSA – Rivest-Shamir-Adleman algorithm

URI – Uniform Resource Identifier

UI – User Interface

GUI – Graphical User Interface

LTS – Long Term Support

MQTT – Message Queuing Telemetry Transport

Chapter 1 – Introduction

The development and deployment process of IoT services (at the edge, middleware and cloud sides) has become a lengthy process that requires considerable time, resources, and effort. We need to have a generic solution that can be used by different service providers so that they can deploy their services earlier while saving time and resources on the development.

Looking at IoT applications from a generic perspective we spot challenges in the huge versatility of IoT, where personal or industrial implementations can be approached in various different manners. These concerns raise some serious questions. What are the required architectural styles, programming patterns, and hardware tools that can be employed to facilitate the development of IoT applications? Which is the most suitable approach to provide a comprehensive solution for the various IoT uses cases? How to configure an IoT infrastructure for fast deployment by service providers and users? Moreover, what are the main indicators for testing the applicability of such infrastructure?

In order to investigate an answer to the research questions, I performed a series of experiments dealing with a plethora of IoT applications in different domains. In these experiments, I developed the necessary IoT capabilities (hardware and software) and inspected the methods to improve the developed tools and achieve a wider usage for these tools (separate or combined) by similar IoT applications.

Cloud technology is a vast area of opportunities for data management of IoT applications [1]. More often than not, cloud services are exploited and tailored for industrial [2], domestic [3], and health applications [4]. Due to the nature and characteristics of these data [5], from accumulating high volumes to growth rate and its variety, these applications are getting the big data treatment. Some solutions are exploiting dynamic cloud-edge resource management [6, 7] and collaborative data storage for IoT analytics. To enhance data management on the cloud, [8] exemplifies novel approaches for composing and interoperating cloud solutions for mobile data. In [9], data storage, management, and analysis are served in a multi-cloud architecture for facilitating information sharing in scientific communities. In contrast, a cloud storage hub [10] is designed to contribute to the integration of different data sources through a mapping and ruling system to access data conveniently through a single API.

Various IoT frameworks are currently available on the market that focuses on different types of IoT services. The Amazon Web Services for the Internet of Things (AWS IoT) [11] is a cloud service. Its main IoT architecture modules concern data management, device connectivity and control, and analytics and event detection. A stack of functions is available on the back-end: DynamoDB, Kinesis, Lambda, S3, SNS, SQS, etc. Since AWS is a cloud service provider, multiple data processing services are already integrated. Likewise, the IoT Applications component enables the connection of further applications to the platform. AWS IoT does not distinguish among sensors, actuators, and devices, as it focuses on the concept of things. However, there are limited custom attributes for things, which challenge manageability and increases latency in [12]. The Microsoft Azure cloud platform [13] enables developers to create cloud-based programs using a Software as a Service (SaaS) commercial platform. Azure Sphere contains tools for edge SDKs [14] to enable secure edge-to-cloud connectivity. Other commercial frameworks [15], like Google Cloud and Bluemix, have a variety of IoT cloud services. Despite their differences in performance and efficiency in practice, [16] indicates long deployment times as a shared issue among such frameworks. We argue that a better-defined content structure could facilitate developers, especially for the applications dealing with measurements.

Several IoT data frameworks have been presented in the literature to deal with specific IoT application domains. Recent examples concern forensics [17], smart homes [18] and smart cities [19].

In a more general approach, [20] proposed a framework dealing with typical IoT challenges (large volume of data, different data types, rapid generating data, complicated requirements, etc.). For structured data, they propose a database management model that combines and extends multiple databases and provides unified accessing APIs. For unstructured data, the framework wraps and extends the Hadoop Distributed File System (HDFS) based on the file repository model to implement version management and multitenant data isolation. A resource configuration module supports static and dynamic data management in terms of the predefined meta-model. Thus, data resources and related services can be configured based on tenant requirements. More recently, [21] presented a functional framework that identifies the acquisition, management, processing and mining areas of IoT big data, and several associated technical modules are defined and described in terms of their key characteristics and capabilities. [22] deals particularly with IoT data storage efficiency and security, proposing a framework that keeps locally (on the edge) time-sensitive data (e.g., control information) and sends the other (e.g., monitoring data) to the cloud.

In order to support remote and cross-platform data access, several data frameworks integrate APIs implementing Representational State Transfer (REST) services [23] which provide a platform-independent HyperText Transfer Protocol (HTTP) interface (e.g., [15]).

In a similar fashion to Atmosphere, [24] proposes a framework, which supports developers in modeling smart things as web resources, exposing them through RESTful APIs and developing applications on top of them. The framework supports resource-type definition and design, general-purpose software for operations on web resources, a mapping between web resources and data sources, and programming and publishing tools.

[25] singles out four main characteristics of IoT data in cloud platforms: multisource high heterogeneity, huge scale dynamic, low-level with weak semantics, inaccuracy. These characteristics are important, as they highlight key features that should be provided by an effective IoT data framework (e.g., source characterization, variety of source data configurations/aggregation, outlier computation [26]), that we kept into account in the design of Atmosphere.

The presented frameworks and solutions from the literature expose different strategies to serve data-driven IoT applications. The available approaches, whether proprietary or open-source, are either specialized and are thus difficult to reuse, or high-level in their design model that requires significant time for development. Cost-wise, commercial solutions offer a pay-as-you-go pricing structure, in which the vendor only charges for the computing resources used by the application, usually saving customers money. However, in an IoT setup where data is continuously being circulated between edge, cloud, and user interface applications, this pricing model may be unfitted. My proposed cloud solution shows similarity to the WoT framework in [27]. Both works adopt a RESTful web resource approach and expose the accessibility and management of IoT devices for similar target users. Table I summarized the advantages and disadvantages of existing cloud solutions on the commercial and academic fronts.

TABLE I
COMMERCIAL VERSUS ACADEMIC CLOUD SOLUTIONS FOR IOT

	Advantage	**Disadvantage**
Commercial	Stack of functions is available on the back-end	Does not distinguish among sensors, actuators, and devices
	Enables connection of further applications to the platform	Limited custom attributes for things, increases latency
	Contains tools for edge SDKs	Long deployment times
Academic	Unified accessing APIs	Specialized for a specific IoT domain
	Configurable data resources and services	Limited reusability
	Time-sensitive data preservation on edge	Limited scalability

The tech giants are also commercial leaders in the IoT edge services and solutions. Recently, AWS offered serverless functions called Lambda@Edge [28] in a pay-per-computation billing scheme. It provides a stand-alone execution environment for individual functions written in Node.js, Python, Java, or C#. Content delivery through the Amazon CloudFront [29] can be customized as well as compute resources and execution time. Since AWS Lambda is serverless, the cost of execution is reduced. For this ease of execution and reduction in cost, though, the downside is losing control over the environment. While AWS Lambda functions run on Amazon Machine Instances

(AMIs), and thus use industry-standard tools for web and general development, consumers are not able to custom install packages or software on the running environment, nor do they have given control over how the instance is maintained. Furthermore, the design of AWS Lambda does have the potential to increase the design-time cost of the IoT application [30]. AWS Lambda functions are timeboxed, with a default timeout of three seconds (it is configurable up to five minutes). This means the consumers need to spend more time orchestrating and organizing their functions so that they can work in a distributed fashion on their data.

Azure IoT Edge [31] offers deployment of models - built and trained in the cloud - on the edge. In the case of intermittent connectivity, Azure IoT Edge device management automatically synchronizes the latest state of edge devices - after they reconnect - to ensure seamless operability. Azure IoT Edge aims to target image recognition and contact signaling as the main use cases in IoT applications. Despite its costly deployment [32], Azure IoT Edge is a Microsoft service, and it is only compatible with other Microsoft Azure products, such as Azure IoT Hub, Central, and cloud [33].

The deployment of IoT applications to distributed nodes is a tedious procedure. In [34], a proposed approach is presented where the IoT application can be modeled in one place, where after modeling; the different pieces of the application are annotated with location information. Based on this annotation, the application is decomposed into fragments that are deployed to the corresponding individual compute nodes, automatically generating code to remotely connect the application fragments to other application fragments on other compute nodes in the edge or in the cloud. In addressing the domain-diversity aspect in data sharing in IoT, [35] proposes a cross-domain, secure, and feasible data sharing scheme in cooperative edge computing. To ensure the data’s safety achieve data’s fine-grained access, the scheme employs CP-ABE as an encryption mechanism for data privacy.

A Fog node, denoted as “IoT Hub,” placed at the edge of multiple networks, which enhances the capabilities of the network by implementing the following functions: border router; cross-proxy; cache; and resource directory is presented in [36]. The IoT Hub enables clients to effectively discover and access resources hosted by heterogeneous Smart Objects (SOs). The benefit of the IoT Hub is its capability to hide completely the diverse nature of SOs, in terms of hardware, wired/wireless communication protocol, and application-layer protocol used, to clients, which can interact with them using uniform interfaces and without requiring any prior configuration. An implementation of the IoT Hub to evaluate its practical feasibility and performance in a real-world IoT testbed comprising several heterogeneous devices resulted in positive results regarding deployment on low-end devices, such as the Raspberry Pi. Steel [37], is a high-level abstraction framework designed specifically for building complex data processing applications in the emerging edge-cloud environment. It is an extensible framework where common but crucial optimizations for the edge (e.g., placement, load balancing, communication) can be built as pluggable and configurable modules. Steel is claimed to have reduced the initial deployment effort by 2.6x on average. Several lightweight solutions on the edge [38, 39, & 40] propose novel approaches to edge computing, from exposing RESTful web services and relying on specific portions of commercial IoT services such as IBM Watson IoT platform, to establishing multi-source feedback information fusion towards alleviating the concerns of numerous IoT users.

TABLE II
COMMERCIAL VERSUS ACADEMIC EDGE SOLUTIONS FOR IOT

	Advantage	**Disadvantage**
Commercial	Easily programmable	Increased design-time cost
	Complementing same-vendor cloud services	Limited customizability
	Cross-platform support	Limited support for third-party services
Academic	Cost reduction	Fit for specific edge configurations
	Optimized operability	Complex programming
	Provides feasible data sharing schemes	Reliance on commercial services

We have seen the available IoT edge solutions in the tech market as well as in the latest academic research. Solutions found in the literature present significant features such as cost reduction [34, 37], optimizing operability [36], or providing feasible data sharing schemes [35]. On the other hand, as listed in Table II, each solution has its own limitations and/or drawbacks with respect to the general industrial IoT application. These solutions are best suited and applicable in their targeted scenarios such as complex data structure, information fusion, and real-time analytics. Commercial solutions [28, 31] suffers from design-time cost and lack of customizability and third-party compatibility. Our proposed concept of the edge solution aims to satisfy the widest range of IoT device management features in one abstract service while focusing on reduced design-time cost and easy deployment.

IoT applications are demanding for developers when they try to build a whole customized solution. The choices are, as we saw in the state-of-the-art, start from scratch, buy one or more commercial solutions (train and get used to those), or use an open-source project. What I learned from the experience of mine and other developers in this field is that there is a substantial common ground that can be exploited and thus make it more feasible for IoT developers to realize their applications. All smart things stream measurements with attributes (user, feature, device, service, provider, etc.) to the cloud or local buffer.

APIs are the communication and data sharing mechanisms between two different applications or systems. APIs are the natural evolution of web services; they facilitate dynamic information sharing. Moreover, APIs have more lightweight architectures, which makes them more suitable for limited bandwidth devices such as IoT devices. When building APIs, some considerations must be dealt with such as latency, partial failure, and security concerns. Latency is the amount of time it takes a request to return a response. Latency increases when client and server are running on different machines. Modern APIs are developed to keep the latency at minimum figures (sub milliseconds). Partial failure occurs when a component of the server or network that is supposed to send data back to the calling client fails to respond. It happens when a network is down or the server is overloaded by requests and cannot respond. The more web services used in a single application, the bigger the risk of experiencing a partial failure. Thus, a developer has to balance the number of web services with the requirements of the API. Considering API security, two common procedures must be followed: authentication and authorization. Authentication is about validating the identity of the client that is attempting to call a web service that accesses secure data. The identity is usually validated with user credentials (username/ password). Authorization determines the level of a client's access. A well-designed API must incorporate a sturdy and well-structured authentication/authorization model.

In this context, we (myself and supervisors) have developed Atmosphere, an open-source, provider-independent, and comprehensive data management framework. Atmosphere is easily configurable for different IoT applications, providing an abstract interface towards measurements. We focus on the concept of measurement, as this type of data is very common in IoT, which makes the process of abstraction needed in order to support efficient application development in a variety of domains and operational contexts easier and more effective.

The contributions of this work are evident in Atmosphere's main characteristics, where it combines features from both commercial and academic solutions and package these features in a general-purpose architecture that suits different IoT applications and domains. On the cloud, atmosphere provides high scalability, ease-of-deployment, preservations of data integrity, large reusability, Standardized IoT terminology in a domain-independent generalizability and applicability, and user management. In addition to those, edge features such as versatility, portability, real-time reactivity, simplicity in function support, and high customizability while being lightweight form the characteristics that distinguish Atmosphere among its alternatives.

The remainder of the book is organized as follows. Chapter 2 describes Atmosphere with its two main components, the cloud API and the edge engine. Chapter 3 describes the experimental use cases of Atmosphere in two automotive European projects, one e-Health Italian project, and an in-lab smart home simulation. These experiments are discussed thereafter to reflect on the results and shed light on their significance to the research project. Finally, chapter 4 concludes the topic with a preview of the entire thesis.

Chapter 2 – Atmosphere

Section I – Overall Architecture

Our effort, extracted from analyzing the state-of-the-art, aimed first at leveraging cloud principles to provide feasible access and programmability to different IoT applications. Second, we worked towards complementing the principles on the cloud by exploiting edge functionalities for a general-purpose. We specifically experimented with an e-Health application and two automotive applications that resemble IoT environments. An additional experiment simulating home automation was performed in-lab. The experiments revolved around a comprehensive (cloud to edge) development as a framework that provides the necessary devices to ease the deployment of IoT cloud and edge APIs. The cloud API makes stored data more accessible to other systems in a secure fashion. A wide range of audiences and platforms can use the data. The edge IoT solution provides feasible and seamless connectivity for collections of IoT edge devices to the cloud as well as local computations. The scope of this framework is the design and implementation of a generic web service that accommodates numeric-type data in compliance with REST guidelines in a reusable fashion. Furthermore, we set on adopting a document-based DBMS such as MongoDB to complement the RESTful nature of the desired web services.

Towards this objective, we proposed Atmosphere, a generic measurement-oriented framework for speeding up the exposure of APIs for accessing and managing smart things in the IoT ecosystem. The contribution of this work is threefold: 1) the proposal of a cloud storage model using abstracted IoT web resources and relations to reflect the structure and functionality of the IoT edge applications. 2) The proposal of an edge engine using abstracted methods and routines to act as a hub for IoT edge devices and 3) the provision of methods and tools to add reusable components to the existing framework. In the context of this development, the target users are mainly IoT developers, IoT service providers, and users with fundamental-level IoT support.

Atmosphere's high-level architecture, shown in Figure 1, is a collection of generic services on the cloud and the edge targeting IoT. Atmosphere introduces programmable models via the edge engine and abstracted API service representation on the cloud. The communication protocols across the usage of the platform – from edge to user interface – are compatible and secure.

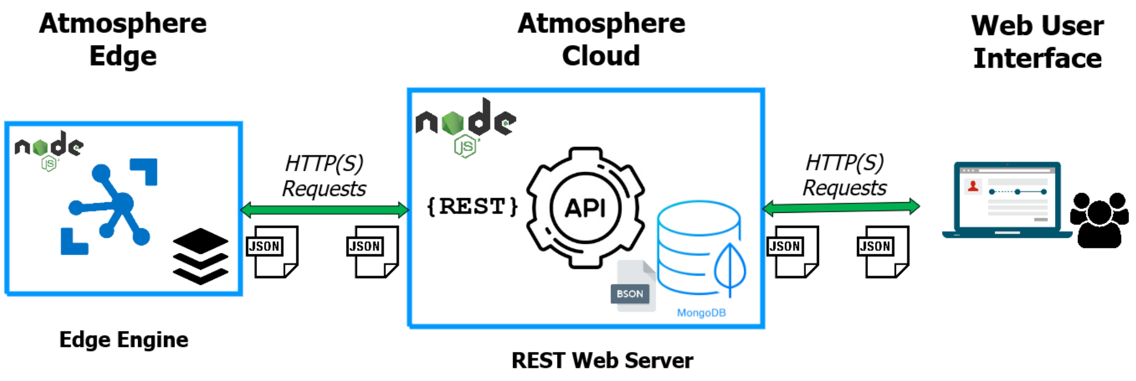


Fig. 1. High-level Block Diagram of Atmosphere cloud and edge

Atmosphere uses JavaScript Object Notation (JSON) as a data-interchange format, internally and externally. JSON is a lightweight text or data format that is easy for humans and computers to read. It is used for exchanging data between applications and web APIs or services. Although it has its origins in the JavaScript programming language, it is often looked at as language-independent.

At the center of our development is the cloud API. The API is responsible for connecting backend data and applications, devices and services, and external party applications together. Atmosphere API is a RESTful API that is based on the architectural style known as REST. A seamless communication concept is realized through the adoption of the REST paradigm for HTTP. The adoption of REST facilitates the use of IoT by both the end-users and service developers across networks containing smart objects [23]. A RESTful API concocts Simplicity, it is easy to grasp since HTTP verbs are based on CRUD (Create, Remove, Update, and Delete) operations. REST design is stateless and separates the concerns of the client and the server. Moreover, REST reads can be cached for better performance and scalability. Even though REST supports many data formats, the predominant use of JSON allows for better support by browser clients. Client requests consist of a URI (Uniform Resource Identifier), an HTTP verb, a request header, and an optional request body. The alternative for RESTful is SOAP [41], where different components are required such as XML format data communication. SOAP-based web services require more work to pack and unpack the data. The complexities with SOAP-based web services lead to REST-based APIs. RESTful APIs are easier to create and easier for clients to consume.

Atmosphere relies on HTTP as a client-server communication protocol. HTTP status codes are employed throughout the API, including Informational responses: 1xx, Successful responses: 2xx, Redirection responses: 3xx, Client error responses: 4xx, and Server error responses: 5xx. HTTP is an unencrypted communication protocol. Without HTTPS (HTTP Secure) enabled, the communication channels edge-cloud as well as User Interface (UI)-cloud are vulnerable. An HTTPS extension was added for communication security. We obtained and installed an HTTPS certificate for the deployed instance of Atmosphere's cloud server from a publicly trusted Certificate Authority (CA).

On the edge, we implemented a generic edge engine, as a runtime system for embedded boards. The engine is designed to provide feasible edge computing capabilities on low-end IoT edge devices and can be adapted for different use case implementations. The engine runs stream-managing scripts and commands for the sensory devices. Scripts in the edge constitute a set of JSON objects with information about operations performed to process each stream of measurements. These operations perform local computations on the measurements like filter, map, combine, and send as well as support for custom functions. Working towards having Atmosphere as full-stack Node.js development, we used Node.js to code the edge engine. For web developments on the edge, Node.js is advantageous in real-time performance over alternatives (e.g. Python). It is very good at handling projects with many simultaneous connections or applications with high-speed input/output (I/O).

Both the cloud API and the edge engine in Atmosphere rely on asynchronous processing. A usual server-side approach (e.g., in PHP, ASP.net, Ruby and Java) involves multi-threading. Node.js avoids the multi-threading burden by employing a non-blocking single-thread pattern and is able to efficiently serve multiple concurrent clients by operating asynchronously, employing the event-loop mechanism. The asynchronous model allows releasing the thread that is handling the request to become available for other processes. Nevertheless, we invoked some synchronous functions in cases where we needed the thread to stay engaged until the request is processed, with a defined timeout to avoid an endless wait.

Using Atmosphere, developers and service providers will find a deployment-ready cloud platform containing the common IoT entities as well as an edge engine containing computational options for their own choice of sensor objects. Enabling the edge to perform computation on data before sending it to the cloud supports the concept of Edge Computing, which implies reducing traffic on communication channels and facilitating maintenance across the IoT network. On the other hand, developers using Atmosphere are required to do the mapping of the application-specific attributes to the abstracted attributes offered by Atmosphere. Atmosphere’s components; cloud API and edge engine; are highly suitable and compatible with each other, but they can be independently integrated with third-party components as well.

Section II – Cloud API

A. Requirements

Based on the literature analysis and our insight in industrial research projects (e.g., [42]) elicited the need for an IoT domain-independent data framework to support the efficient development of IoT applications dealing with measurements. The solution would support easy problem and context modeling and facilitate collaboration between developers, customers, and stakeholders. When the data model is defined, developers should be able to quickly configure the framework, so as to allow edge devices to upload data and (third-party) applications to seamlessly access them, either directly or after processing, for instance extracting statistical values, such as means or histograms.

From an architectural point of view, requirements concern scalability, ease of deployment, preservation of data integrity, large reusability, standardized IoT terminology in a domain-independent generalizability and applicability, user management. In detail, scalability is the ability of cloud development to be used or produced in a range of capabilities. Ease of deployment is the fast and feasible setup of the cloud server. In such a cloud setup, the preservation of data integrity is achieved in maintaining the reliability and trustworthiness of the data. The way the system can be deployed in different settings represents reusability. Standardized IoT terminology means that the employed terminology addresses standard IoT components. Finally, user management is required in order to ensure different levels of user access through the cloud.

B. System architecture

Atmosphere’s cloud framework represents IoT resources as objects, with connections between these objects as relations in the IoT application. This representation allows direct and fast mapping of IoT entities into a series of related objects to build a comprehensive solution. The framework underwent multiple stages of development, starting with core models and functionalities, and eventually expanding to support an expansive set of beneficial resources and services. The latest deployed solution offers processing, querying, and scalable storage of addressable information on the cloud. The following sections (1, 2, 3, 4, and 5) details the design of the model and, based on the proposed design choice, the practical implementation of Atmosphere.

1. Platform choices

Based on the above requirements, we opted for designing a framework leveraging cloud principles to support scalability and ease of deployment. The framework provides a generic web service that accommodates numeric-type data in compliance with REST guidelines in a reusable fashion. Figure 2 shows a block diagram of Atmosphere cloud components and platforms. A major set of design choices for a data-oriented framework is related to the definition of the resources it will expose. A RESTful API separates the user interface from the server and data storage, which improves portability, scalability, and independent development.

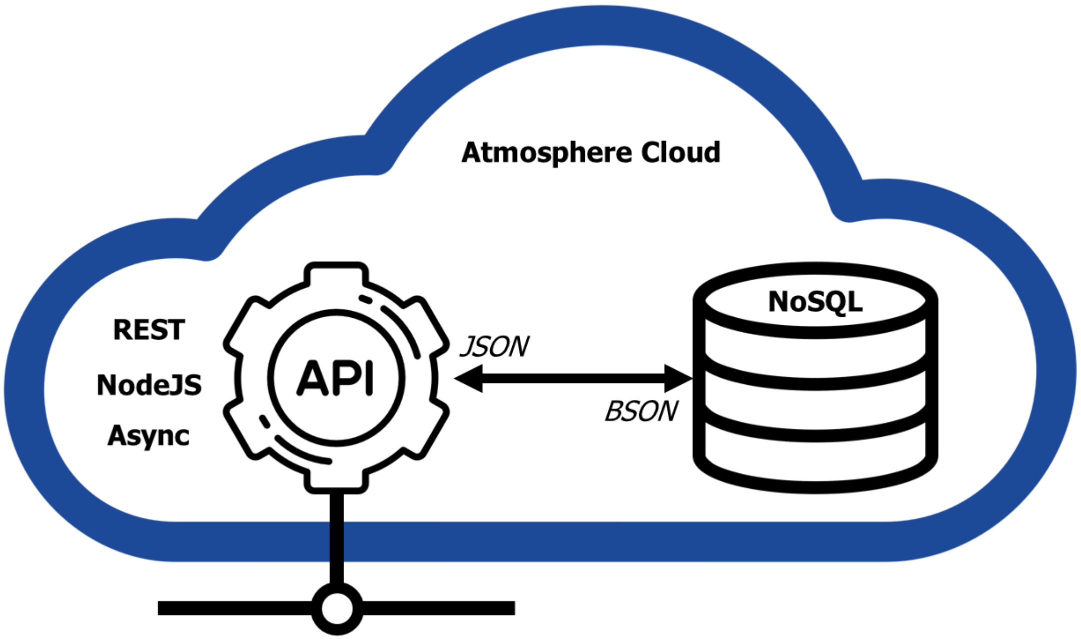


Fig. 2. Atmosphere Cloud Block Diagram.

API services in Atmosphere are implemented in Node.js – a JavaScript-based open-source server environment – within the Express.js framework. Express.js is a web application framework for Node.js. Node.js shines in real-time applications employing push technology over WebSockets, which makes it very suitable to Atmosphere's concept. This open-source framework offers easy integration of third-party services and middleware. The single-threaded, event-driven architecture of Node.js allows it to handle multiple simultaneous connections efficiently. Most of the popular web platforms create an additional thread for each new request, using up random access memory RAM for the whole time it takes to process it. Node, on the other hand, operates on a single thread, making use of the event loop and callbacks for I/O operations, delegating tasks such as database operations as soon as possible. This allows it to handle hundreds of thousands or even a million concurrent connections. What's more, Node.js embraces scalability from the get-go, with powerful features such as the Cluster module enabling load balancing over multiple central processing unit (CPU) cores. The Node Package Manager, known as an "NPM", allows developers to install, update, and use smaller open-source software packages (modules), which means they do not have to write common features from scratch and can avoid new layers of complexity that often come with that particular territory. In our modern world, things are constantly shifting and new technologies rise and fall, sometimes without even entering long-term support (LTS). Moreover, it is difficult to develop and maintain an app written in an outdated language. According to the 2018 Node.js User Survey Report [43], 61% of programmers consider LTS for Node.js an important feature. That knowledge allows developers to assess what the future holds for their application and to plan further development according to the timeline. Programmers can easily plan to adopt new versions based on their regular development cycles. Every major release of Node.js will be actively maintained for 18 months from the date it enters LTS, after which it will transition to maintenance mode lasting another 12 months.

For cloud storage, we chose the NoSQL document-oriented database platform MongoDB [44]. NoSQL databases provide a series of features that relational databases cannot provide, such as horizontal scalability, memory, distributed index, and dynamically modifying data schema [45]. Compared to their traditional alternative; the Relational Data-Base Management System (RDBMS); MongoDB is easier to scale and tune due to its schema-less nature, where the number of fields, content, and size of the collection documents can differ from one document to another [46]. The database schema is thus defined by Atmosphere's code schema. In addition, conversion and mapping of application objects to database objects are not needed. Since it is open-source, MongoDB has a wide software development community and professionals that contribute to enhancements, unified modifications, and bug fixing of the code. There is the cost of a certain difficulty in coding complex queries. However, this is hidden to the user of the framework, who will access data through predefined routes to the modeled resources, according to the REST API design principles [23]. REST supports the mapping of HTTP verbs (Get, Post, Put, Delete) and the classical CRUD database actions (Create, Read, Update, Delete). Behind the scenes, MongoDB saves documents in a Binary JSON format (BSON). BSON extends the JSON model to provide additional data types, ordered fields, and to be efficient for encoding and decoding within different languages. Like JSON, MongoDB's BSON implementation supports embedding objects and arrays within other objects and arrays – MongoDB can even 'reach inside' BSON objects to build indexes and match objects against query expressions on both top-level and nested BSON keys. This means that MongoDB gives users the ease of use and flexibility of JSON documents together with the speed and richness of a lightweight binary format. Unlike relational DBMSs, MongoDB does not impose the prerequisite of defining a fixed structure. Models in MongoDB allow hierarchical relationships representation, with the ability to modify the structure of the record. Furthermore, MongoDB recognizes data in JavaScript Object Notation (JSON) [47], a natural JavaScript format, which means that no conversion is required on a Node.js server. JSON facilitates the exchange of data between web apps and servers in a compact and human-readable format, preventing the need for a persistence layer.

For securing the API, JSON Web Tokens (JWTs) were employed. JWT is based on the RFC 7519 standard [48] that defines how access tokens can be generated and encoded as JSON objects, to enable the secure transmission of data. A JWT consists of a set of claims, which refers to information in the form of key/ value pairs (Claim Name/Value) that are used for authentication, authorization and for exchanging sensitive information. A JWT is trusted since it is digitally signed using an HMAC algorithm or an RSA signature with SHA-256 (RS256).

2. Modeling

Atmosphere cloud was designed to represent the application context and its elements as interrelated software objects, onto which to build applications. These objects are modeled as resources, with their own schemas and functionalities, accessible through the API routes. Defining resources to expose the interface is thus a key design step and requires abstraction in order to support flexibility, extendibility, and scalability. Not only do the choices concern the terminology, but also the semantic of each resource. These resources correspond to documents (tables) in the storage database. Each resource constitutes of a number of attributes (fields), some are mandatory while others are optional.

Resources are the collection of elements that (collectively) define the IoT application. At the core of these resources are the essential elements that are common in the IoT environment: Thing, Feature, Service, Device, and Measurement. In Atmosphere's API,

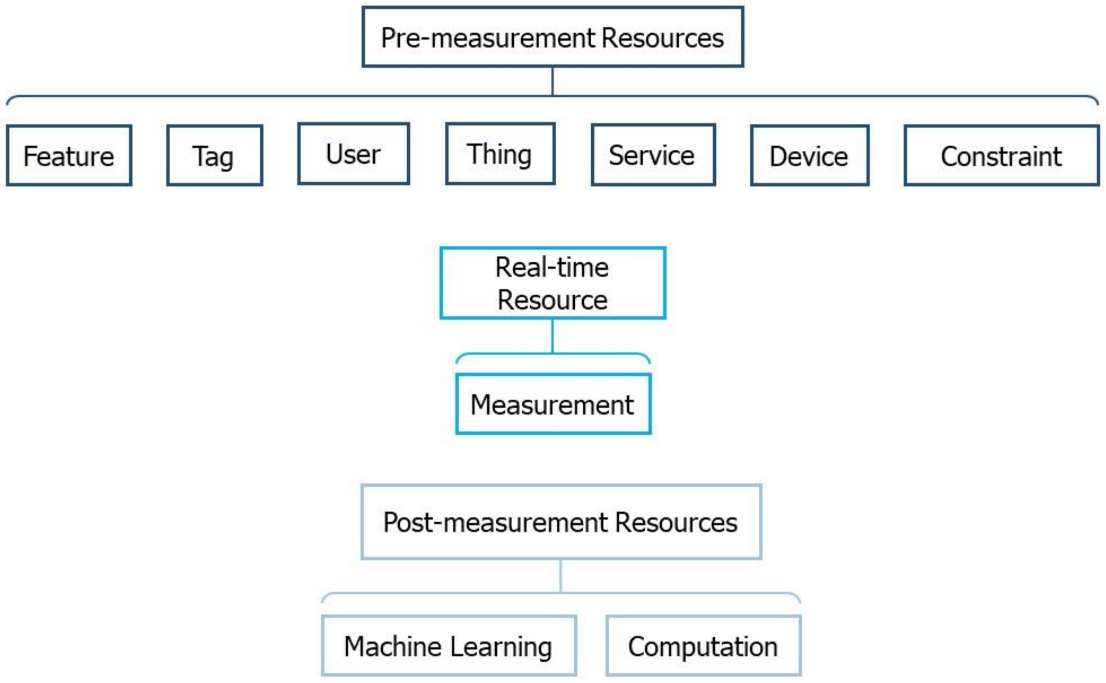


Fig. 3. Categorized Resources of Atmosphere's API

we grouped the available resources into three categories: pre-measurement (mandatory and optional), the measurement resource, and post-measurement resources. Figure 3 shows the different levels of supported resources in Atmosphere's API model.

Pre-measurement resources are those that get populated prior to uploading a measurement to which they are subjects. Such resources could be mandatory, such as thing, feature, and user, or optional such as service, device, and tag.

A Thing represents the subject of a Measurement. A Feature represents a (typically physical) dimension measured by a Device. Each dimension has a name and a unit. A Service denotes the type of the target IoT deployment. A Device is a tool providing measurements regarding a thing (or an actuator that acts within a thing to modify its status). A Measurement represents a value of a Feature measured by a Device for a specific Thing in the context of a certain Service. Other supplementary resources are Alert, User, Provider, Subscription, Log, Login, Script, Tag, and Constraint resources.

As post-measurement resources, machine Learning and Computation resources supply analytical abilities to the API. In the machine learning resource, traditional machine learning algorithms are available with a wide variation for options. For regression, the resource can perform linear and polynomial regression of univariate or multivariate data sets. For classification, the resource implements the logistic regression

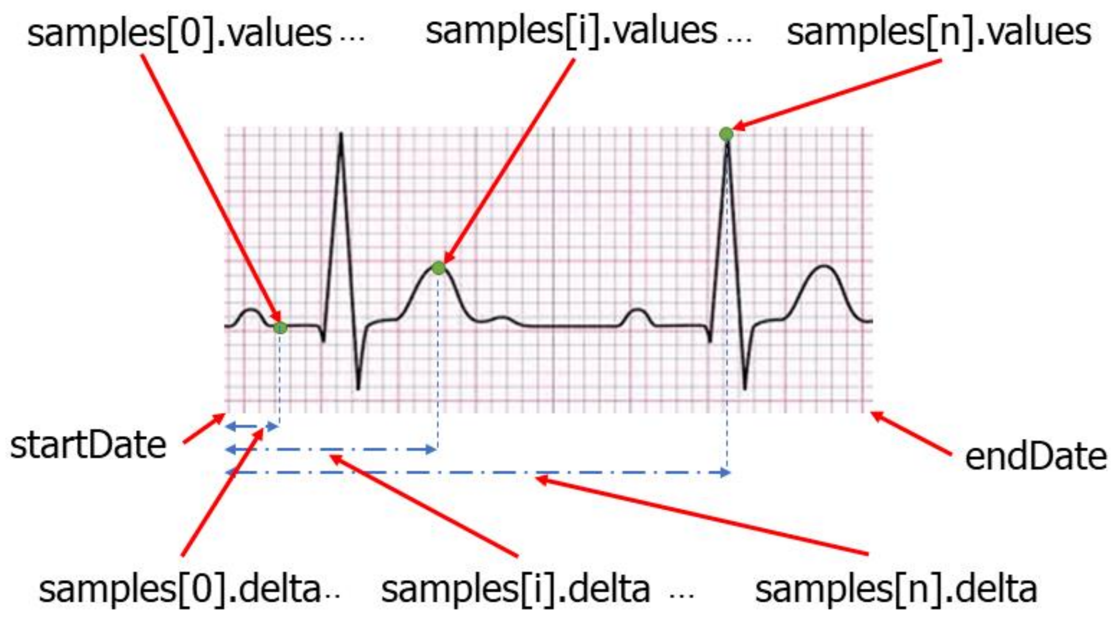


Fig. 4. Example of a Measurement Sample.

algorithm with k-NN (k-Nearest Neighbor) algorithm. For clustering, I implemented a fast variation of the k-means algorithm. I took advantage of TensorFlow for JavaScript [49] in implementing the machine learning resource.

The concept of measurement abstracts the samples posted to and retrieved from the database. Its structure must match its Feature. A measurement can contain one or more homogeneous samples, the latter being the typical case when sampling signals (Figure 4). Each sample contains a vector of values. Values are not necessarily homogeneous. For instance, a sample could represent a set of statistical information on a quantity (e.g., average, stdev, etc.). Each value can be a scalar (e.g. a temperature), a vector (e.g. the orientation in space) or a tensor of numbers (e.g., general multidimensional data points). The Feature resource is used to check the integrity of each received measurement. The size of each measurement sample value must match the corresponding dimension attribute stored in the corresponding feature item.

Figures 5, 6, and 7 preview with examples the defining relationship between the measurement and features resources. For example, a measurement whose feature is weather-conditions must have in each of its Sample elements of the samples array exactly two values as the number of items defined within the weather-conditions feature. Furthermore, the dimension of each value in the sample must also correspond to the

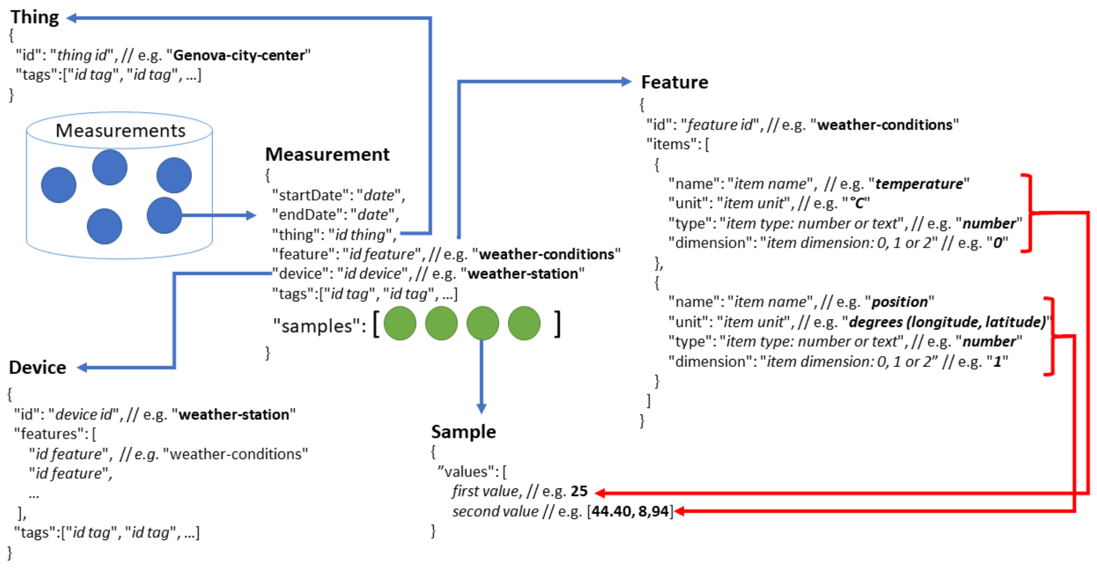

Thing
{
"id": "thing id", // e.g. "Genova-city-center"
"tags":["id tag", "id tag", ...]
}
Measurements
Measurement
{
"startDate": "date",
"endDate": "date",
"thing": "id thing",
"feature": "id feature", // e.g. "weather-conditions"
"device": "id device", // e.g. "weather-station"
"tags":["id tag", "id tag", ...]
"samples": [ ]
}
Feature
{
"id": "feature id", // e.g. "weather-conditions"
"items": [
{
"name": "item name", // e.g. "temperature"
"unit": "item unit", // e.g. "°C"
"type": "item type: number or text", // e.g. "number"
"dimension": "item dimension: 0, 1 or 2" // e.g. "0"
},
{
"name": "item name", // e.g. "position"
"unit": "item unit", // e.g. "degrees (longitude, latitude)"
"type": "item type: number or text", // e.g. "number"
"dimension": "item dimension: 0, 1 or 2" // e.g. "1"
}
]
}
Device
{
"id": "device id", // e.g. "weather-station"
"features": [
"id feature", // e.g. "weather-conditions"
"id feature",
...
],
"tags":["id tag", "id tag", ...]
}
Sample
{
"values": [
first value, // e.g. 25
second value // e.g. [44.40, 8,94]
}

Fig. 5. Atmosphere resource model relations around the Measurement resource.

specified item dimension in the feature resource, which is 0 and 1 for weather-conditions feature items respectively.

We also defined the Computation resource, which makes complex queries to perform post-processing calculation on raw data (typically measurements) exploiting the cloud server capabilities. A set of (typically statistical) computation types are executable, identified by the 'Code' attribute. Currently, the following codes are supported: maximum, minimum, average, median, standard deviation, variance, first quartile, third quartile, histograms. A custom computation is also available, which executes a custom script uploaded by the user. In the case of stdev() and var(), the engine offers two variations: population-based and sample-based [50]. Codes are parametrized, to allow the user to specify the dimensionality of the measurements' values to be aggregated.

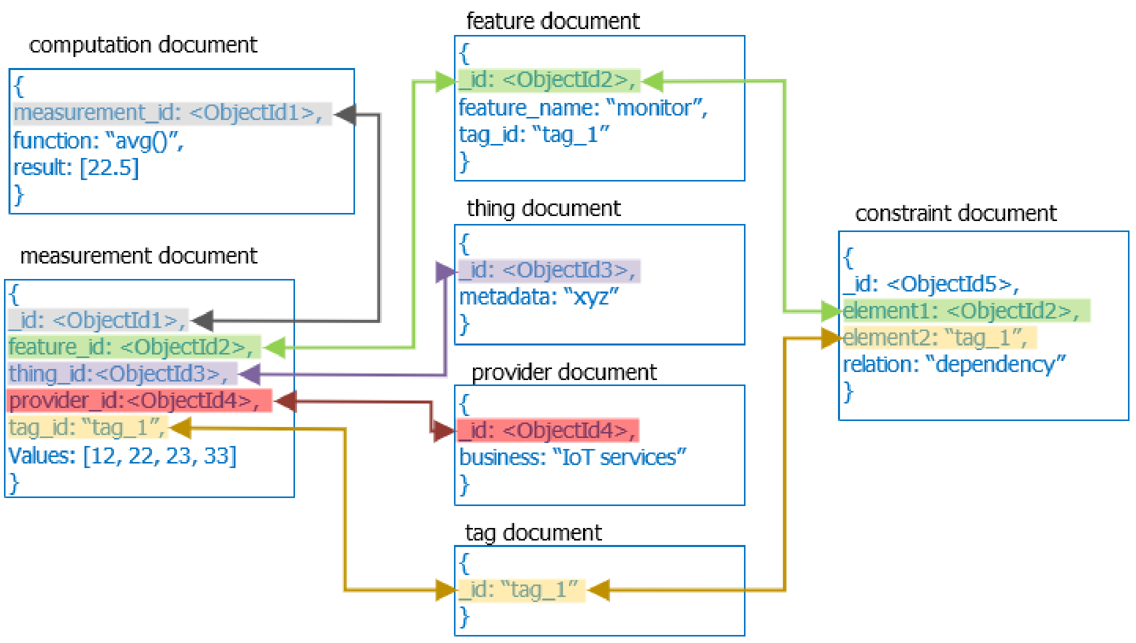


Fig. 6. Block diagram showing some resources relations in Atmosphere.

From the filtered measurements, the value vector can be chosen as a whole, include only a number of value arrays, or include only a number of elements in each value array. Another type of computation concerns outlier detection, which is key to guarantee the quality of data series. The result of a Computation is structured and stored as a Measurement, thus allowing further processing.

Another abstraction that we defined to support the possible needs of a data consumer service, is the Constraint, which allows defining relationships between different resources. We modeled this as a resource in order to the maximum flexibility in adding associations/dependencies between resources while avoiding hard-coding them inside the resources themselves. As a use case, a user interface that is dynamically built by

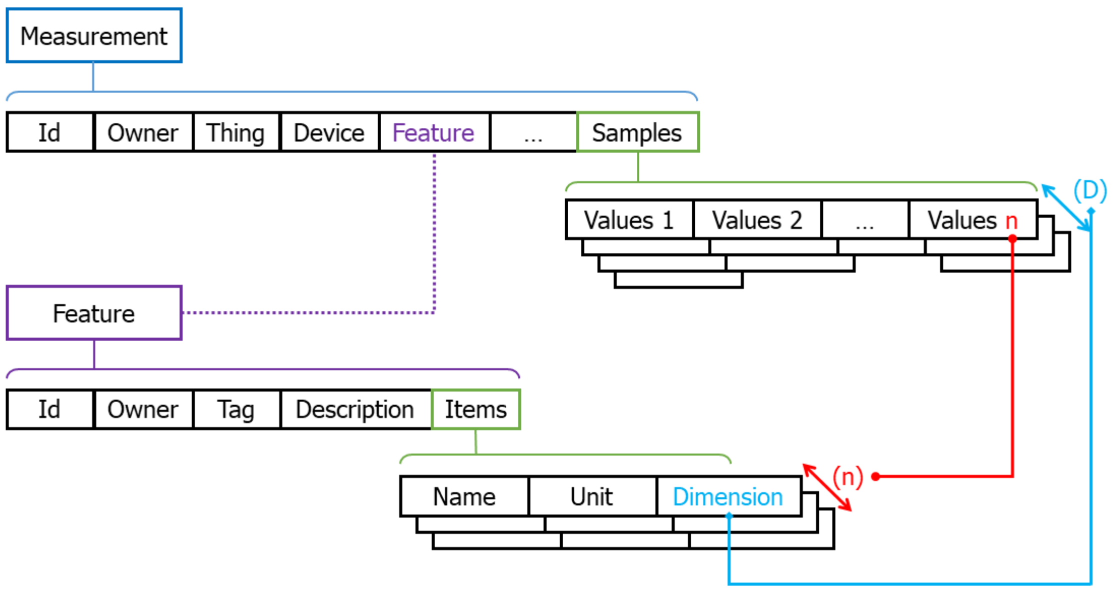


Fig. 7. Depiction of the cross-validation between ‘Feature’ and ‘Measurement’ resources.

querying the DB can show in a drop-down menu the proper options only. Figure 5 gives an overview example of the Atmosphere resource modeling.

3. *Implementation*

Within the API, resources are modeled through two stages: schema and controller. The schema defines the resource structure while the controller defines its functionality.

A resource schema describes the fields, including their types, default values, and references for other resource fields. Fields that refer to other resources have cross-validation functions implemented within the resource schema. The schema also includes plugin definitions as well as indexing options. The controller defines the HTTP methods that are supported by the resource. The main methods are GET, POST, PUT, and DELETE. Practically, the GET method is represented in the controller by two asynchronous functions: ‘get’ (many), which supports filter and aggregate queries to retrieve multiple records and ‘getone’ which fetches one record by its ID. A POST method is responsible for the insertion of new records, while the PUT method updated existing records. The DELETE method removes (permanently or soft deletes) records from the database. Each of those asynchronous functions takes an HTTP request as an input, performs a specific function, and then issues a response to the requesting (source) Internet Protocol (IP) address.

The framework guarantees that all its exposed resources can be manipulated through the previously mentioned HTTP methods. Standard response codes were defined for each method like success codes (2xx) and client error codes (4xx) with suited information in the response message.

We have implemented the RESTful API services in Node.js (JavaScript-based) within the Express.js framework. This open-source framework offers easy integration of third-party services and middleware, particularly the MongoDB database and its seamlessly utilizable Mongoose persistence layer. A usual server-side approach (e.g., in PHP, ASP.net, Ruby and Java) involves multi-threading. Node.js avoids the multi-threading burden by employing a non-blocking single-thread pattern and is able to efficiently serve multiple concurrent clients by operating asynchronously, employing the event-loop mechanism.

In storage, resources correspond to collections in the database. Each resource involves a number of fields, some of which are mandatory, while others are optional. Within the API, resources are implemented through two stages: schema and controller. The schema – which is needed for data-checking, given the schema-less nature of MongoDB – defines the resource structure, while the controller implements the resource functionality. A resource schema prescribes the fields, including their types, default values, and references to other resource fields. Fields that refer to other resources have cross-validation functions implemented within the resource schema. The schema also includes plugin definitions as well as indexing options. The controller defines the HTTP methods that are supported by the resource (typically: GET for fetching resources, POST for inserting, PUT for updating, DELETE for removing, either permanently or softly).

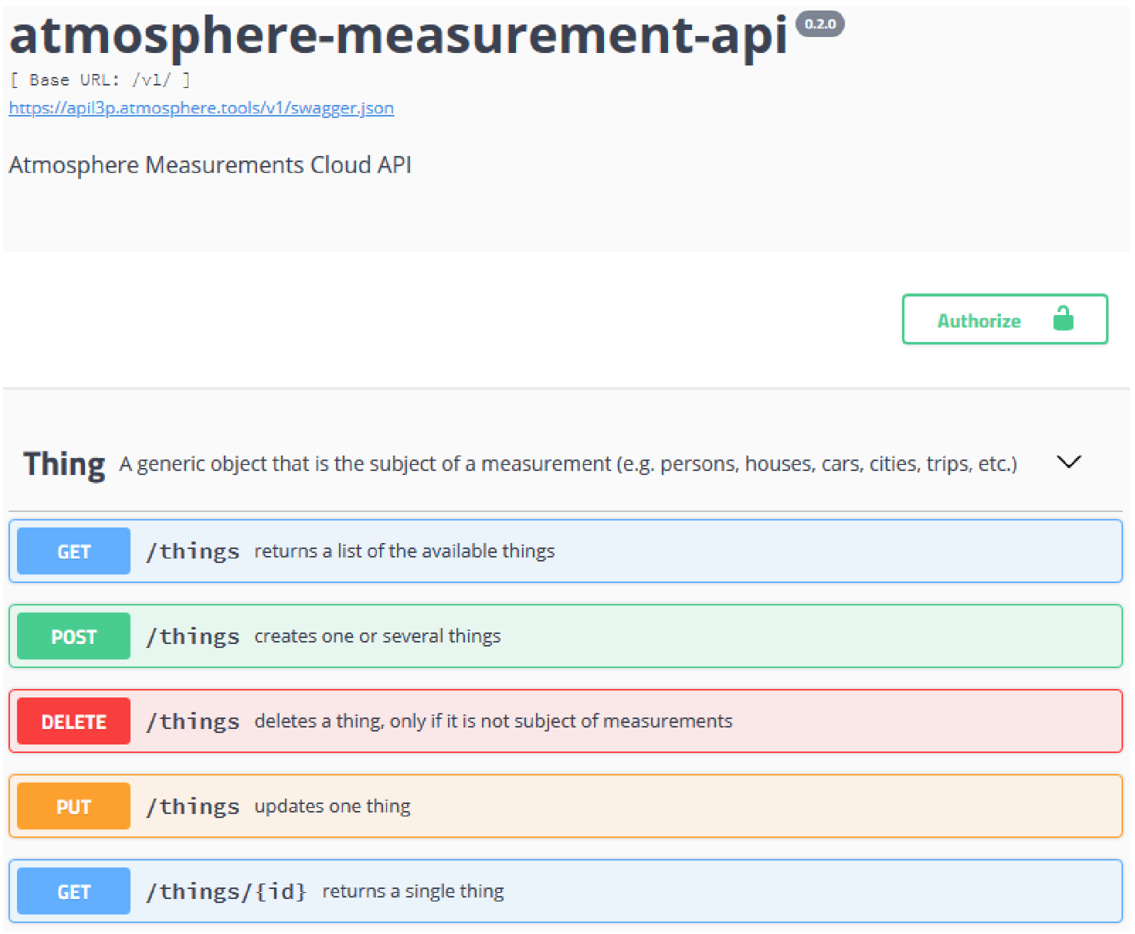
The Computation controller performs incremental calculations in order to avoid exhausting the system’s resources (e.g., memory), as computations are typically performed on huge quantities of data. Computations are obtained in a two-step process, where the client first issues a computation request, which opens a WebSocket through which the client can get information from the system about the progress of the execution.

According to the best practice in software engineering, we included in the framework an automatic test suite for all routes and methods, supported by the NPM [51, 52]. This is key to ensure that all the anticipated API operations perform as intended. Given the custom nature of the API, we developed tailored tests for each method within the resources. I employed ‘chai’ [53], a Test-Driven Development (TDD)/ Behavior-Driven Development (BDD) assertion library for Node.js, paring it to my custom-made testing mechanism through NPM.

Script 1. A portion of the test code from the GET method in the User resource

```
…
chai.use(chaiHttp);
// Test the /GET route
describe('/GET users', () => {
 it('it should GET all the users', async () => {
     await factory.dropContents();
     await factory.createUser("test-username-1", "test-password-1");
     await factory.createUser("test-username-2", "test-password-2");
     const res = await
chai.request(server).get('/v1/users').set('Authorization', await
factory.getAdminToken());
     res.should.have.status(200);
     res.body.docs.should.be.a('array');
     res.body.docs.length.should.be.eql(3);
   });
…
```

The code example in Script 1 shows one test scenario of the GET route in the user resource. In this test, the code creates two new users, then calls the get method for all users. For the test to be successful, each of the last three check must be true. The returned status must be 200 (success), the content of the response body is expected to be an array, and the number of returned users must be three (the two created earlier plus the admin user that is pre-loaded).


atmosphere-measurement-api 0.2.0

[Base URL: /v1/]

https://apil3p.atmosphere.tools/v1/swagger.json

Atmosphere Measurements Cloud API

Thing A generic object that is the subject of a measurement (e.g. persons, houses, cars, cities, trips, etc.)

GET /things returns a list of the available things

POST /things creates one or several things

DELETE /things deletes a thing, only if it is not subject of measurements

PUT /things updates one thing

GET /things/{id} returns a single thing

Fig. 8. Thing resource described in Swagger graphical user interface documentation of Atmosphere API.

As a manual for new consumers of the API, we developed comprehensive online documentation using Swagger UI [54]. The documentation, available via https://apil3p.atmosphere.tools/, is a collection of HTML, JavaScript, and CSS that dynamically generate documentation from Atmosphere's API (a Swagger compliant API). The API documentation provides a description of every web resource and clearly indicate the purpose of each. It also provides the expected JSON request and response schemas to simplify client integration. The deployed API documentation, shown partially in Figure 8, acts as a catalog where developers can easily see what resources are available with detail on how to interact with each. Furthermore, dependencies are pointed out to ensure the awareness of resource relationships. Finally, I added the design documentation, which takes two forms: one for documenting how each resource fits into the greater architecture of the API, and another that illustrates the logic of each resource.

The information provided in Appendix A is heavily influenced by the online Swagger documentation.

4. *Working System*

When the API is first initialized, it connects to the storage server and creates the database 'Atmosphere-DB' with one collection inside. That collection is the 'users' collection, and it is essential to have one admin user in order to create other users and other collections. This first instance of a user is pre-defined within the source code. The 'admin' user can create two other types of users: 'provider and analyst'. While the 'admin' has full access to the DB (all methods unrestricted), the 'provider' user has restricted access that depends on ownership (allowed access to records that are owned by the user), and the 'analyst' user has limited access, that only includes the GET methods. Beside this Authorization access mechanism, another security measure is Authentication. It is based on a JSON Web Token (JWT) as a security pass that expires every 30 minutes. To get a JWT, users must POST to the 'login' resource with their given credentials. The API will reply with a JWT with the specific authorization level of the requesting user. The JWT must then be used as a header attribute for any further requests to the API.

5. *Supported Workflow*

The above system has been designed in order to support an efficient workflow for preparing different measurement-based data-rich applications. The first step consists of the domain mapping, where the field objects are to be mapped to the Atmosphere API's resources. In this phase, the IoT application designer has to define features (i.e., types of measurements), devices (i.e., measurement instruments), things (i.e. main subjects of the measurement), tags (i.e., labels that can be attached as attributes to other resources – typically measurements, features and things), constraints (i.e., relationships between elements in the DB), and user types (e.g., Provider and Analyst). Once these objects are POSTed to the API, the system becomes operational, allowing insertion/update of users, things, field measurements and computation requests; and retrieval of results in terms of things, measurement and computation outcomes. The structure can be updated during the operation as well, by POSTing/PUTting features, devices, and tags. All these actions happen only through the exposed resource routes, with the well-known advantages of the RESTful approach in terms of scalability, encapsulation, security, portability, platform independence, and clarity of terminology and operations. The latest version of Atmosphere API is hosted online and can be accessed via https://api.atmosphere.tools/v1.

Section III- Edge Engine

The amount of data generated by IoT edge devices is exploding. Storage and processing of all the data in the cloud have become too slow and costly to meet the requirements of the end-user. Edge computing presents a substantial solution through facilitating the processing of device data closer to the source. However, developing and deploying computation engines for heterogeneous edge devices is a formidable challenge for IoT developers and service providers. This section presents a generic edge engine for low-end open-hardware that serves as an edge-to-cloud hub as well as an edge computational node. The aim of this development is to support IoT edge developers by facilitating development and cutting on deployment time. The results, observed in two experiments, show a positive impact on deployment speed with low-end hardware.

A. Requirements

The requirements for a generic edge engine stem from the challenges of edge computing in the literature [55, 56]. In edge computing, as several research projects have demonstrated, programmers must partition the functions of their applications between the edge and the cloud [57]. Most early efforts in this area were done manually and carefully tuned, which is not scalable or extensible. Thus, easy-to-use programming frameworks and tools are required. Such tools must include remote configurability and auto-updates.

The edge engine solution must address include difficulties of high latency, where measurement collection, processing, and routing all form a bottleneck due to network congestion and limited computing resources. Furthermore, the solution needs to tackle the user’s familiarity with edge computing by providing simple functions to deploy with the ability to expand and deploy complex functions.

While focusing on providing a lightweight and easy-to-deploy edge engine, the requirements for are versatility, portability, real-time reactivity, simplicity in function support, and high customizability. Versatility describes a flexible way of adapting to change in function. Portability is manifested in the ease of script transfer across the IoT network (also involving cloud). Real-time reactivity is the main feature of edge setups where responses to triggers as well as processing and uploading of data must be as fast as possible. The simplicity of the supported functions ensures the feasible use of processing functions by the less experienced users. While high customizability offers experienced users, the ability to run functions that are more complex and tailored for their specific IoT service or setup.

These requirements are suitable for limited bandwidth on the communication channels available for IoT node connections (I2C, BLE, Wifi) as well as an internet connection to the cloud. Furthermore, for better data management and structural organization of edge devices, a common edge hub for multiple sensory nodes works better than connecting each node directly to the cloud.

B. System architecture

The concept behind the generic edge engine is based on the assumption that it will be deployed on low-end hardware, that ranges from microcontroller development boards and single-board computers such as Arduino, ST microcontroller, Raspberry Pi, or other systems with a similar level of computational capabilities. For a detailed description of the range of targeted computational capabilities associated with low-end hardware, refer to Appendix B.

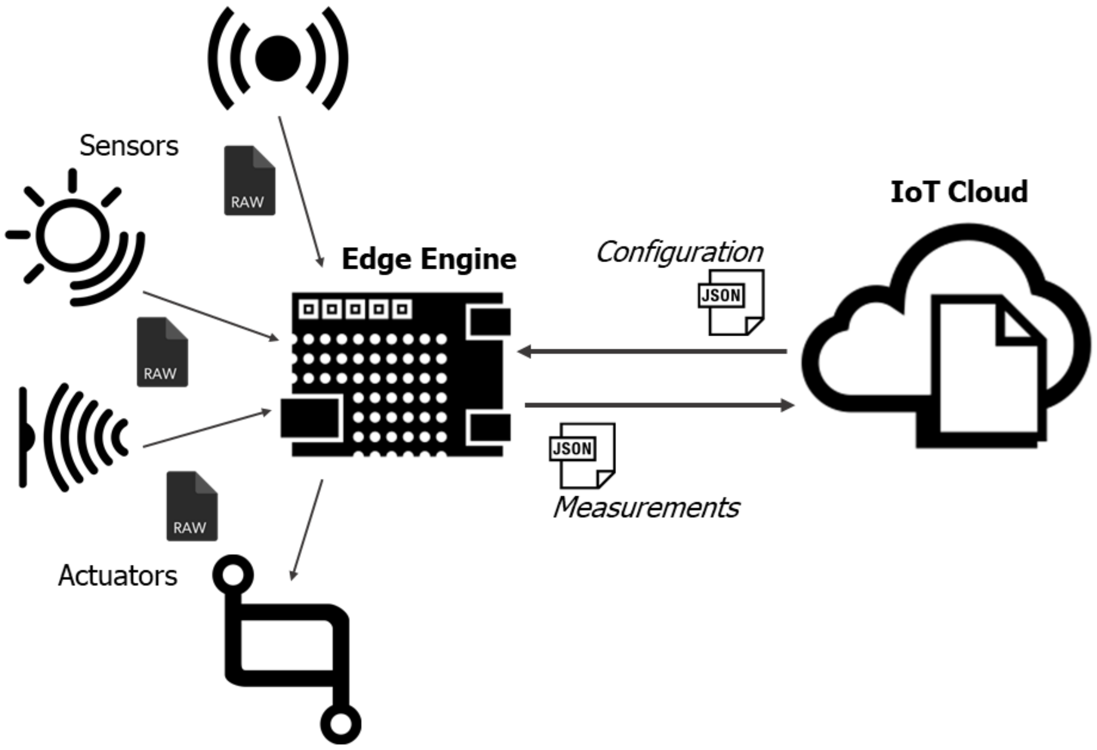


Fig. 9. Edge Engine Ecosystem

1. Design

The nature of the edge engine deployment, shown in Figure 9, is close to the physical IoT sensors on-site. In the edge engine implementation, we made sure of the transparent remote configurability via HTTP.

The edge engine supports local processing with a set of predefined functions by the Array.prototype JavaScript library. Furthermore, the engine supports the execution of custom functions defined by the developers in the script. It is possible to run edge computing with one or more functions serially (on the same data stream) in a chain fashion.

Uploading data streams to the cloud can be done in one of three supported fashions: Continuous (Upload data as soon as received or processed), Batch (upload data buffer at a specified interval), and triggered (upload data upon specific conditions).

2. Implementation

The implementation of the edge engine was manifested by several processes that depend on parameters specified by the user in the edge script or, if missing, initialized from a pre-defined default script within the source code. The processes are login to cloud, get script from the cloud, read from the sensors, process data locally, store locally, and upload to cloud. Three of those processes require a time schedule for the delay in their repetitive execution.

The first is the process ‘login to cloud’ where the interval must be appropriate to the expiration time of the security token, which, in our implementation with Atmosphere, was a JSON Web Token (JWT). The second is the ‘read from sensors’ process where the time interval sets the pulling rate from the edge devices. The third is the ‘upload to cloud’ where the time interval specifies the rate of POSTing the contents of the local buffer – whether it contains raw or processed data – to the cloud. Moreover, this interval decides the nature of the upload – batch or stream – when contrasted with the pulling rate from the sensors.

If the ‘uploadInterval’ equals the ‘readInterval’, the upload is streaming every measurement immediately to the cloud. While if the ‘uploadInterval’ is set bigger than the ‘readInterval’, then the upload will become a batch upload since the uploaded buffer would contain multiple measurements the size of which is determined by the difference between the two intervals: read and upload. The example in Script xx is a JSON script with an upload interval 10 times the readInterval. This makes the measurement upload batches of 10 measurements at a time.

Other met-data parameters such as ‘device’, ‘tags’, ‘thing’, and ‘feature’ are used as identifying attributes for the POSTed measurements. The ‘operation’ field would contain the computational functions that the user intends to apply to raw data in local processing. If this field is empty, the edge engine discards local processing and the raw data are uploaded as-is. The ‘operation’ field can specify pre-defined operations, such as filter, map, window, and others, and custom operations as well that are valid JavaScript functions. In the case that one or more of these parameters (fields) in the script are missing or not specified on purpose, the edge engine will load default parameters that are pre-defined in the source code. Furthermore, the edge engine keeps checking for modifications of the script on the cloud DB to update those parameters immediately.

To deploy the edge engine properly, a number of preparatory steps have to be completed beforehand. As the typical user to deploy the edge engine solution, we will be referring to the consumer – developer or service provider – as the deploying entity of the edge engine.

Script 2. JSON script example with edge engine parameters

```
{
    "_id": "script_Sample_Id",
    "device": "IoT_Sensor",
    "tags": ["Edge_Device"],
    "command": {
      "method": "POST",
      "resource": "measurements",
      "thing": "Smart_Thing",
      "feature": "Sample_Feature",
      "readInterval": "5",
      "uploadInterval": "50",
      "loginInterval": "1798000",
      "operation": [
            {
            "type": "filter",
            "condition": "value > 30"
            }
      ],
      "custom": ""
    }
}
```

At first, the consumer would prepare a JSON script edge descriptor. It is not crucial to have all fields in the script filled, but it is recommended to fill the essential parameters that are best suitable for the IoT deployment rather than leave it up for the default values. The script would look like the sample script in Script 2. The script would then be sent to

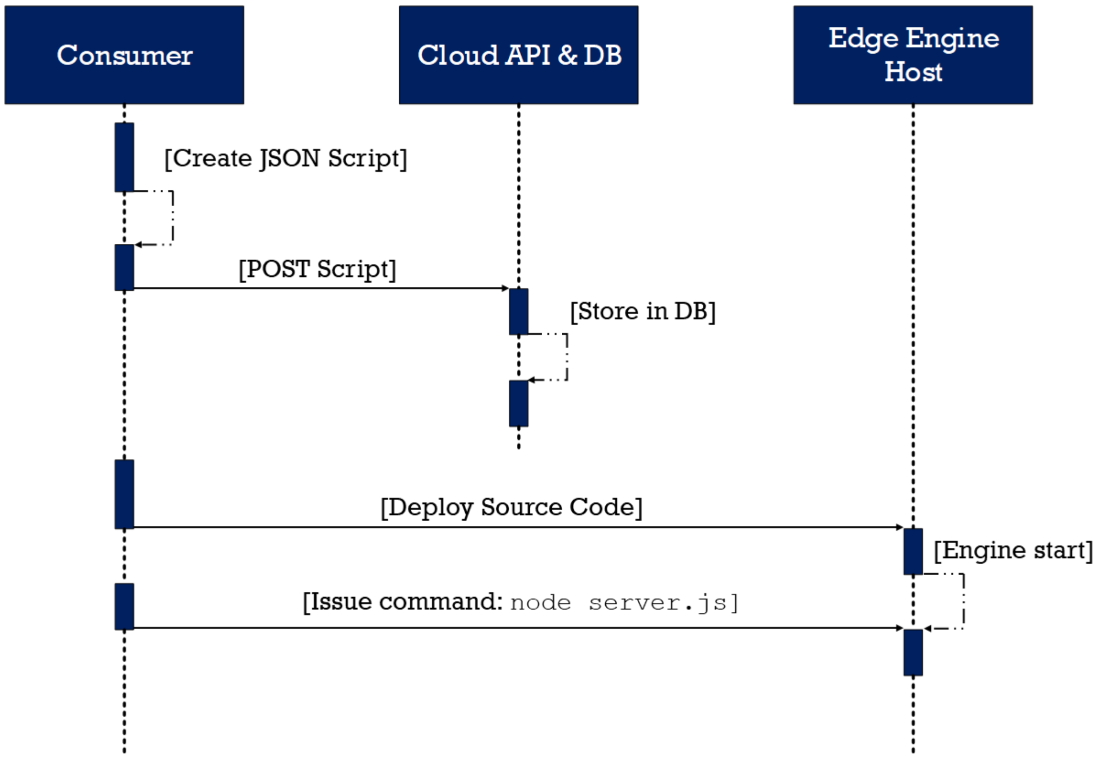


Fig. 10. Preparation procedure for the setup of the edge engine deployment.

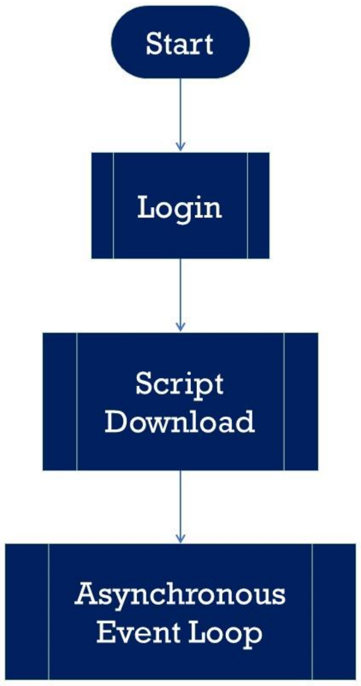


Fig. 11. Flow chart of the entire edge engine source code.

the cloud through the Atmosphere API, or an alternative cloud service, as the body part of a POST request to the 'script' resource: {{url}}/v1/scripts/. The API will store the uploaded script – after it runs the necessary checks – into the DB. Now, the script can is accessible to the edge engine. These preparatory steps are visualized in Figure 10.

In the IoT setting, the host device of the edge engine acts as a hub, thus it must be connected to the IoT sensing and actuating devices on the edge as well as to the cloud via an internet connection. In both cases where we experimented with the deployment of the edge engine, the host hardware was a Raspberry Pi single-board computer. Refer to the Use Cases of Chapter 4, Sections II and IV for details on the edge engine deployment in my IoT experiments. The second step is the transfer of the edge engine source code to the host device. The consumer must make sure to set up the necessary runtime environment support such as NodeJS and NPM libraries and dependencies such as 'express', 'http', 'mathjs', 'prompts', and 'request'. Then, the consumer would start the edge engine by issuing a command in the terminal of the operating system (OS): node source-code-directory/server.js.

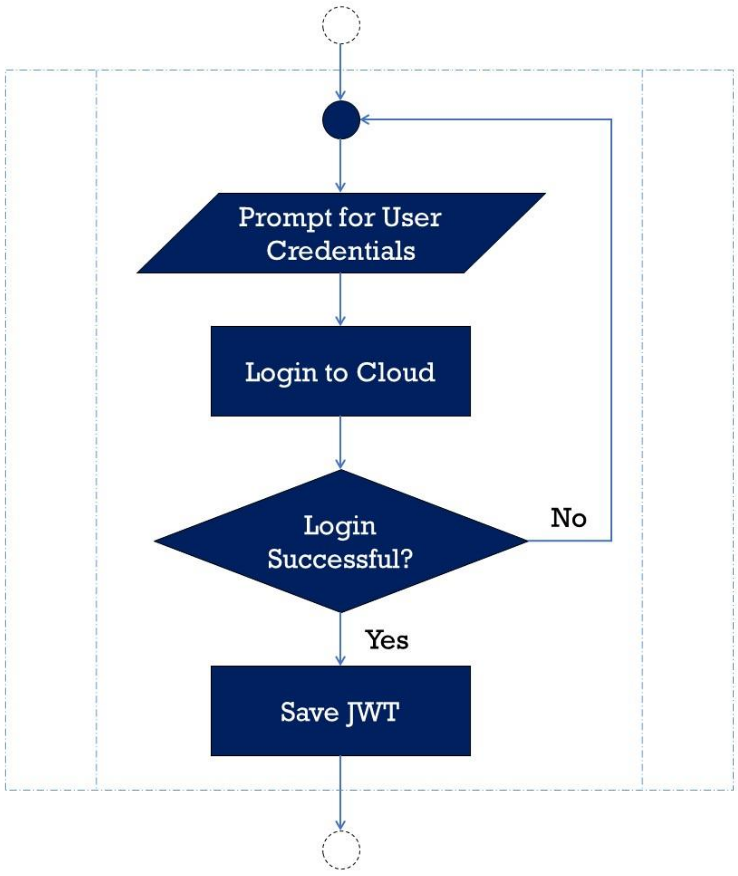


Fig. 12. Flow chart of the Login subroutine.

The edge engine is configured to connect to a cloud (e.g. Atmosphere cloud) by specifying DNS/IP address as an endpoint such as, in my case, https://api.atmosphere.tools/v1. If the cloud is secured, as with Atmosphere cloud, then the consumer must follow the authentication procedure enforced by the API. Otherwise, it depends on the security measures employed by the endpoint cloud service chosen by the consumer. In my particular implementation, the edge engine prompts the client for authentication credentials (username and password) that are predefined for the consumer by the cloud administration. This process is presented in Figure 11 – a high-level flow chart representation of the edge engine – as the 'Login' subroutine. A more detailed representation is provided in Figure 12.

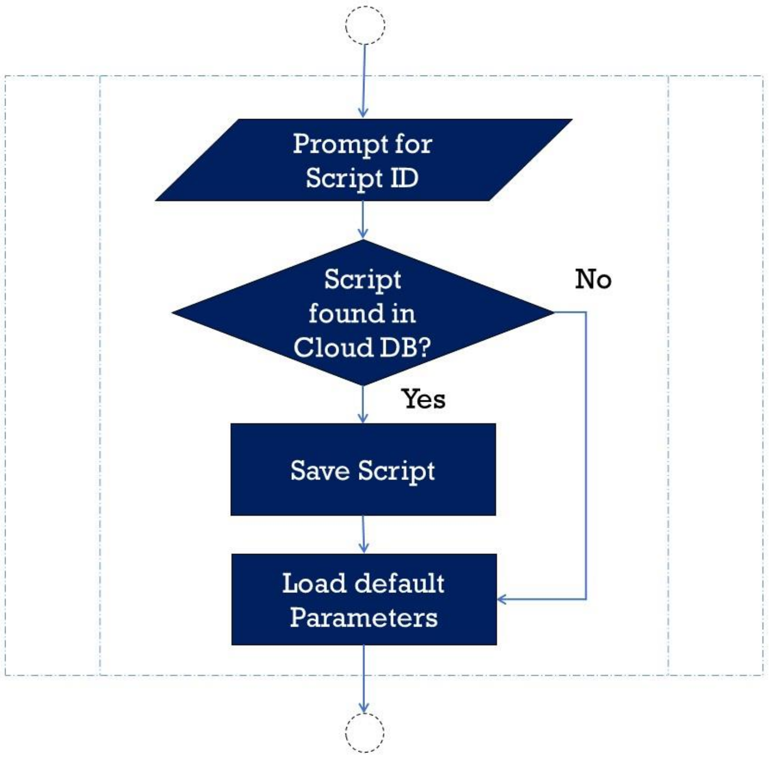


Fig. 13. Flow chart of the Script Download subroutine.

The edge engine sends these credentials through an HTTP request to the specified end-point. The cloud server responds with a JWT that validates the identity of the client. The edge engine awaits this response and stores the JWT token to be used for authorization of further HTTP communication (e.g. uploading measurements) with the cloud server. The edge then prompts the consumer for the script Id, a reference to the script which has been stored earlier by the customer on the cloud database. This step is denoted as the ‘Script Download’ subroutine as shown in Figure 11. A more detailed demonstration of the inner processes of the script download is presented in Figure 13. The ‘Script Download’ flow chart also shows that when the script is not found on the cloud database, the edge engine would load parameters of a default script saved locally within the source code.

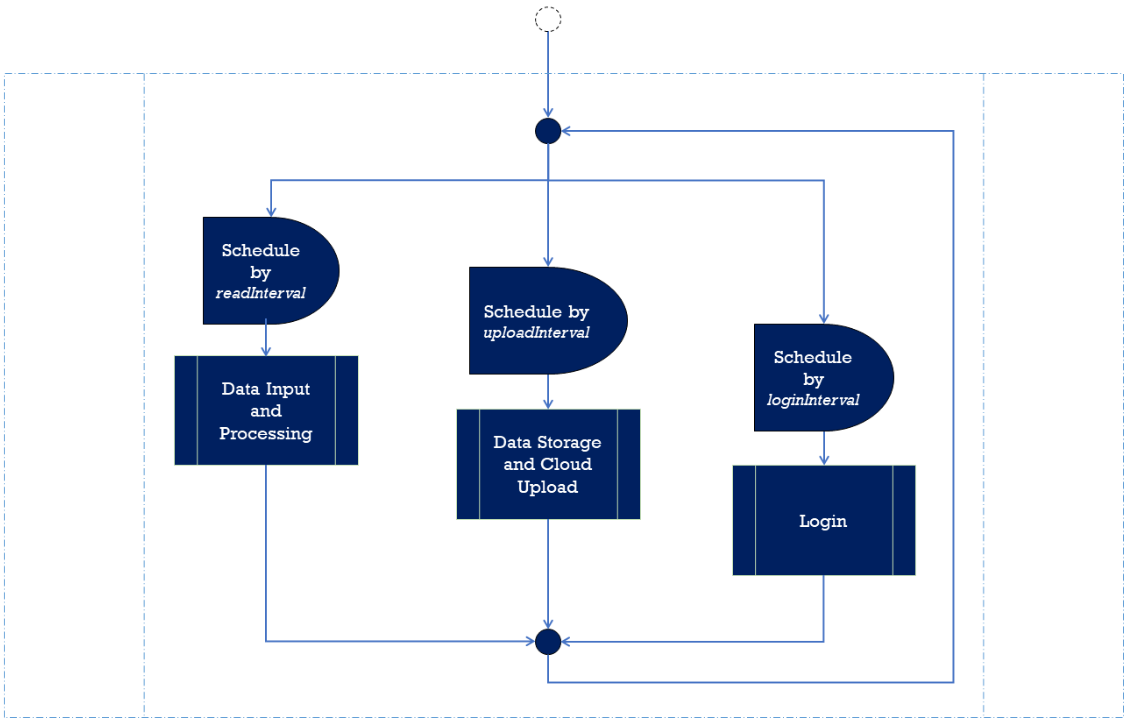

Schedule by *readInterval*
Data Input and Processing
Schedule by *uploadInterval*
Data Storage and Cloud Upload
Schedule by *loginInterval*
Login

Fig. 14. Flow chart of the Asynchronous Event Loop subroutine.

After the ‘Login’ and ‘Script Download’ subroutines are executed, the edge engine enters a state of an infinite event loop where asynchronous functions run in parallel. These functions are significant to the main usage of the edge engine as an IoT edge hub. Since Node.js is single-threaded one functions must precede the other for smooth execution. Precedencies are derived from the logical flow of the edge engine functionalities. Since the ‘Login’ function was executed in the preparatory (initialization) phase, and thus the security token must still be valid (my usual preset is 30 minutes), then the ‘Login’ function is scheduled last in precedence.

Figuring the order of execution of the other two asynchronous functions comes naturally since ‘Data Input and Processing’ is responsible for acquiring and manipulating the resources that the ‘Data Storage and Cloud Upload’ function manages. the order of precedence is key upon the first cycle of execution of the ‘Asynchronous Event Loop’ as well as cycles where two or all of the embedded functions coincidentally schedule the same execution time. Another cause could be implicitly specifying the scheduling intervals by the script cases to coincide such as equal intervals or intervals that are factors of other intervals. As Figure 14 shows, the three functions: ‘Data Input and Processing’, ‘Data Storage and Cloud upload’, and ‘Login’ are scheduled each by its own time interval with the priority to the function in precedence from the left to the right.

The subroutine ‘Data Input and Processing’, also shown in a flowchart form in Figure 15, collects data from connected IoT devices in the form of an ordered stack of arrays, where each array is reserved for one device input stream. Then, this subroutine checks the script for existing ‘operations’ – pre-defined and/or custom operations – to enable local processing then executes those operations according to their attached parameters or conditions if any.

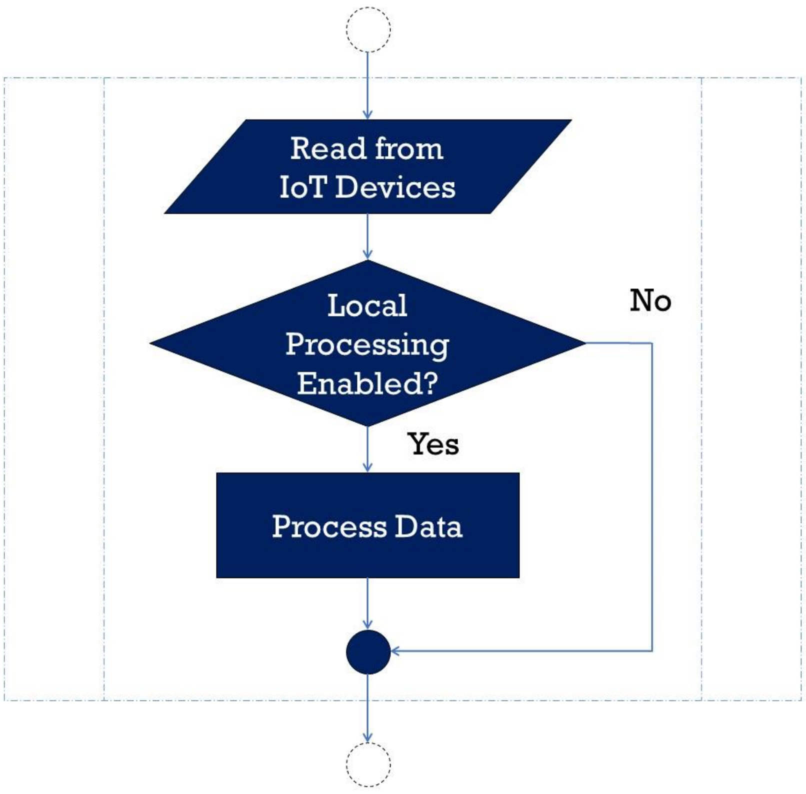


Fig. 15. Flow chart of the Data Input and Processing subroutine.

For example, Script 2 shows a single operation ‘filter’ with condition ‘value > 30’. This is straight forward; the engine would filter the stream allowing only numeric values that are greater than thirty. In another example, Script 3 shows two operations ‘window’ and ‘absolute’ respectively. Operation ‘window’ is pre-defined in Array.prototype that creates a new stream applying a function against an accumulator and each element in a fixed (size) subset of the stream. In the example, the accompanying parameters specify the multiplication as the method, 0 as the initial position, and 2 as the size of the accumulator. The edge engine uses its script parameters to form a mathematical function. The formed function of the current example is ‘window(y = xi*xi+1*xi+2)’, where ‘i’ starts from 0 and increments till the end of the array with a step of three (accumulator +1). The custom operation ‘absolute’ does not exist in Array.prototype, thus the operation code must be provided as part of the script. The edge engine will inject this new code into the Array.prototype constructor and execute it upon call.

Script 3. JSON script example with two operations, one pre-defined and another custom

```
{
    …
    "command": {
      …
      "operation": [
            {
            "type": "window",
            "params": "*, 0, 2"
            },
            {
            "type": "absolute"
            }
      ],
      "custom": "Array.prototype.absolute = function () {
                  var newArray = [];
                  for (i = 0; i < this.length; i++) {
                        newArray[i] = math.abs(this[i]); }
                  return newArray;}"    }
}
```

The resulting data from ‘Data Input and Processing’, whether raw or processed, are handed to the subroutine ‘Data Storage and Cloud Upload’, where the stack of arrays is saved into a local buffer that preserves the data until an upload to the cloud is accomplished. A process tests for cloud connection availability, the success of which allows a measurement POST with the whole buffer in the body part of the HTTP request. Simultaneously, a GET request is issued for the script on the cloud (using the saved Script Id from the initialization phase). Once the new script is received, the edge engine updates its local script. To avoid data accumulation, the local buffered is erased after successful

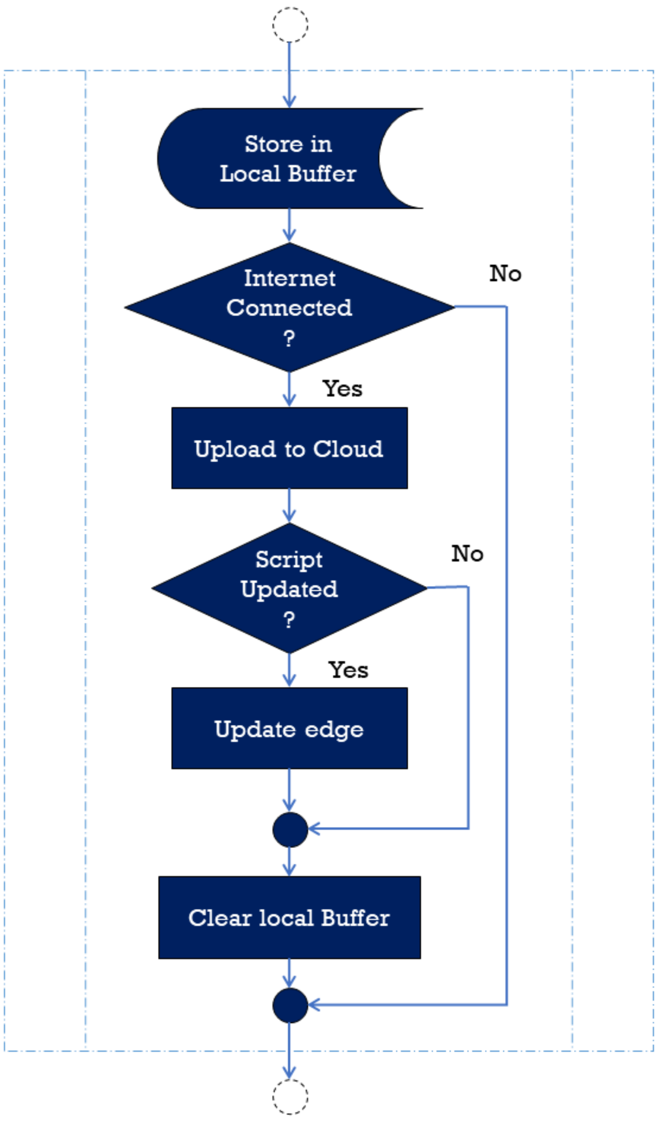


Fig. 16. Flow chart of the Data Storage and Cloud Upload subroutine.

upload to the cloud. Figure 16 shows the internal processes composing the ‘Data Storage and Cloud Upload’ subroutine.

The local buffer grows in size with the increase in the difference between the scheduled intervals of the two subroutines denoted by ‘uploadInterval’ and ‘readInterval’.

The third subroutine in the Asynchronous Event Loop is the Login subroutine, which is presented earlier in Figure 12. The purpose of the ‘Login’ subroutine is to keep generating a valid JWT for the HTTP requests issued by the preceding subroutine ‘Data Storage and Cloud Upload’. The time interval for the schedule managing this subroutine, the ‘loginInterval’, must be less than the expiration interval of a JWT specified by the cloud API.

Chapter 3 – Use Cases (Experiments)

Section I – H@H

Health at Home (H@H), a domestic project funded by the Italian Ministry of economic development and aimed at supporting the elderly with Chronic Health Failure (CHF) [58], developed a complete IoT Cloud service consisting of the Home Monitoring Sensor System (front-end), the Home Gateway (middleware), and a Remote Cloud (back-end). Through the Gateway, several physiological quantities (electrocardiogram signal, heart rate, breathing waveform, breathing rate, oxygen saturation, blood pressure, glycemia, etc.) are collected and provided to the cloud. Through a web-based user interface, a clinician can view the measurements, and modify the pharmacological therapy according to the symptoms. Atmosphere acted as the backbone of the application in order to implement the H@H API described in [59]. Atmosphere had to accommodate the needs and usage variations of different service providers, as the front-end (edge) and the gateway (fog) were provided and maintained by third parties in the health industry.

The mapping of the H@H quantities onto the Atmosphere resources was straightforward: for each physiological signal, we create a corresponding "feature" and for each sensor a "device", in order to collect information acquired on patients as "measurements" in Atmosphere. The mapping between H@H concepts and Atmosphere resources provided in Table III. The 'operator' entity, a new instance of the user resource, was required, with a special set of behaviors and permissions. We implemented this as an extension to the main ‘User’ resource exploiting the Object-Oriented Programming (OOP) paradigm.

TABLE III
ATMOSPHERE RESOURCE MAPPING TO H@H MODELS

Resource	H@H data
Measurement	A record containing the value of a medical indicator (e.g., breath rate, heart rate, body temperature)
Feature	Medical indicators. For instance: breath rate, heart rate, body temperature, etc.
Device	Different types of medical devices (e.g., heart pulse sensor, blood pressure sensor)
Thing	The physical object being monitored (e.g. patient, organ)
Tag	Associated medication and active treatment (e.g. physical therapy)
Constraint	Association between things and services
Computation	Outlier detection
Service	Specific health monitor services (e.g., post-surgery rehabilitation, ambulance call)

In H@H, several service providers offer various e-Health services (e.g., post-surgery rehabilitation support, daily activity monitoring, pain self-assessment, etc.). This required a management framework to organize the services on both the patient and the provider side. Thus, we implemented a publish/subscribe (pub/sub) pattern, through a supplementary Subscription resource. Atmosphere takes advantage of asynchronous Webhooks to provide automated real-time callbacks. Webhooks broadcast any service update by the service providers to all subscribers. The new resource couples the users with their subscribed services (the ones that are supported by their installed edge sensory system). One field (IsActive) specifies whether the payload delivery is enabled, while another (Endpoint) specifies the Uniform Resource Locator (URL) address to which the service and later updates must be published. Figure 17 shows a demonstration of roles assigned to entities in H@H within the Pub/Sub pattern. Webhooks manages service topics between service providers and patients. In this pattern, patients are subscribers to e-Health services while service providers are publishers of those services and subsequent updates.

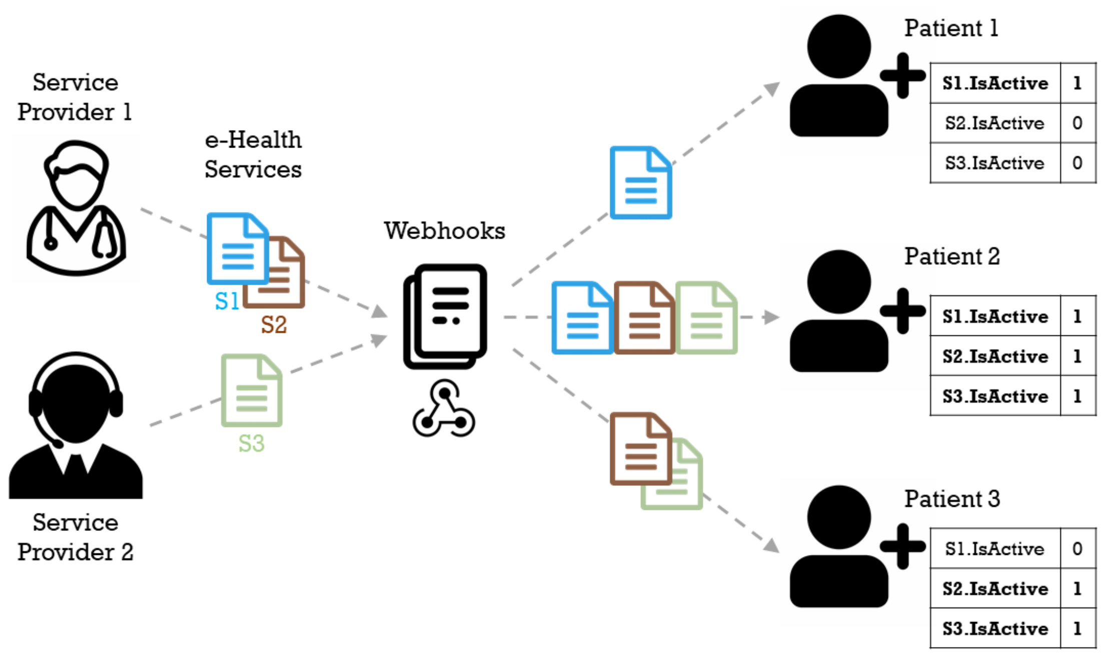


Fig. 17. Pub/Sub pattern in Atmosphere use case H@H using Webhooks.

Section II – FABRIC

In Fabric, a 7th Framework Programme European industrial research project which implemented an on-road testbed for Dynamic Wireless Charging (DWC) of electrical vehicles (EVs) [60], we realized a charging process metering service for the vehicles passing through the charging lane [61, 62]. The system senses and computes on the edge information about the charging process and stores it on Atmosphere's cloud server to support new electro-mobility (e-mobility) services (e.g., billing, energy-aware car navigation) which can be implemented by relevant companies (e.g., energy providers, navigation providers). Figure 18 shows the ecosystem of the wireless charging of electric vehicles including the road power supply infrastructure as well as the IoT infrastructure.

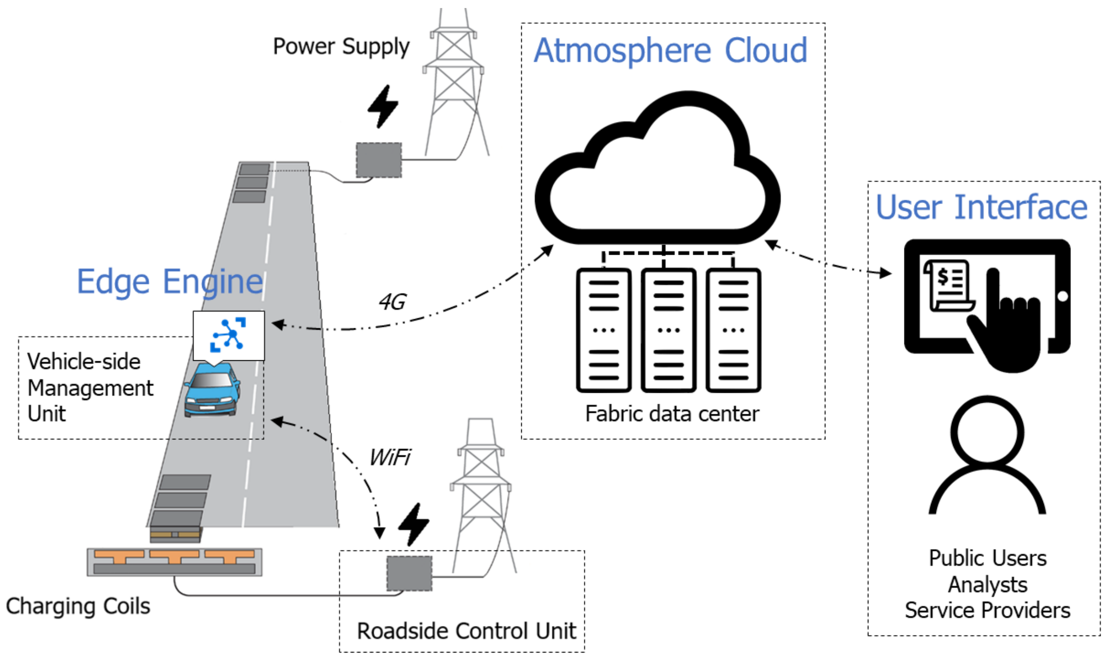


Fig. 18. Integration of Atmosphere components (Edge engine and Cloud) into the Fabric ecosystem.

A. Edge

The edge engine was hosted on a Raspberry Pi 3b on the vehicle-side. Also running on the host device was a Grid Alignment Assistant System (GAAS). Measurements from which were sent to the edge engine internally (within the Raspberry Pi itself, from one memory buffer into another), while other measurements from the vehicle-side DWC electronic sub-system were sent through a wired Ethernet connection and measurements from the road-side DWC electronic sub-system were sent through wireless Ethernet connection (WiFi). Thus, the edge engine collects data from three IoT sources, processes each, then uploads it to the cloud.

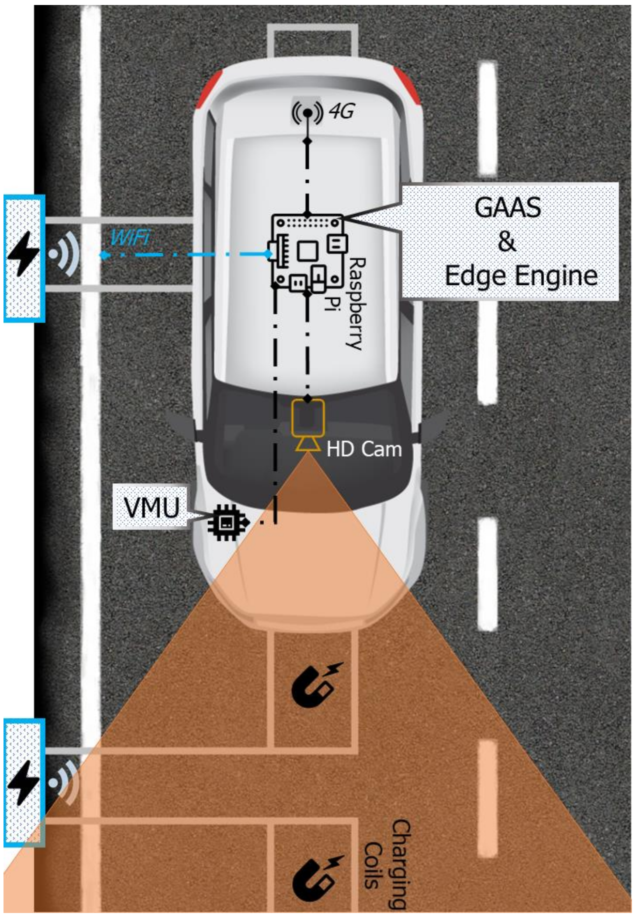


Fig. 19. Fabric edge deployment schematic.

GAAS computes in real-time the misalignment between the axis of the vehicle and that of the charging grid (i.e., the coils) in the road (described in Figure 19). As an optical sensor, GAAS used the Logitech c920, a high definition (1920 x 1080) webcam to capture 30 frames per second of the road ahead. The webcam performs automatic luminosity adaptation. Mounted on the top-middle of the windshield, using a suction cup, the camera is connected to a server-node Raspberry Pi 3 running Python 3.7.0. The Python server node processes each frame, estimating the current position of the center of the vehicle w.r.t the center of the grid. The computed alignment offset (in cm) is displayed to the driver as a pointer across a gauge scale as part of the Human-Machine

Interaction module implemented in the on-board unit (OBU) on a tablet. The alignment offset is also sent, through Atmosphere's edge engine, to Atmosphere cloud for storage and further analysis. The alignment offset is passed over to the local buffer within the RAM of the Raspberry Pi edge engine as is without further processing. The edge engine fills in the suitable attributes for this particular measurement and executes a POST to the cloud.

The road-side control unit (CU) sends to Atmosphere data representing the state of each consecutive charging grid. The vehicle management unit (VMU) generates a stream of measurements recording the vehicle-coil alignment (which is key to power transfer efficiency and was supported by a vision system [63]) and the charging parameters and battery status. In an approach suitable to the nature of each data stream, we implemented the edge-computing module to process data in two different ways. The road-side uploads to the cloud when a change in measured values occurs. That was accomplished by implementing the custom operation 'ComparePrev' within the edge engine. Since the values collected are already stored in the buffer, this function blocks the upload of the newest values of the record if they were equal to their previous values. As part of the edge script, the function ComparePrev is written as: `"custom": "Array.prototype.ComparePrev = function () { for (i = 0; i < this.length; i++) { if (this[i] != prevBuffer[i]) {this.splice(i, 1); } } }"`.

Due to the oscillating nature of the data generated by the vehicle-side sub-system, it was required to average the measurements every specific batch. To control such batches, I manipulate the processing rate with respect to the data input rate. Thus, vehicle-side data were averaged within a predefined period. We used the custom operation 'Average' which incorporates a pre-defined method 'reduce' as the following snippet of the script: `"custom": "Array.prototype.ComparePrev = function () { this = (this.reduce((accumulator, currentValue)=> accumulator + currentValue))/this.length } } }"`. The method 'reduce' is used for summing the values of the array to be divided by the array length.

The host device was connected to the internet – for cloud access – via 4G LTE mobile network. The edge engine performed all three required local computations on top of the running GAAS system (live image processing instance in python). Furthermore, the edge engine's local buffer was exploited to deal with the intermittent connection between the edge and the cloud.

B. Cloud

The stored values represent the electrical charging process in current and voltage. Figure 20 shows the real measured values of the current in Ampere of a charging session. The mapping of the Fabric objects onto the Atmosphere resources is reported in Table IV. Further parameters are required to contribute to the metering and billing services such as the electrical power transfer, energy stored, and final electrical bill. Acquiring these data was made possible through the Computation resource.

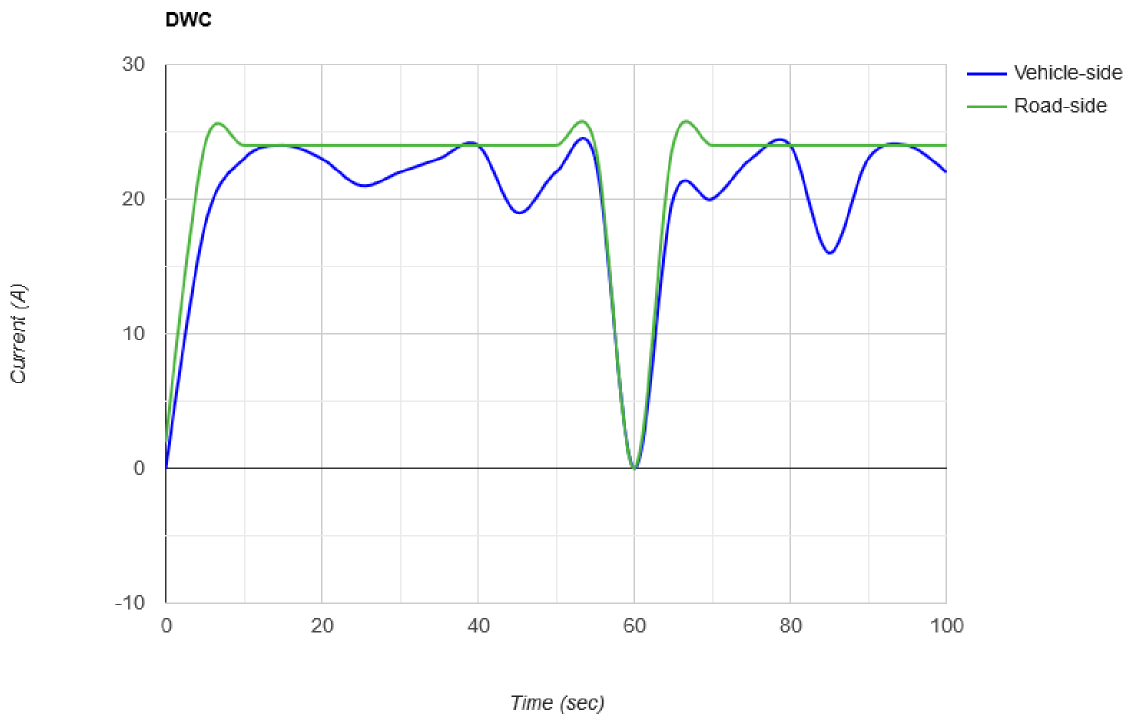


Fig. 20. Raw charging parameters of a FABRIC trip recorded and presented with Atmosphere.

Power transfer is computed directly from the stored measurements. But computing energy requires integrating the result of previous power computations. I thus altered the computation resource to get the source data from the Computation collection as well as the Measurement collection. The alteration concerns a new attribute in the computation resource to specify the target resource of the computation (i.e. the data resource) and that

TABLE IV
ATMOSPHERE RESOURCE MAPPING TO FABRIC MODELS

Resource	Fabric data
Measurement	A record containing the vehicle-coil alignment estimation, or the values of the charging process
Feature	Measured quantity. For instance: vehicle-coil displacement, charging parameters (with dimensions: current, voltage, speed, etc.)
Device	Vehicle-coil alignment system, Dynamic Wireless Charger (DWC) road and vehicle side
Thing	Passage of an electric vehicle in the charging road lane
Tag	Charging conditions (e.g., preparing, charging, fault)
Constraint	Association between features and services
Computation	Outlier detection, Power, Energy
Service	Vehicle-coil alignment, Wireless charging, Billing, Lane keeping

could be a measurement resource or a computation resource. For instance, computing the power constitutes of a POST request to the computation resource with the request body shown in Script 4. The header of this HTTP request would filter the desired measurement by trip: `filter={"thing": "Charging_Trip"}`.

Script 4. Power Computation Script

```
{
      "code": "multiply()",
      "name": "power-computation",
      "target": "measurements",
      "items": {
                "item1": "voltage",
                "item2": "current",
               }
}
```

The result of the request is a computation record with an array of time-series power values. Another request is issued to compute the total power consumption, this time targeting the computation resource. To distinguish between the targeted data source, the header would contain a filter with the name of the computation: `filter={"name": "power-computation"}`. The result of the request, whose body is shown in Script 5, is a value representing the total power consumption for the recoded charging trip and the time of the trip in seconds (delta time).

Script 5. Total Power Computation Script

```
{
      "code": "add()",
      "name": "total-power-computation",
      "target": "computations"
}
```

Using the total power and the time frame of the charging trip, the energy can be computed using Script 6. Given the pricing schematics from the service provider (wireless charging), billing for the charging trip I thus available per consumed energy.

Script 6. Energy Computation Script

```
{
      "code": "multiply()",
      "name": "energy-computation",
      "target": "computations"
}
```

The whole end-to-end deployment was tested in the lab and (the vision module for vehicle-coil alignment) on the test site inside the MotorOasi Piemonte safe drive track in Val di Susa, Italy. Through enabling edge computing functionalities, the edge device managed to drop data upload size by approximately 96% from 720 MB/h to 27 MB/h.

Based on stored data, we performed tests in lab tests for a simple prototype billing service. In a three-step procedure, starting with power (KW) then energy (KWH), we were able to obtain an estimate of the electrical bill for a specific charging lane passage. In all cases, concurrent and consecutive HTTP requests were maintained at a stable 40 ms delays in response time.

Section III – L3Pilot

L3Pilot [64] is a Horizon 2020 research project aimed at assessing the impact of automated driving (AD) on public roads, testing the Society of Automotive Engineers (SAE) Level 3 (and some Level 4) functions [65]. The pilots involve 1,000 test subjects, 100 cars, by 12 vehicle owners (either Original Equipment Manufacturers, or suppliers), across 10 European countries. The project uses the Field opErational teST support Action (FESTA) methodology [66], driven by a set of research questions and hypotheses on technical aspects, user acceptance, driving and travel behavior, as well as the impact on traffic and safety.

In order to answer such research questions, the project has defined a data toolchain, that translates the proprietary vehicular signals to a shared format [67], and processes them to extract the driving scenario (e.g., “lane change”, “cut-in”) and other event information [68, 69]. This toolchain, developed by L3Pilot partners, eliminates the need for the edge engine. Filtered data, aggregated from all the pilot sites, needs to be analyzed for an overall impact assessment. For that, Atmosphere provides a shared data storage back-end [70]. Figure 21 shows the data management process initiating from the autonomous vehicles, through the toolchain, and into Atmosphere cloud.

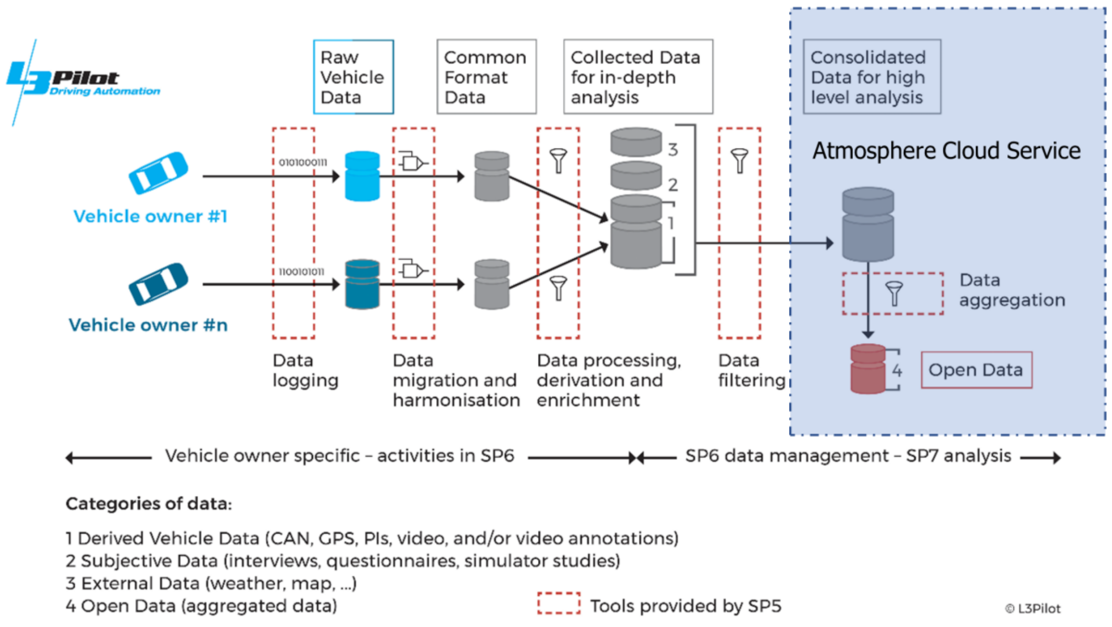

L3Pilot Driving Automation
Raw Vehicle Data
Common Format Data
Collected Data for in-depth analysis
Consolidated Data for high level analysis
Atmosphere Cloud Service
Vehicle owner #1
Vehicle owner #n
0101000111
1100101011
3
2
1
4
Data aggregation
Open Data
Data logging
Data migration and harmonisation
Data processing, derivation and enrichment
Data filtering
Vehicle owner specific – activities in SP6
SP6 data management – SP7 analysis
Categories of data:
1 Derived Vehicle Data (CAN, GPS, PIs, video, and/or video annotations)
2 Subjective Data (interviews, questionnaires, simulator studies)
3 External Data (weather, map, ...)
4 Open Data (aggregated data)
Tools provided by SP5
© L3Pilot

Fig. 21. High-level schema of the overall data management architecture in L3Pilot.

The mapping of the L3Pilot objects onto the Atmosphere resources is reported in Table V, with a Thing (i.e., the subject of a Measurement) corresponding to every single experimental trip. Stored data – produced by the above-mentioned toolchain - is not the original signal time-series, but meaningful aggregations (called “Datapoints” and various types of “Indicators”, with various types of items such as: avg speed, minimum longitudinal distance, time headway at minimum time to collision, percentage of driving times in the various scenarios, etc.). Thus, these “Datapoints” and “Indicators” are the Features in the L3Pilot installation. A complete list of L3Pilot features is presented in Table C1 of Appendix C. Different driving scenario types have different datapoint structures. All these features are tagged as Datapoint to facilitate data retrieval according to the jargon. Tags were very useful also to specify driving scenarios, experimental conditions (baseline, system available, system active, etc.) and road types (e.g.,

TABLE V
ATMOSPHERE RESOURCE MAPPING TO L3PILOT MODELS

Resource	Fabric data
Measurement	A record containing all the values of a Driving Scenario Instance (DSI) Datapoint, or of the Performance Indicators (PI) per Trip, or of the PI per SI
Feature	Measured quantity (e.g., Trip PI, DSI Datapoint).
Device	Dimensions include: average speed, time headway at maximum speed, percentage time in each type of driving scenario, etc.
Thing	The L3Pilot data toolchain
Tag	Trip (i.e., the unit of the information source processed by the toolchain)
Constraint	Driving scenarios (e.g., traffic jam, cut-in), road types (e.g., urban, motorway) and experimental conditions (e.g., baseline)
Computation	Association between higher and lower order features, and between tags and features.
Service	Outlier detection

motorway, urban), that are all used to segment a trip's data. A list of the used tags for L3Pilot populates Table C2 in Appendix C.

The Constraint resource was introduced to define abstract relationships (e.g., dependencies) between two documents in the DB. The web-browser-based graphical user interface (GUI) – that was developed by another project partner [69] – exploits Constraint documents in order to allow the data analyst user to select the available measurement types from dynamically populated drop-down menus. To this end, constraints were used for relating tags among each other. Tags, in fact, are used to specify driving scenarios, experimental conditions (baseline, system available, system active, etc.) and road types (e.g., motorway, urban), that are all used to segment a trip's data. The measurement resource is also modified accordingly, to allow specifying features and items with different dimensionalities. The validation of the value vector size with the item's dimension size is done through two checks on both specified features.

The toolchain processes offline the data gathered during a pilot vehicle's trip and POSTs the resulting measurements in batches to the DB. Batch sizes differ from one trip to another, and we monitored the effect of batch sizes on the API response time. As Figure 22 shows, the response time is almost linear with respect to the number of

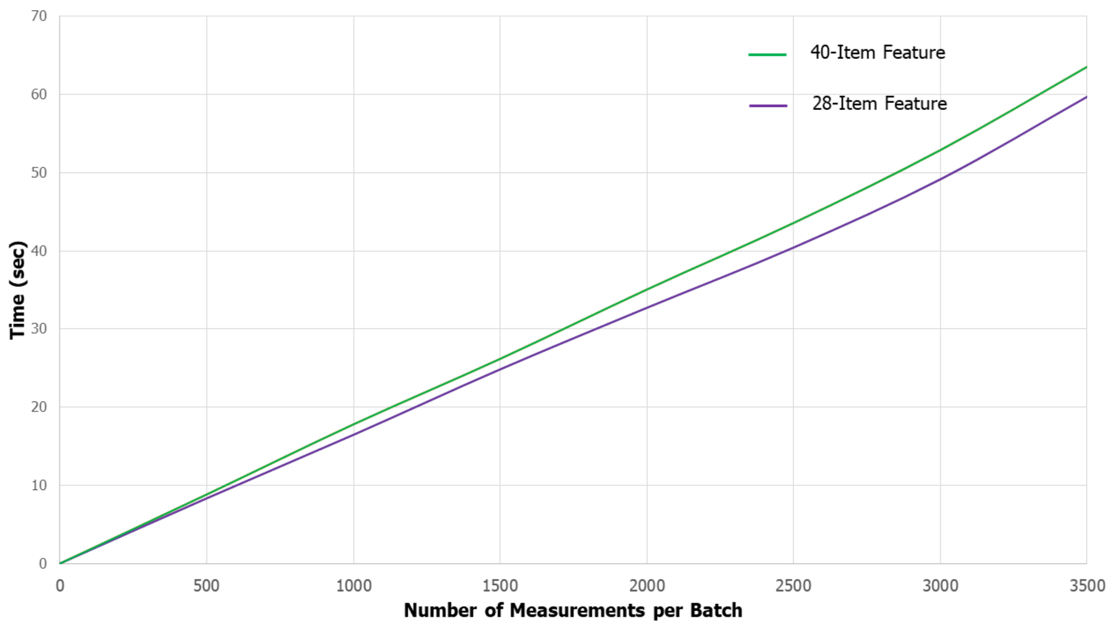


Fig. 22. Line graph of Atmosphere's API response times to batch uploads by L3Pilot with different batch sizes of two types of measurements (one with 28 dimensions and one with 40 dimensions).

measurements per batch. The size of the sample vector (feature's items) has a minor impact.

Examining the effect of the batch approach on the single measurement response time. On a rough analysis, we observed a significant improvement in response times down to 17 ms on batches with a larger number of measurements. Figure 23 shows a decline in response time, which settles at batches with 30 measurements. The addition of the support of HTTPS had only a minor impact, even while enabling SSL certificate verification. We observed a +/- 4 ms difference at most between encrypted and unencrypted requests, with a slight peak of 1% in CPU load upon HTTPS connection.

To POST measurements to Atmosphere, vehicle owners are given 'Provider' credentials. They are owners and can see and manage (through the web UI) their own data only. Analyst users can see all the measurements. The admin user has full control over all the data and defines the domain by posting Features, Constraints, Device (the L3Pilot toolchain) and Service (only one, L3Pilot), according to the Atmosphere workflow described in chapter 2, section II.B.5.

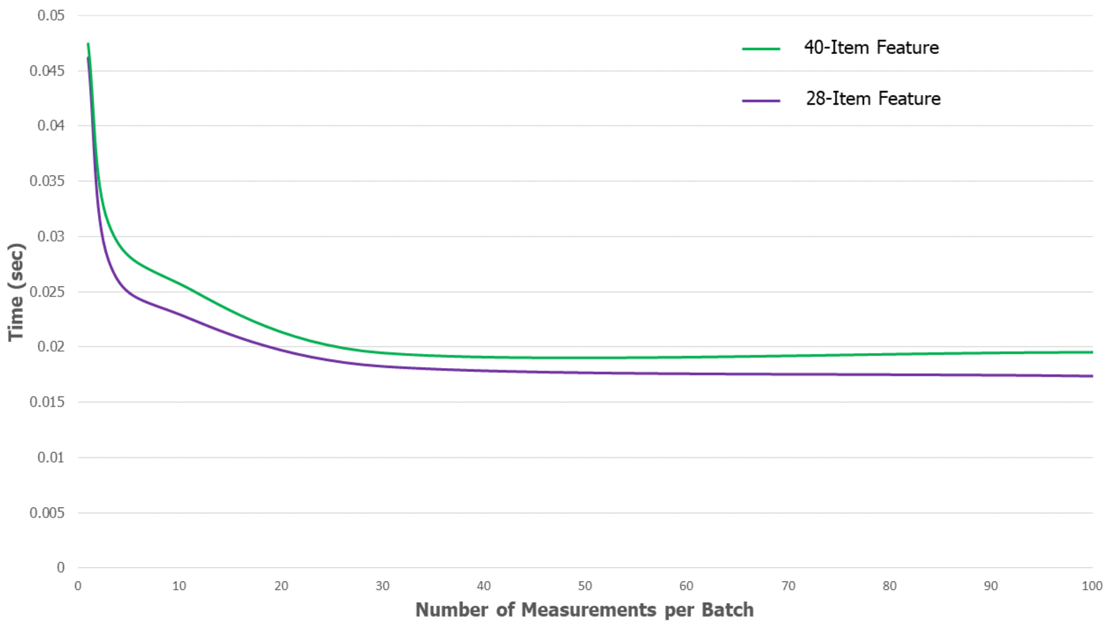


Fig. 23. Line graph of Atmosphere's API relative response times of single measurements to batch uploads by L3pilot with different batch sizes in close range (1 to 100).

Section IV – Home Automation

We designed a lab experiment to test the performance and parametric limits of the edge engine deployment on a Raspberry Pi 3 b [71]. The experiment [72] was designed to simulate a smart home IoT environment, shown in Figure 24. It included up to 16 sensors, wire connected to the GPIO (general-purpose input/output) port of the Raspberry Pi. Those sensors consist of 4 dual temperature and pressure sensors, 4 switch sensors, 3 photodetectors, 3 passive infra-red (PIR) sensors, 1 humidity sensor, and 1 moisture sensor. These sensors have different polling rates, with the fastest at 100 Hz frequency reached by the PIR sensor. That indicated that the minimum delay that still captures a change in measurements from the sensors is 10 ms.

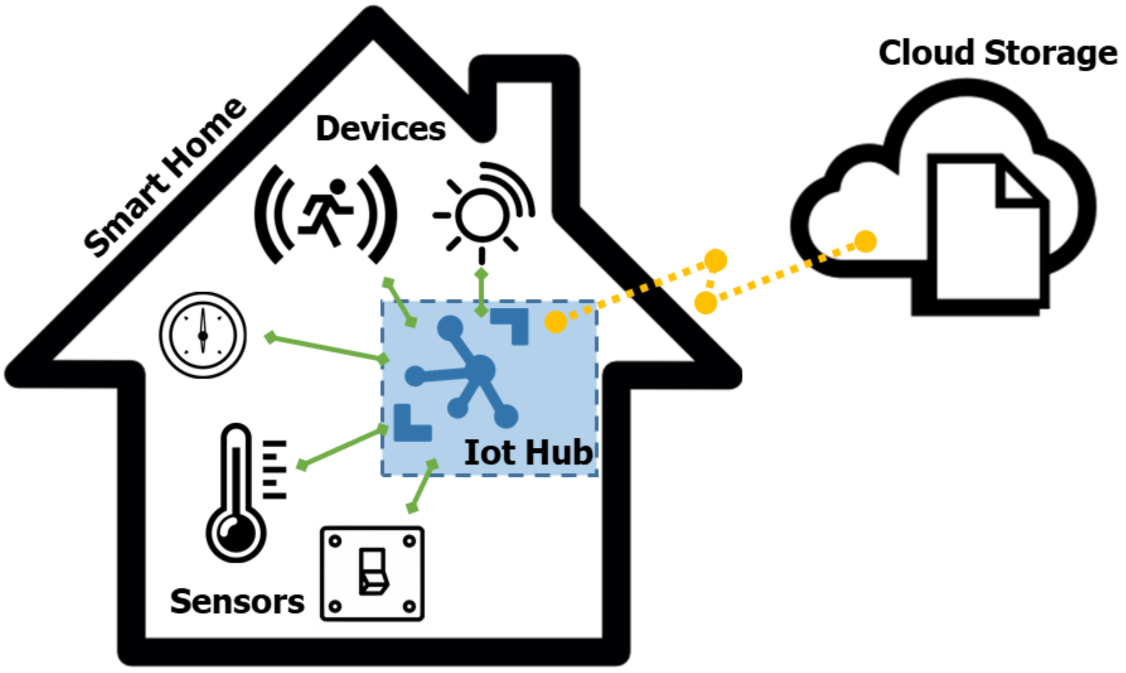


Fig. 24. Block Diagram depicting the Home Automation Simulation Experiment.

In the simulation, I experimented with multiple versions of the edge scripts. Each script specifying different delay parameters for input reading from sensors and output uploading to the cloud. Local processing operations were varied as well. These constitute the two variable factors across the experiments in this simulation. The minimal script would have the same values for ‘readInterval’ and ‘uploadInterval’ and no active local processing operations, which results in an instantaneous stream upload of raw data to the cloud. The maximal script would have the ‘uploadInterval’ 20x the ‘readInterval’ and four consecutive local operations applied on each data stream. That amounts to 6400 executed operations per second on the raw input, and 1280 new entries of processed data into the local buffer before uploading to the cloud then clearing the buffer. Script 7 includes a sample of the edge script used in the home automation simulation at maximal operating configurations.

Script 7. Sample of the edge script for Home Automation

```
{
    "_id": "Home_Automation_Script_1",
    "device": "Thermostat",
    "tags": ["Edge_Device", "Heating",],
    "command": {
      "method": "POST",
      "resource": "measurements",
      "thing": "Smart_Home",
      "feature": "Heating",
      "readInterval": "5",
      "uploadInterval": "100",
      "loginInterval": "1798000",
      "operation": [
            {"type": "window", "params": "*, 0, 2"},
            {"type": "absolute"},
            {"type": "filter", "condition": "value > 30"},
            {"type": "map", "condition": "value * 9/5 + 32"}
      ],
      "custom": "Array.prototype.absolute = function () {
                  var newArray = [];
                  for (i = 0; i < this.length; i++) {
                        newArray[i] = math.abs(this[i]);
                  }
                  return newArray;}"
            }
}
```

The change in script parameters allowed me to test the engine’s auto-check for script modifications on the cloud. This feature explained in chapter 2, section III.B.2, allows the engine to update its local script based on the cloud version of the script. Varying the delay intervals for reading and uploading allowed me to test both the edge engine’s and the cloud API’s performance with respect to different upload fashion within the same experimentation scenario and environment, which is the smart home. Further contextualized discussion of the methods and results of this experiment are discussed in the following section.

Section V- Discussion

The deployment of Atmosphere cloud in three diverse IoT applications showed its effectiveness, flexibility, and ability to support an efficient workflow. Additionally, we were able to integrate new functionalities in an abstract manner, keeping the reusability objective valid across all resources and methods. Table VI summarizes the mapping between Atmosphere and the three project models.

While the other two projects are finished, L3Pilot is in progress, and a full account of Atmosphere's cloud deployment will not be available before the project concludes in 2021. However, requirements by L3Pilot partners and pilot tests are being satisfied efficiently by Atmosphere, which showed a great deal of versatility in meeting the automotive requirements, also considering that we dealt with statistically pre-processed data, with complex semantic structures. When upgrades were needed (e.g., for increasing the dimensionality of each measurement sample), we managed to keep the deployment delay upon structure change relatively low (a couple of days at most). L3Pilot allowed testing and upgrading Atmosphere to the use case in which different IoT data source providers share post-processed data, typically in batches. The structural data checks implemented in Atmosphere allowed detecting bugs in the complex data preparation toolchain (e.g., in terms of dimensions), which saved significant development time.

Across the three use cases, not only did the deployment environment change but the manner of data streaming to the cloud as well. In H@H, we experienced a steady stream of raw data, Fabric uploads contained a high frequency of aggregated data, and in L3Pilot we implemented batch upload of pre-processed data. Upon examination of each use case, we extracted the relevant parameters that give an insight into the efficacy of Atmosphere's deployment. We spotted a remarkable difference in API response time to stream POSTs (one measurement with a single or multi-dimension value vector) versus batch POSTs. Stream POST requests resulted in an average 40 ms response time. Batch POSTs contained up to 4,000 measurements, with multiple dimensions (5 to 40) value vectors. Figure 25 shows the response times of two batch uploads to Atmosphere with different item sizes. Each batch POST cost around one minute, for 3,500 measurements with 40-sized value vectors, resulting in a per measurement cost of 17ms on average.

TABLE VI
ATMOSPHERE RESOURCE MAPPING TO PROJECT MODELS

Resource	**H@H data**	**Fabric data**	**L3pilot data**
Measurement	A record containing the value of a medical indicator	A record containing the alignment estimation, or the charging parameters	A record containing all the values of a Datapoint, or of the Performance Indicators
Feature	Medical indicators. For instance: breath rate, heart rate, body temperature, etc.	Measured quantity. For instance: vehicle-coil displacement, charging parameters	Measured quantity (e.g., Trip PI, DSI Datapoint).
Device	Different types of medical devices (e.g., heart pulse sensor, blood pressure sensor)	Vehicle-coil alignment system, Dynamic Wireless Charger (DWC) road and vehicle side	Dimensions include: average speed, time headway at maximum speed, percentage time in each type of driving scenario, etc.
Thing	The physical object being monitored (e.g. patient, organ)	Passage of an electric vehicle in the charging road lane	The L3Pilot data toolchain
Tag	Associated medication and active treatment	Charging conditions (e.g., preparing, charging, fault)	Trip (i.e., the unit of the information source processed by the toolchain)
Constraint	Association between things and services	Association between features and services	Driving scenarios (e.g., traffic jam, cut-in), road types and experimental conditions
Computation	Outlier detection	Outlier detection, Power, Energy	Association between higher and lower order features, and between tags and features.
Service	Specific health monitor services	Vehicle-coil alignment, Wireless charging, Billing	Outlier detection

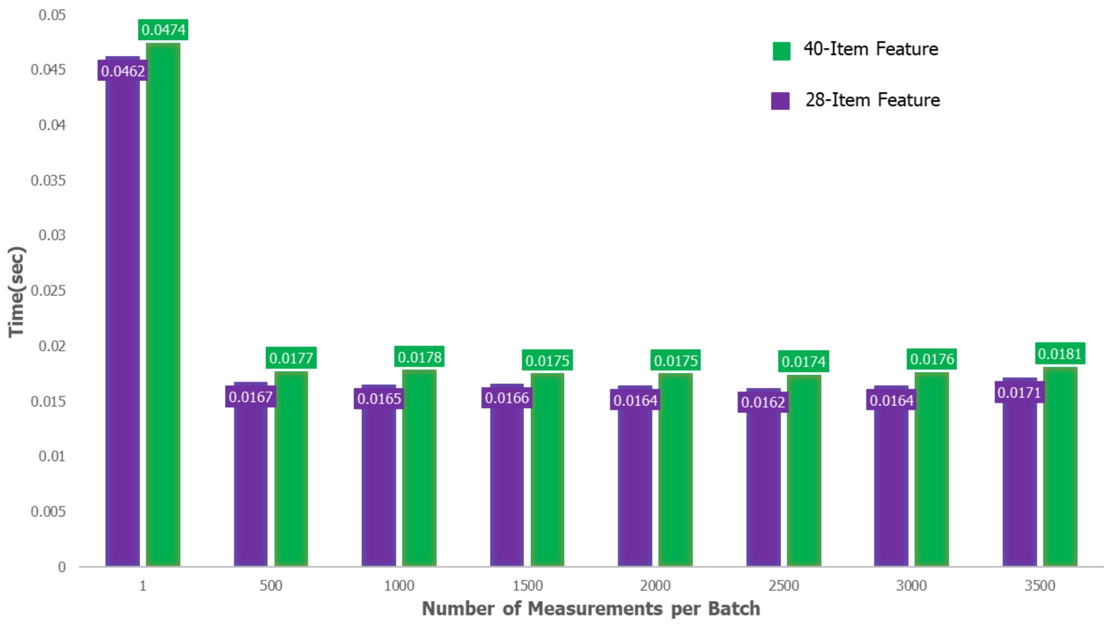


Fig. 25. Bar graph of Atmosphere's API relative response times of single measurements to batch uploads by L3pilot with different batch sizes.

We argue that the reason for this contrast in response time per measurement between the stream and batch POSTs is partly because of the enabling of the response compression using the 'npm compression' middleware and partly due to overhead avoidance of extra HTTP requests/responses header upon large batches.

The consecutive deployments of Atmosphere cloud in three diverse IoT applications served the proof-of-concept profoundly. Additionally, we were able to integrate new functionalities into Atmosphere in a generic manner keeping the reusability objective valid across all resources and methods.

Regarding the edge engine deployments, in Fabric and the Home Automation Simulation, multiple requirements were experienced. The feasible script updates, the simplicity in supported functions, and the high customizability of the edge engine as a whole as well as the operational functions within were all experienced in the two mentioned deployments.

In Fabric, we used the edge engine to collect IoT measurements from three sources, each with a different processing operation, stored the results locally, then uploaded seamlessly to the cloud. Grid alignment and billing information were delivered to the cloud in real-time. Moreover, the engine ran on a busy host (Raspberry Pi running GAAS), this is only possible because it is designed to be lightweight and very suitable for low-end hardware devices that are typical for IoT settings. We maintained a proper balance between the edge engine configuration and the GAAS configuration to run both instances on the same device while preserving high accuracy and performance. We particularly selected the highlighted configurations in Table VII and Table VIII to run simultaneously. The sensory delay of 10 ms for the edge engine configuration is very suitable since it corresponds to the fastest polling rate of the connected devices (i.e. VMU and road-side CU). In the GAAS configuration, a 5x5 kernel size was chosen since it provided the same accuracy as a 7x7 kernel with less CPU load and better FPS performance. Selecting these configurations together amounts to 62 % of CPU load which leaves enough resources for an unexpected influx in CPU demand and serving the edge engine and GAAS simultaneously.

TABLE VII
EDGE ENGINE HARDWARE TEST ON THE RASPBERRY PI 3 B

Sensory Delay (ms)	**CPU**	**Freq. (MHz)**	**Threads**	**RAM**	**Network (KBps)**
1	90%	1200	4	7.4%	3.0
5	55%	1200	3	7.4%	0.6
10	28%	1200	2	6.5%	0.3
50	13%	1200	1	6.4%	0.06
>100	<10%	600	1	<5%	<0.03

TABLE VIII
GAAS HARDWARE TEST ON THE RASPBERRY PI 3 B

Kernel size	CPU	Freq. (MHz)	Threads	FPS	Network (KBps)	Acc. (<10 cm)	Acc. (<20 cm)
9x9	99%	1200	4	3	1.5	100%	100%
7x7	71%	1200	3	10	2.4	80%	100%
5x5	34%	1200	2	24	2.9	80%	100%
3x3	20%	1200	1	28	3.0	5%	20%
Idle	3%	600	1	30	0	-	-

In the Home Automation simulation, the performance test was performed to prove the applicability and operability of the edge engine on low-end devices. The experiment resulted in two main observations as presented in Figure 26 (shows CPU load of different edge engine scripts) and Figure 27 (shows RAM usage of different edge engine scripts). The variation in the number of incorporated operations for local processing had little to no effect on the CPU load or RAM usage. The CPU usage reached its maximum at 90% with 4 threads running on the 4-core CPU at the minimum limit of possible input stream delay at 1 ms. The typical delay of 10 ms for input stream corresponded to 27% CPU usage with 3 running threads. Such usage is acceptable considering the number of input streams (16) and computations (4) running 100 times per second.

The other observation, which concerned the memory usage was unexpected. The test recorded a decline in memory usage in regards to higher output stream delays. One explanation for this observation is the cash management mechanism within the Raspbian OS, which keeps the buffers that were cleared by the engine saved for a while. Therefore, the more buffers are cleared by the engine in a smaller timeframe, the more buffers the OS is cashing. The amount of used RAM varied from 4 to 8 percent that is 40-80 MB of the available 1 GB. We measured a steady 60% of the RAM usage (between 30 and 50 MB) occupied by the OS. The edge engine approximately uses up to 30 MB on the highest configuration. This is substantially comfortable with respect to the Raspberry Pi

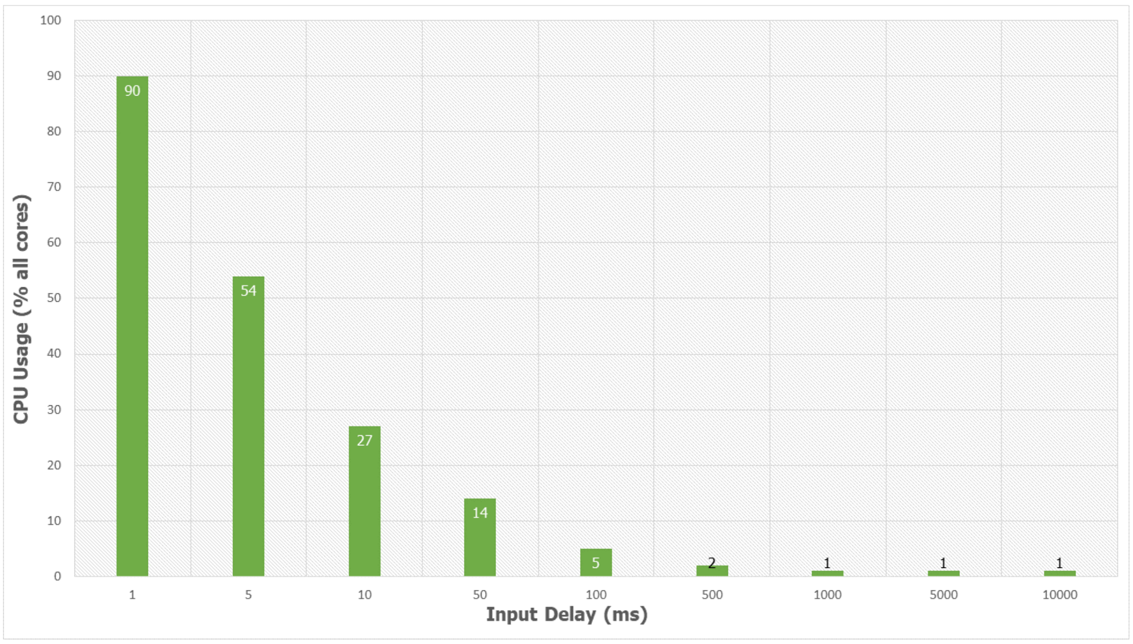


Fig. 26. Raspberry Pi 3b Edge Engine CPU Performance Test.

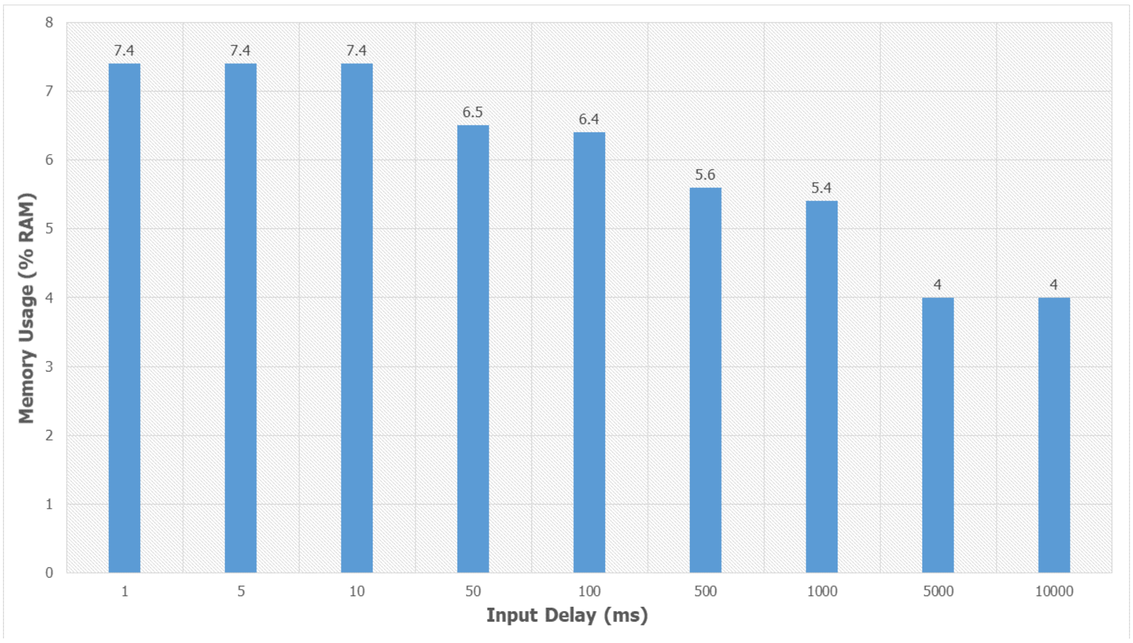


Fig. 27. Raspberry Pi 3b Edge Engine RAM Performance Test.

specifications but could be more challenging to manage on hosts with less memory space such as the devices presented in Table B1 in Appendix B.

The two factions of the developed Atmosphere solution, the edge engine, and the cloud API, demonstrated our manifestation of the proposed concept: a generic IoT edge-to-cloud solution for smart cities. Compared to similar systems and solutions in the literature as well as in the tech market, Atmosphere is distinguished by its comprehensive breadth, abstraction, and customizability. While it is still being updated frequently, I contributed to developing Atmosphere's core and establishing valid and tested resources and methods.

According to our goals, Atmosphere is presented as a deployment-ready IoT SaaS framework. To that end, we can judge the usability of Atmosphere in comparing its deployment time with a start-from-scratch approach on a commercial or open-source alternative (presented in chapter 1). When deploying Atmosphere, the only preparatory stage is mapping, where IoT specific labels and entities have to be mapped to the API's resources. Using an alternative, deployment would require more stages to configure including attribute definition, user roles, relations (and references) between the resources, security (authorization and authentication), methods calibration, and automated test suite for routes and methods. Such tasks could take a considerable amount of time that can be avoided with minimal to no compromise by adopting Atmosphere. Furthermore, customizing Atmosphere has been proven to be a feasible task and one that can adhere to the generic concept for reusability. Customizable space in Atmosphere is vast; almost any resource can be customized to fit IoT industrial applications where measurements are numeric in nature. Most of these properties are not available in alternatives either commercially or open-source. Whether on the cloud, on the edge, or both, we easily deployed Atmosphere in diverse IoT projects under the smart city concept.

Chapter 4 – Conclusion

As IoT technologies are increasing the capabilities of collecting huge quantities of data from the field, it is ever more important to have tools for creating new, data-rich applications. This book has presented Atmosphere, an independent, abstract, measurement-oriented edge-to-cloud framework for managing smart things in the IoT ecosystem. The original contribution of our work consists of proposing an edge-to-cloud computing model using abstract IoT web resources and relations to reflect the structure and functionality of cross-domain IoT applications. In the context of this project, our target users are mainly IoT developers and service providers, and in order to support the IoT developer community, we release Atmosphere open source on GitHub: https://github.com/Atmosphere-IoT-Framework.

Atmosphere has been designed as a deployment-ready IoT data storage and computation support service. It focuses on measurement data and exposes key resources (including Computations and Machine Learning) to support measurement-rich applications.

Our experience in three industrial research projects in addition to an in-lab simulation of the smart home – while quite various in nature – showed that the tool can seamlessly support a variety of IoT applications, providing benefits in terms of efficiency and effectiveness, as its resources support a structured and modular approach to application modeling and development. Atmosphere does not tie the development to a proprietary commercial platform, nor requires the huge set-up times needed to start from scratch a solution. Furthermore, customizing Atmosphere based on new/changing requirements has proven to be easily feasible, also without compromising on abstraction for reusability. Code extensions were efficiently implemented when needed, only in few situations.

For developing a new cloud application, Atmosphere defines a workflow in which the first step consists of mapping the field objects to the API's resources, such as Features, Devices, Tags, Constraints, and User types. The Atmosphere instance is configured by POSTing these objects to the API. Then, the system becomes operational, allowing insertion/update of users, things, field measurements and computation requests, and retrieval of results. The system implements a rigorous RESTful architecture. Computation and machine learning resources can be exploited to allow further analysis of the acquired data.

On the edge, the engine development aimed at supporting IoT edge developers especially focusing on deployment speed with low-end hardware. The two experiments are indicators of the feasibility of deployment and configuration of the edge engine on heterogeneous IoT devices. The edge engine demonstrated our manifestation of the proposed concept: a generic IoT edge solution for smart cities. Compared to similar systems and solutions in the literature as well as in the tech market, the proposed engine is distinguished by its versatility, abstraction, and customizability.

The tests indicated that Atmosphere offers promising support also for starting up new generation e-mobility data-driven services for metering and energy-awareness.

Future works will involve enriching the existing set of computations also with machine learning processing, for instance for time-series prediction and automated clustering. Moreover, we would like to integrate psychometric measurements and user survey/questionnaire data, for supporting user acceptance studies in the field.

Committing to REST has shown some limitations that popped in while scaling up to the larger number of devices and a greater number of translations per second. An alternative to REST over HTTP based connectivity is Message Queuing Telemetry Transport (MQTT). MQTT [73], the lightweight protocol designed exclusively for IoT has its advantage in the much better data transfer rate. The major function of this feature packed protocol is that it caters enhancement for scalability & large-scale industrial deployments. Thus, we recommend bridging MQTT and REST in future work on this development.

APPENDIX A

API resources description

Thing:

- Route:

  https://api.atmosphere.tools/v1/things

- Methods:
  - GET/things: returns a list of the available things
  - POST/things: creates one or several things
  - DELETE/thing: removes a thing (if it is not a subject of existing measurements)
  - PUT/things: updates one thing record
  - GET/things/{id}: returns a single thing record
- Model:

```
{
  "_id": "string",
  "metadata": "string",
  "tags": [string]      // tags Ids
}
```

Device:

- Route:

  https://api.atmosphere.tools/v1/devices

- Methods:
  - GET/devices: returns a list of the available devices
  - POST/devices: creates one or several devices
  - DELETE/devices: removes a device (if it is not a subject of existing measurements)
  - GET/devices {id}: returns a single device record
- Model:

```
[
  {
    "_id": "string",
    "features": [string],      // features Ids
    "owner": "[objectId]",     // user Id
    "tags": [string]           // tags Ids
  }
]
```

Feature:

- Route:

  https://api.atmosphere.tools/v1/features

- Methods:
  - GET/features: returns a list of the available features
  - POST/features: creates one or several features
  - DELETE/features: removes a feature (if it is not a subject of existing measurements)
  - GET/features /{id}: returns a single feature record
- Model:

```
[
  {
    "_id": "string",
    "items": [
      {
        "name": "string",
        "unit": "string",
        "dimension": "number"
      }
    ],
    "owner": "[objectId]",          // user Id
    "description": "string",
    "tags": [string]                // tags Ids
  }
]
```

Tag:

- Route:

  https://api.atmosphere.tools/v1/tags

- Methods:
  - GET/tags: returns a list of available tags
  - POST/tags: creates one or several tags
  - DELETE/tags: removes one or more tags (if they are not a subject of existing measurements)
  - GET/tags /{id}: returns a single tag record
- Model:

```
[
  {
    "_id": "string",
    "description": "string",
    "tags": [string],         // tags Ids
    "owner": "objectId"       // user Id
  }
]
```

User:

- Route:

  https://api.atmosphere.tools/v1/users

- Methods:
  - GET/users: returns a list of registered users
  - POST/users: creates one or several users
  - DELETE/users: removes one or more users (if they are not a subject of existing measurements)
  - GET/users /{id}: returns a single user record
- Model:

```
[
  {
    "username": "string",
    "password": "string",
    "type": "string"          // Admin/ Provider/ Analyst
  }
]
```

Constraint:

- Route:

  https://api.atmosphere.tools/v1/constraints

- Methods:
  - GET/constraints: returns a list of available constraints
  - POST/constraints: creates one or several constraints
  - DELETE/constraints: removes constraints (according to a filter)
  - GET/constraints /{id}: returns a single constraint record
- Model:

```
[
  {
    "type1": "string",        // resource type
    "type2": "string",
    "element1": "string",     // resource Id
    "element2": "string",
    "relationship": "string"
  }
]
```

Script:

- Route:

  https://api.atmosphere.tools/v1/scripts

- Methods:
  - GET/scripts: returns a list of available scripts
  - POST/scripts: creates one or several scripts
  - DELETE/scripts: removes scripts (according to a filter)
  - GET/scripts /{id}: returns a single script record
- Model:

```
[
  {
    "id": "string",
    "tags": [string],               // tags Ids
    "device": "string",             // device Id
    "command": {
      "method": "string",           // http verb type
      "resource": [string],         // resource type
      "thing": "string",            // thing Id
      "feature": "string",          // feature Id
      "readInterval": "number",     // count of seconds
      "uploadInterval": " number",
      "loginInterval": " number",
      "operation": [object]
    }
  }
]
```

Measurement:

- Route:

  https://api.atmosphere.tools/v1/measurements

- Methods:
  - GET/measurements: returns a paginated list of filtered/aggregated measurements
  - POST/measurements: creates one or several measurements
  - DELETE/measurements: removes measurements (according to a filter)
  - PUT/measurements: updates one measurement record
  - GET/measurements/{id}: returns a single measurement record
- Model:

```
{
  "owner": "objectId",         // user Id
  "location": "geoJSON",
  "startDate": "date",
  "endDate": " date",
  "thing": "string",           // thing Id
  "device": "string",          // device Id
  "feature": "string",         // feature Id
  "samples": [
    {
      "values": [[object]],
      "delta": "number"
    }
  ],
  "tags": [string]             // tags Ids
}
```

Computation:

- Route:

  https://api.atmosphere.tools/v1/computations

- Methods:
  - GET/computations: returns a list of available computations
  - POST/computations: creates one or several computations
  - DELETE/computations: removes computations (according to a filter)
  - PUT/computations: updates one computation record
  - GET/computations/{id}: returns a single computation record
- Model:

```
[
  {
    "_id": "string",
    "name": "string",
    "owner": "[objectId]",    // user Id
    "code": "string",
    "filter": "string",
    "status": "string",
    "progress": number,
    "startDate": "date",
    "endDate": "date",
    "tags": [string]          // tags Ids
  }
]
```

Machine Learning:

- Route:

  https://api.atmosphere.tools/v1/machineLearning

- Methods:
  - GET/machineLearning: returns a list of trained models
  - POST/machineLearning: trains and creates one or several models
  - DELETE/machineLearning: removes models (according to a filter)
  - GET/machineLearning /{id}: returns a single model record
- Model:

```
{
    "mlAlgorithm": "string",        // e.g. Regression
    "params": {
      "iterations": "number",       // max. number of training iterations
      "error": "number"             // threshold for acceptable error
    }
}
```

APPENDIX B

TABLE B1
LOW-END HARDWARE RANGE OF SPECIFICATIONS

	Processor speed	Memory	Network interface	Logic level voltage
Waspmote	14 MHz	128 KB flash, 8 KB SRAM	Zigbee	3.3V
Arduino Uno	16 MHz	32 KB flash, 2 KB SRAM	I^2C, Ethernet Shield	5V
Zolertia Re-mote	32 MHz	512 KB flash, 32 KB SRAM	Wifi, Bluetooth	3.3V
Atmel SAM	48MHz	256 KB flash, 32 KB SRAM	Zigbee	3.6V
STM32 Nucleo	84 MHz	512 KB flash, 96 KB SRAM	I^2C, Ethernet Shield	3.3V
BeagleBone Black	1 GHz	512 GB RAM	Ethernet	3.3V
Raspberry Pi	1.2 GHz	1 GB RAM	Wifi, Ethernet	3.3V

APPENDIX C

TABLE C1
L3PILOT FEATURES STRUCTURE IN ATMOSPHERE

Feature	**Number of core Items**	**Additional metadata Items**	**Represents**
ScenarioInstance	22	6	Performance Indicator
Trip _ScenarioSpecific_	2	3	Performance Indicator
Trip	14	3	Performance Indicator
Following_a_lead_vehicle	11	6	Datapoint
Approaching_a_static_object	16	6	Datapoint
Approaching_a_traffic_jam	27	6	Datapoint
Approaching_a_lead_vehicle	27	6	Datapoint
Cut_in	11	6	Datapoint
Driving_in_traffic_jam	7	6	Datapoint
Lane_change	15	6	Datapoint
Uninfluenced_driving	2	6	Datapoint
Incident_with_rear_vehicle	13	6	Datapoint

TABLE C2
L3PILOT TAGS IN ATMOSPHERE

Tag	**Used In**	**Represents**
Measurement	Tags	Measurement
Thing	Tags	Thing
Condition	Tags	Trip condition
Scenario	Tags	Trip scenario
Road_Type	Tags	Trip road type
UI	Tags	User interface
Treatment	Measurements	Condition
Treatment_SysAvailable	Measurements	Condition
Treatment_ADF	Measurements	Condition
Treatment_SysNotAvailable	Measurements	Condition
Treatment_SysAvailable_ADF_OFF	Measurements	Condition
Baseline	Measurements	Condition
Lane_change	Measurements	Scenario
Following_a_lead_vehicle	Measurements	Scenario
Approaching_a_static_object	Measurements	Scenario
Approaching_a_lead_vehicle	Measurements	Scenario
Approaching_a_traffic_jam	Measurements	Scenario
Cut_in	Measurements	Scenario
Driving_in_traffic_jam	Measurements	Scenario
Uninfluenced_driving	Measurements	Scenario
Incident_with_rear_vehicle	Measurements	Scenario
Motorway	Measurements	Road type
All	Measurements	Road type
Urban1	Measurements	Road type
Urban2	Measurements	Road type
Urban3	Measurements	Road type
Parking	Measurements	Road type
Driver	Things	Driver
Trip	Things	Trip
Datapoint	UI	Feature
ScenarioInstance_PI	UI	Feature
Trip_ScenarioSpecific_PI	UI	Feature
Trip_PI	UI	Feature

LIST OF REFERENCES

[1] L. Jiang, L. D. Xu, H. Cai, Z. Jiang, F. Bu and B. Xu, “An IoT-Oriented Data Storage Framework in Cloud Computing Platform,” in IEEE Transactions on Industrial Informatics, vol. 10, no. 2, pp. 1443-1451, May 2014.

[2] D. Mourtzis, E. Vlachou, N. Milas, “Industrial Big Data as a Result of IoT Adoption in Manufacturing,” Procedia CIRP, vol. 55, 2016, pp. 290-295.

[3] M. Soliman, T. Abiodun, T. Hamouda, J. Zhou and C. Lung, “Smart Home: Integrating Internet of Things with Web Services and Cloud Computing,” 2013 IEEE 5th International Conference on Cloud Computing Technology and Science, Bristol, 2013, pp. 317-320.

[4] G. Suciu, V. Suciu, A. Martian, R. Craciunescu, A. Vulpe, I. Marco, S. Halunga, O. Fratu, “Big Data, Internet of things and Clod Convergence – An Architecture for secure E-Health Applications,” Journal of Medical Systems, vol. 39, no. 11, 2015.

[5] J. H. Kim, “A Review of Cyber-Physical System Research Relevant to the Emerging IT Trends: Industry 4.0, IoT, Big Data, and Cloud Computing,” Journal of Industrial Integration and Management, vol. 02, no. 03, 2017.

[6] S. Shekhar and A. Gokhale, “Poster Abstract: Enabling IoT Applications via Dynamic Cloud-Edge Resource Management,” 2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI), Pittsburgh, PA, 2017, pp. 331-332.

[7] S. K. Sharma and X. Wang, “Live Data Analytics With Collaborative Edge and Cloud Processing in Wireless IoT Networks,” in IEEE Access, vol. 5, pp. 4621-4635, 2017.

[8] A. Kertesz, “Interoperating Cloud Services for Enhanced Data Management,” 2014 IEEE Fourth International Conference on Big Data and Cloud Computing, Sydney, NSW, 2014, pp. 269-270.

[9] L. Mossucca, V. Romano, O. Terzo, P. Ruiu, L. Spogli, A. Salvati, C. Rafanelli, “Polar Data Management Based on Cloud Technology,” 2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems, Blumenau, 2015, pp. 459-463.

[10] R. Petrasch and R. Hentschke, “Cloud storage hub: Data management for IoT and industry 4.0 applications: Towards a consistent enterprise information management system,” 2016 Management and Innovation Technology International Conference (MITicon), Bang-San, 2016, pp. MIT-108-MIT-111.

[11] Amazon Web Services AWS IoT, 2019, https://aws.amazon.com/iot/solutions/industrial-iot/?nc=sn&loc=3&dn=2

[12] W. Tarneberg, V. Chandrasekaran, and M. Humphrey, “Experiences creating a framework for smart traffic control using AWS IOT,” In Proceedings of the 9th International Conference on Utility and Cloud Computing (UCC '16). ACM, New York, NY, USA, 63-69, 2016.

[13] Azure IoT, Microsoft, 2019, https://azure.microsoft.com/en-us/overview/iot/

[14] S. Jiong, J. Liping, and L. Jun, “The Integration of Azure Sphere and Azure Cloud Services for Internet of Things,” MDPI journal on Applied Sciences, vol. 9, no. 13: 2746, 2019.

[15] T. Pflanzner and A. Kertesz, “A survey of IoT cloud providers,” 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), Opatija, pp. 730-735, 2016.

[16] M. Zúñiga-Prieto, J. González-Huerta, E. Insfran, and S. Abrahão, “Dynamic reconfiguration of cloud application architectures,” Journal of Software: Practice and Experience, vol. 48, pp. 327– 344, 2016.

[17] H. Chi, T. Aderibigbe and B. C. Granville, “A Framework for IoT Data Acquisition and Forensics Analysis,” 2018 IEEE International Conference on Big Data (Big Data), Seattle, WA, USA, 2018, pp. 5142-5146.

[18] J. Jung, K. Kim and J. Park, “Framework of Big data Analysis about IoT-Home-device for supporting a decision making an effective strategy about new product design,” 2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC), Okinawa, Japan, 2019, pp. 582-584.

[19] Ş. Kolozali et al., “Observing the Pulse of a City: A Smart City Framework for Real-Time Discovery, Federation, and Aggregation of Data Streams,” in IEEE Internet of Things Journal, vol. 6, no. 2, pp. 2651-2668, April 2019.

[20] L. Jiang, L. D. Xu, H. Cai, Z. Jiang, F. Bu and B. Xu, “An IoT-Oriented Data Storage Framework in Cloud Computing Platform,” in IEEE Transactions on Industrial Informatics, vol. 10, no. 2, pp. 1443-1451, May 2014.

[21] H. Cai, B. Xu, L. Jiang and A. V. Vasilakos, “IoT-Based Big Data Storage Systems in Cloud Computing: Perspectives and Challenges,” in IEEE Internet of Things Journal, vol. 4, no. 1, pp. 75-87, Feb. 2017.

[22] J. Fu, Y. Liu, H. Chao, B. K. Bhargava and Z. Zhang, “Secure Data Storage and Searching for Industrial IoT by Integrating Fog Computing and Cloud Computing,” in IEEE Transactions on Industrial Informatics, vol. 14, no. 10, pp. 4519-4528, Oct. 2018.

[23] S. S. Solapure and H. Kenchannavar, “Internet of Things: A survey related to various recent architectures and platforms available,” 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Jaipur, 2016, pp. 2296-2301.

[24] M. Weinberger, M. Köhler, D. Wörner, and F. Wortmann, “Platforms for the Internet of Things: An Analysis of Existing Solutions,” 5th Bosch Conference on Systems and Software Engineering (BoCSE) – Ludwigsburg, 2015.

[25] S. K. Sharma and X. Wang, “Live Data Analytics With Collaborative Edge and Cloud Processing in Wireless IoT Networks,” in IEEE Access, vol. 5, pp. 4621-4635, 2017.

[26] T. Yu, X. Wang and A. Shami, “Recursive Principal Component Analysis-Based Data Outlier Detection and Sensor Data Aggregation in IoT Systems,” in IEEE Internet of Things Journal, vol. 4, no. 6, pp. 2207-2216, Dec. 2017.

[27] F. Paganelli, S. Turchi, and D. Giuli, “A Web of Things Framework for RESTful Applications and Its Experimentation in a Smart City,” in IEEE Systems Journal, vol. 10, no. 4, pp. 1412-1423, Dec. 2016.

[28] Amazon Web Service. 2019. AWS Lambda@Edge. https://aws.amazon.com/lambda/edge/.

[29] Villamizar, M.; et al.; Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures, 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Cartagena, 2016, pp. 179-182.

[30] William Tärneberg, Vishal Chandrasekaran, and Marty Humphrey. 2016. Experiences creating a framework for smart traffic control using AWS IOT. In Proceedings of the 9th International Conference on Utility and Cloud Computing (UCC '16). ACM, New York, NY, USA, 63-69.

[31] Azure IoT Edge. 2019. Microsoft. https://azure.microsoft.com/en-in/services/iot-edge/.

[32] Forsström, S.; Jennehag, U.; A performance and cost evaluation of combining OPC-UA and Microsoft Azure IoT Hub into an industrial Internet-of-Things system, 2017 Global Internet of Things Summit (GIoTS), Geneva, 2017, pp. 1-6.

[33] Familiar, B.; Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions, Apress, 2015.

[34] Jain, R.; Tata, S.; Cloud to Edge: Distributed Deployment of Process-Aware IoT Applications, 2017 IEEE International Conference on Edge Computing (EDGE), Honolulu, HI, 2017, pp. 182-189.

[35] Fan, K.; Pan, Q.; Wang, J.; Liu, T.; Li, H.; Yang, Y.; Cross-Domain Based Data Sharing Scheme in Cooperative Edge Computing, 2018 IEEE International Conference on Edge Computing (EDGE), San Francisco, CA, 2018, pp. 87-92.

[36] Cirani, S.; Ferrari, G.; Iotti, N.; Picone, M.; The IoT hub: a fog node for seamless management of heterogeneous connected smart objects, 2015 12th Annual IEEE International Conference on Sensing, Communication, and Networking - Workshops (SECON Workshops), Seattle, WA, 2015, pp. 1-6.

[37] Noghabi, S. A.; Kolb, J.; Bodik, P.; Cuervo, E.; Steel: Simplified Development and Deployment of Edge-Cloud Applications, 10th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 18).

[38] Yuan, J.; Li, X.; A Reliable and Lightweight Trust Computing Mechanism for IoT Edge Devices Based on Multi-Source Feedback Information Fusion, in IEEE Access, vol. 6, pp. 23626-23638, 2018.

[39] Xu, X.; Huang, S.; Feagan, L.; Chen, Y.; Qiu Y.; Wang, Y.; EAaaS: Edge Analytics as a Service, 2017 IEEE International Conference on Web Services (ICWS), Honolulu, HI, 2017, pp. 349-356.

[40] Morabito, R.; Cozzolino, V.; Ding, A. Y.; Beijar, N.; Ott, J.; Consolidate IoT Edge Computing with Lightweight Virtualization, in IEEE Network, vol. 32, no. 1, pp. 102-111, Jan.-Feb. 2018.

[41] P. K. Potti, “On the Design of Web Services: SOAP vs. REST,” UNF Graduate Theses and Dissertations, University of North Florida, 2011.

[42] F. Bellotti et al., “TEAM Applications for Collaborative Road Mobility,” in IEEE Transactions on Industrial Informatics, vol. 15, no. 2, pp. 1105-1119, Feb. 2019.

[43] Node.js User Survey Report, 2018. https://nodejs.org/en/user-survey-report/.

[44] MongoDB, 2019: https://www.mongodb.com/what-is-mongodb

[45] J. Guo, L. Xu, G. Xiao, Z. Gong, "Improving multilingual semantic interoperation in cross-organizational enterprise systems through concept disambiguation", IEEE Trans. Ind. Informat., vol. 8, no. 3, pp. 647-658, Aug. 2012.

[46] R. Arora, R. R. Aggarwal, "Modeling and Querying Data in MongoDB," International Journal of Scientific & Engineering Research, Volume 4, Issue 7, July-2013.

[47] Internet Engineering Task Force (IETF), "The JavaScript Object Notation (JSON) Data Interchange Format," December 2017.

[48] JSON Web Token, IETF, 2015: https://tools.ietf.org/html/rfc7519.

[49] D. Smilkov, N. Thorat, Y. Assogba, A. Yuan, N. Kreeger, P. Yu, K. Zhang, S. Cai, E. Nielsen, D. Soergel, S. Bileschi, M. Terry, C. Nicholson, S. N. Gupta, S. Sirajuddin, D. Sculley, R. Monga, G. Corrado, F. B. Viégas, and M. Wattenberg, "TensorFlow.js: Machine Learning for the Web and Beyond," 2019.

[50] T. D. V. Swinscow, "Statistics at Square One," Ninth edition, BMJ Publishing Group.

[51] D. Herron, "Node Web Development," Second Edition, Packt Publishing.

[52] M. Haverbeke, "Eloquent JavaScript: A Modern Introduction to Programming," Third Edition, No Starch Press.

[53] K. Faaborg and S. Pasquali, "Mastering Node.js," Second Edition, Packt Publishing.

[54] Swagger, 2019: https://swagger.io/tools/open-source/getting-started/.

[55] C. Pallasch et al., "Edge Powered Industrial Control: Concept for Combining Cloud and Automation Technologies," 2018 IEEE International Conference on Edge Computing (EDGE), San Francisco, CA, 2018, pp. 130-134.

[56] Y. Song, S. S. Yau, R. Yu, X. Zhang and G. Xue, “An Approach to QoS-based Task Distribution in Edge Computing Networks for IoT Applications,” 2017 IEEE International Conference on Edge Computing (EDGE), Honolulu, HI, 2017, pp. 32-39.

[57] W. Shi and S. Dustdar, “The Promise of Edge Computing,” in Computer, vol. 49, no. 5, pp. 78-81, May 2016.

[58] A. Monteriù, M.R. Prist, E. Frontoni, S. Longhi, F. Pietroni, S. Casacci, L. Scalise, A. Cenci, L. Romeo, R. Berta, L. Pescosolido, G. Orlandi, G.M. Revel, “A smart sensing architecture for domestic moniotring: methodological approach and experimental validation”, Sensors, Vol. 18 Issue 7, July 2018.

[59] L. Pescosolido, R. Berta, L. Scalise, G. M. Revel, A. De Gloria and G. Orlandi, “An IoT-inspired cloud-based web service architecture for e-Health applications,” 2016 IEEE International Smart Cities Conference (ISC2), Trento, 2016, pp. 1-4.

[60] V. Cirimele, M. Diana, F. Bellotti, R. Berta, A. Kobeissi, N. El Sayyed, J. Colussi, A. La Ganga, P. Guglielmi, and A. De Gloria, “The Fabric ICT platform for managing Wireless Dynamic Charging Road lanes,” in press of IEEE Transactions on Vehicular Technology (IEEE TVT), 2019.

[61] Feasibility analysis and development of on-road charging solutions for future electric vehicles, Fabric, 2019, https://www.fabric-project.eu/.

[62] A. Kobeissi, F. Bellotti, R. Berta, and A. De Gloria, “Towards an IoT-enabled Dynamic Wireless Charging Metering Service for Electrical Vehicles,” 4th AEIT International Conference of Electric and Electronic Technologies for Automotive, Turin, Italy, 2019.

[63] A. H. Kobeissi, F. Bellotti, R. Berta, A. De Gloria, “Raspberry Pi 3 Performance Characterization in an Artificial Vision Automotive Application”, Applications in Electronics Pervading Industry, Environment and Society (ApplePies 2018), Pisa, Italy, September 26-27, 2018.

[64] L3Pilot Driving Automation, 2019, https://l3pilot.eu/.

[65] "Definitions for Terms Related to On-Road Motor Vehicle Automated Driving Systems," (J3016), Technical Report, SAE International (2016).

[66] "Updated version of the FESTA Handbook," https://connectedautomateddriving.eu/wp-content/uploads/2017/08/2017-01-31_FOT-NET_FESTA-Handbook.pdf, accessed 8 January 2019.

[67] J. Hiller, E. Svanberg, S. Koskinen, F. Bellotti, F, Osman, N, "The L3Pilot Common Data Format – Enabling Efficient Autonomous Driving Data Analysis", in Proceedings of NHTSA ESV 2019, Eindhoven.

[68] Y. Bernard, S. Innamaa, S. Koskinen, H. Gellerman, E. Svanberg, and H. Chen, "Methodology for Field Operational Tests of Automated Vehicles," Transport Research Procedia, vol. 14, 2016, pp. 2188-2196.

[69] B. Nagy, J. Hiller, N. Osman, S. Koskinen, E. Svanberg, F. Bellotti, R. Berta, A. Kobeissi, A. De Gloria, "Building a Data Management Toolchain for a Level 3 Vehicle Automation Pilot," in press 26th ITS World Congress, Singapore, 2019.

[70] F. Bellotti, R. Berta, A. Kobeissi, N. Osman, E. Arnold, M. Dianati, B. Nagy, A. De Gloria, "Designing an IoT Framework for Automated Driving Impact Analysis," 30th IEEE Intelligent Vehicle Symposium, Paris, June 2019.

[71] Raspberry Pi 3 model b. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[72] A. H. Kobeissi, F. Bellotti, R. Berta, A. De Gloria, "IoT Ubiquitous Edge Engine Implementation on the Raspberry PI", Applications in Electronics Pervading Industry, Environment and Society (ApplePies 2019), Pisa, Italy, September 11-13, 2019.

[73] U. Hunkeler, H. L. Truong, A. Stanford-Clark, "MQTT-S — A publish/subscribe protocol for Wireless Sensor Networks," 2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE '08), Bangalore, 2008, pp. 791-798.