Towards a Seamless Integration of IoT Devices with IoT Platforms Using a Low-Code Approach

Silviu-George Pantelimon*, Tudor Rogojanu*, Andreea Braileanu*,
Valeriu-Daniel Stanciu*, Ciprian Dobre*†

*University Politehnica of Bucharest, Bucharest, Romania
E-mails: {silviu.pantelimon, tudor.rogojanu}@cti.pub.ro, andreea.braileanu@stud.acs.upb.ro, valeriu.stanciu@cti.pub.ro
†SmartRDI - Smart Research, Development and Innovation, Bucharest, Romania
E-mail: ciprian.dobre@smartrdi.net

***Abstract*—To enable the widespread adoption of Internet of Things, innovative mechanisms for seamlessly building solutions, from hardware to intelligent applications, must be found. The processes should become less complex and the integration should be done without effort. In this paper we introduce a novel end-to-end low-code mechanism for managing the relationship between heterogeneous hardware sensors and IoT platforms. Moreover, we provide a comparative analysis of different technologies to be used when implementing such a mechanism.**

***Index Terms*—NETIoT; hpaPaas; decoding mechanism; end-to-end integration.**

I. Introduction

Internet of Things (IoT) has recently become a key element in building the society of the future. The increasing evolution of communication techniques in the low-power wide-area network (LPWAN) space together with the advances on the path of 5G communications offered a powerful traction towards accelerating the widespread adoption of IoT. Numerous initiatives emerged, a plethora of platforms came to the market, covering different aspects of the solution building process.

However, due to the fact that deploying a solution involves dealing with disparate technologies at different levels, the technical complexity can often become overwhelming, requiring skills from a broad pallet. The need for a technical mix ranging from advanced programming skills and hardware knowledge, to specific expertise in the solution's field, along with a widely-acknowledged IT talent shortage ([1], [2], [3]), could heavily affect the impact that deploying an IoT solution might have. Meanwhile, a new movement of low-code / no-code application development arose, aiming to bridge the gap between the technology and the field experts, allowing the line-of-business professionals to build their own applications in the cloud without the hassle of writing code themselves or relying on programmers to do this task. Gartner defined this kind of approaches under the term "high-productivity application platform as a service" (hpaPaaS) [4], releasing yearly Magic Quadrants on this topic since.

In a previous article we introduced NETIoT [5], a versatile IoT hpaPaaS platform covering all the aspects of IoT solutions development, from sensors towards the platform and up to the applications, in a scalable, extensible, highly available and fault tolerant way. A user should be able to add his devices into the platform at a snap of a finger, keeping the complexity of configurations at a minimum and the necessary skills of doing that low. Moving from devices providing raw measurements towards intelligent applications responding to relevant business needs should be a handy process too. Achieving these goals is of paramount importance, representing the core principles of any IoT hpaPaaS platform.

In this paper we will focus on how to efficiently bring measurements from heterogeneous IoT hardware devices to the NETIoT platform in a low-code manner, with zero intervention on hardware and minimum configuration effort platform-wise. We will introduce a complete mechanism, oblivious to the devices being used, which releases the user from the burden of programming his hardware or buying overpriced ready-configured devices to build his solution, as opposed to common vendor lock-in strategies. Design considerations will be presented, together with an innovative integration of technologies for devices management and decoding procedures; an extensive analysis and comparison between alternative options will be provided, thus strongly motivating our decisions, and a thorough evaluation of a proof-of-concept implementation will be performed.

The remainder of this paper proceeds as follows. In Section II we briefly describe the NETIoT platform. Section III is dedicated to presenting our work's design considerations and end-to-end mechanism, followed, in Section IV, by a number of experiments and evaluation procedures. In Section V we present related work in this field. Finally, Section VI concludes the paper.

II. NETIoT

NETIoT is an IoT platform built following the hpaPaaS principles, centered around the applications being supported by a set of IoT devices owned by a user. It covers all the spectrum of IoT solutions development, from devices management to intelligent applications. A goal of utmost importance is to enable the user to benefit from his hardware devices by being able to use, on the fly, applications compatible with his range of sensors. NETIoT's architecture is depicted in Fig. 1.

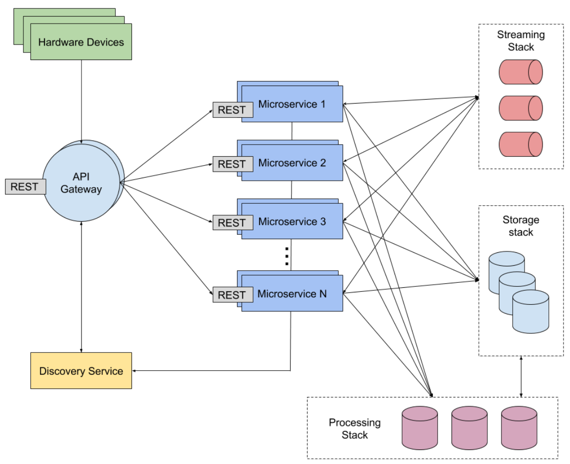


Fig. 1. NETIoT architecture, as illustrated in [5, Figure 1].

From a technical perspective, the platform was implemented using a microservices-based architecture. It is meant to ensure high availability, scalability, fault tolerance and further development.

III. Decoding Mechanism

The main motivation pushing us forward in this article is that we need an end-to-end mechanism for understanding the data sent by IoT devices. Building a platform capable of understanding this data regardless of the device manufacturer is a key element for the democratization of IoT. Each manufacturer has different ways of encoding the data, this representing one of the biggest challenges. For example, in several LPWAN technologies, such as LoRaWAN, the data to be sent is limited to 242 bytes [6], so the length of the payload should be kept as small as possible to ensure the arrival of messages and to diminish the energy consumption; thus, the sensors manufacturers have to implement different encoding methods, tailored to specific devices. We argue that the NETIoT platform should be able to deal with data sent by a variety of devices running on different technologies, encoded in diverse ways, by stripping this heterogeneity layer from the data and transforming it into a common internal representation.

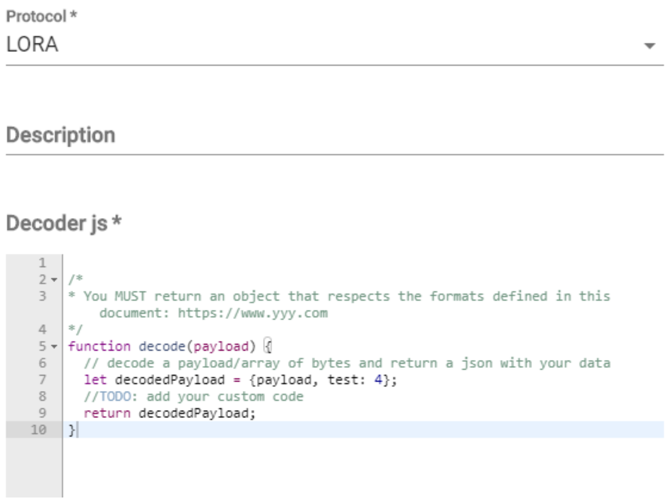
While accommodating a limited number of manufacturers and devices is at hand, achievable by just hardcoding a couple of decoding mechanisms in the backend, it is not desirable since this will become a heavy limitation of the platform. At the same time, forcing the users to deal with programming their devices to send data in a particular format is also out of discussion since it requires more skills and it introduces additional time in the process. We desire to accommodate the decoding of data sent by devices in a way which is agnostic to devices' type or manufacturer and, at the same time, in a low-code manner.

To fulfill all the demands presented above, we decided to go for a mixed approach, combining already defined decoders covering a set of well-known IoT device manufacturers and device types with the possibility to enrich the decoders' database with custom defined decoders for new types of devices. In other words, besides being offered a broad range of devices to choose from, the platform user is also given the opportunity to use any other device he desires, as long as he provides a decoder for it.

From a technical point of view, we need an end-to-end decoding mechanism, easily configurable on the frontend and efficiently executed on the backend. We opted for JavaScript as the vehicle for describing the decoders and linking the frontend with the backend. The choice fell on JavaScript for several reasons, including aspects such as:

1) Ease of coding (it requires little programming knowledge and skills), which makes it the best option for low-code implementations;
2) Flexibility given by the interpreted nature of the language, decoders being loaded and directly executed in the backend while new decoders can be introduced at any time;
3) Good documentation available for the JavaScript interpreters and JavaScript itself.

In the following paragraphs we will describe the whole process enabling a device to be understood by the platform: defining devices and decoders on user side, messages ingestion and decoding on the backend.



Protocol *
LORA

Description

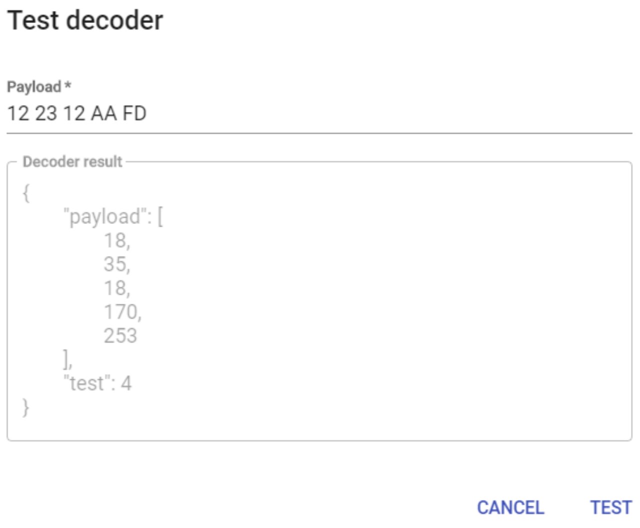
Decoder js *

```
/*
* You MUST return an object that respects the formats defined in this
    document: https://www.yyy.com
*/
function decode(payload) {
  // decode a payload/array of bytes and return a json with your data
  let decodedPayload = {payload, test: 4};
  //TODO: add your custom code
  return decodedPayload;
}
```

Fig. 2. Defining a decoder.

A. User Side

Adding and modifying devices and decoders should be as simple as possible, so that the average platform user can do it with limited effort. As we have previously mentioned, besides having the already existing decoders for common devices, the user should be offered the possibility to add new decoders. In order to create his own decoder (Fig. 2), the user must specify the protocol used by the devices that will benefit from this decoder and fill in a minimal JavaScript decoding function returning a valid JSON adhering to the types of data supported by the platform (these are provided to the user when writing the piece of JavaScript code). Once the decoder is saved, the user can test it (Fig. 3) by sending some bytes as payload and analyzing the resulting JSON containing the decoded message. This payload is processed directly in the browser, using the decode function of the selected decoder. Validation is also involved in the process, such that function's name and parameters are exactly as defined in the JavaScript template, the function has a return statement and there are no warnings or errors in the JavaScript code.


Test decoder

Payload *
12 23 12 AA FD

Decoder result
```
{
    "payload": [
        18,
        35,
        18,
        170,
        253
    ],
    "test": 4
}
```

Fig. 3. Testing a decoder.

When setting up a device, among other things, the user must choose a decoder to use for that specific device. Once the device is saved, the user can test the end-to-end integration: simulate sending some bytes as payload from the device, check whether they generate any error and if the message reaches the server. This payload is not processed in the browser any more. It will pass through the whole chain of processing, being handled by a decoding microservice in the backend; the decoded data is validated in the backend too, alerts are generated if there are any JavaScript errors and the resulting JSON is enforced to adhere to the types of data supported by the platform. In the end, the user will see if there is any error and, if not, he will be able to find the decoded message in the table of devices. At this point, the user has a complete setup. The device is properly configured, it can send data and the data is correctly decoded, saved and presented to the user on the frontend.

In case the user wishes to do changes to his hardware, e.g. adding or removing some sensors from the device, he will simply have to update the decoder and parse the new device data accordingly. If the new sensors send data that is not yet supported by the platform, it can be easily accommodated by request due to the flexible and extensible nature of the NETIoT platform, an administrator just having to add a new JSON model for the new type of sensor data.

B. Backend Decoding Architecture

To provide high-availability and scalability, our platform's architecture is based on microservices. Each microservice runs in a Docker container, orchestrated by Kubernetes, an open-source container-orchestration system. Thus, the decoding process will take place in a microservice; the number of instances is dictated by the load. The technology used for building the microservice could be of multiple kinds, depending on multiple aspects explained in more detail in the evaluation section.

The decoding architecture is presented in Figure 4. This construction is plugged into the NETIoT platform. The device data enters our platform through a cloud gateway over a REST API, then it is distributed to one or more Decoder microservices. The decoding microservice interrogates a Redis key-value database containing all the JavaScript decoders and downloads the specific decoder needed for the particular device which generated the data. We decided to use Redis for storing the decoders due to its in-memory nature, high speed of retrieval and scalability. The microservice then should interpret the JavaScript string and execute it with the payload as a parameter. The result is sent, asynchronously, to an Apache Kafka queue. The Kafka queue distributes the result to multiple consumers, in our case an IO-Server process which stores the date in the Cassandra database and an Apache Spark process whose purpose is to execute further processing with the decoded data.

IV. Performance Evaluation

We wanted to analyze the performance reached by the decoding microservice, implemented with different technologies. The tests were performed on an Intel Quad-core i7-6700 3.40GHz CPU under Ubuntu 16.04 LTS. We tested firstly the proficiency of different JavaScript libraries and secondly, if provided a large amount of workload, how the different implementations and components are behaving under similar settings. In our case, the key performance indicator is the amount of processed messages per second.

Our proof-of-concept microservice was implemented in three different programming languages, C++, Java and JavaScript, using four technologies: Google's V8 written in C++[1], Java Rhino[2], Java Nashorn[3], and Node.js 10[4].

We first looked at Java engines for JavaScript, since most of the NETIoT platform is based on Java Spring, being at hand to continue with the same practice. Initially we ran a few tests with Mozilla's Rhino library, but, compared to the same implementation in C++ and Node.js, it performed very poorly. We also investigated Oracle's Nashorn library integrated into JDK 8, although it had been announced to be removed in the future and it is currently deprecated in Java 11. An alternative to Java Spring was Node.js. Being a run-time environment for JavaScript made it a candidate technology for us. Still, we also considered developing under C++, because Node.js wraps over the V8 JavaScript engine and connects to libraries written in C/C++ for various features. We found it interesting enough to

[1]https://v8.dev/
[2]https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino
[3]http://openjdk.java.net/projects/nashorn/
[4]https://nodejs.org/en/

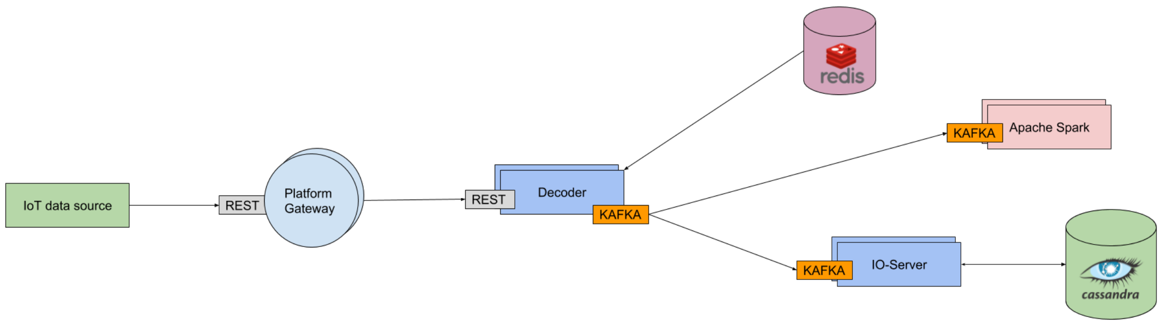


Fig. 4. Backend decoding architecture.

determine how much the performance differs as Node.js added an additional layer of abstraction over the V8 engine. Similarly to Java, the V8 engine creates a virtual machine for memory management within which the JavaScript code is executed.

For our tests we took an example decoder for a LoRaWAN sensor existing on the market [7]. In the case of LoRaWAN and similar LPWAN technologies like NB-IoT or Sigfox, the message payload is relatively small so the tests ran on a 16-byte long payload, a common length for these cases. On the other hand, LoRaWAN is designed to manage hundreds of thousands of devices, so we also wanted to see the behavior of the microservice under high computational stress. In the following graphs, on the x axis is the total number of messages sent to the microservice and on the y axis is the number of messages per second as a performance measure.

The first set of tests (Fig. 5) consisted of loading a JavaScript decoder multiple times in the different technologies and then executing the decoding function. In the second set of tests (Fig. 6), the JavaScript decoding function was retrieved from the Redis dictionary, executed, and then the result sent towards Kafka. The clients used for Redis where acl-redis[5] for C++, the redis package[6] for Node.js and Jedis[7] for Java. The Kafka libraries in each case were the librdkafka[8], the kafka-node package[9], and the default Kafka library in Java[10]. The Kafka producers are used asynchronously in all implementations.

The first set of tests immediately excluded the Java Rhino library since preliminary tests showed limited performance, mainly because it translates the JavaScript code into Java objects. The other technologies had comparable execution times (see Figure 5) as the V8 engine and the Nashorn library work similarly; in V8, the JavaScript code is compiled into machine code and in Nashorn into JVM instructions. Still, Node.js was the fastest of all, but all the results were within the same order of magnitude.

[5]https://github.com/acl-dev
[6]https://www.npmjs.com/package/redis
[7]https://github.com/xetorthio/jedis
[8]https://github.com/edenhill/librdkafka
[9]https://www.npmjs.com/package/kafka-node
[10]https://kafka.apache.org/documentation/

In the second set of tests we observed that while the Redis query and the actual decoder execution didn't take that much time, all technologies spent the most time on sending messages to the Kafka queue. In all cases the Kafka producer was a separate thread. We determined that the main thread flooded the producer thread with messages to be queued because it was faster than the producer could send. Again all the results fall within the same order of magnitude (see Figure 6). Until a certain load threshold is reached, Node.js performs the best, then the C++ implementation outperforms all the other options, though not by a high margin.

Summing up this section, we state that even if the different technologies perform similarly, each of them has pros and cons. While C++ might obtain marginally better performance on the long run and it has good support for multithreading, it requires a high development effort and the most programming skills and understanding. Java means easy development and good performance, but, at the same time, Nashorn supports only ECMAScript 5 and it is already deprecated, on the verge of being removed from the JDK. Node.js is a viable option, being a JavaScript runtime, offering easy development and good performance, even if it currently doesn't offer multithreading support.

V. Related Work

In recent years, numerous IoT platforms appeared, each covering a particular aspect in the building process of IoT solutions. A recent survey [8] indicated no less than 128 different IoT platforms being available on the market. The approaches of IoT platforms with requirements such as communication over the Internet and scalability are vast. In some cases, bidirectional communication is assured with the purpose of collecting the data, but also controlling the environment.

With regards to this matter, Oracle Internet of Things (IoT) Cloud Service PaaS solution[11] manages to provide a secure bidirectional communication between their devices. When deployed, the devices can connect in various network topologies such as Client Library, Gateway Software and REST APIs. Client libraries are available for Java SE, Java ME, Android,

[11]https://docs.oracle.com/en/cloud/paas/iot-cloud/index.html

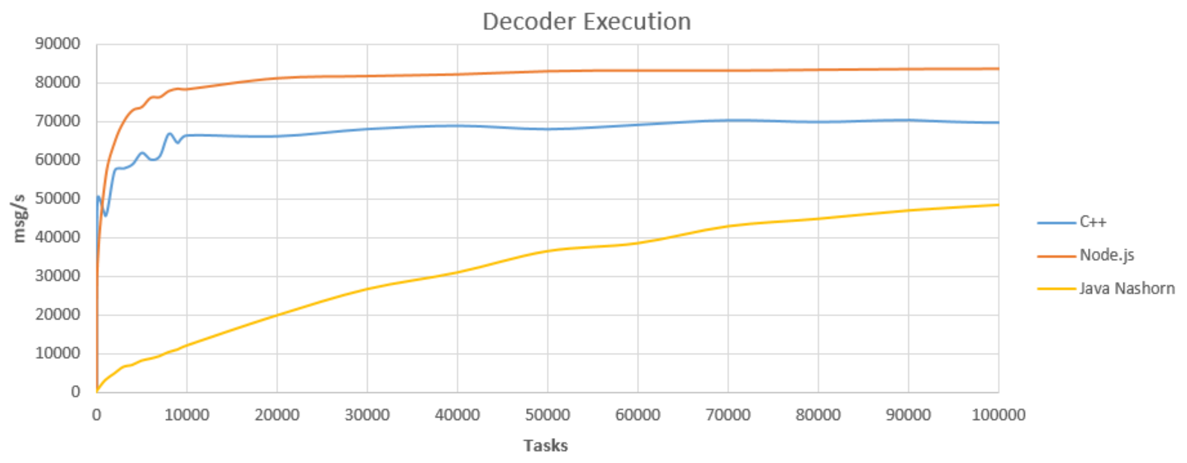


Fig. 5. First set of tests.

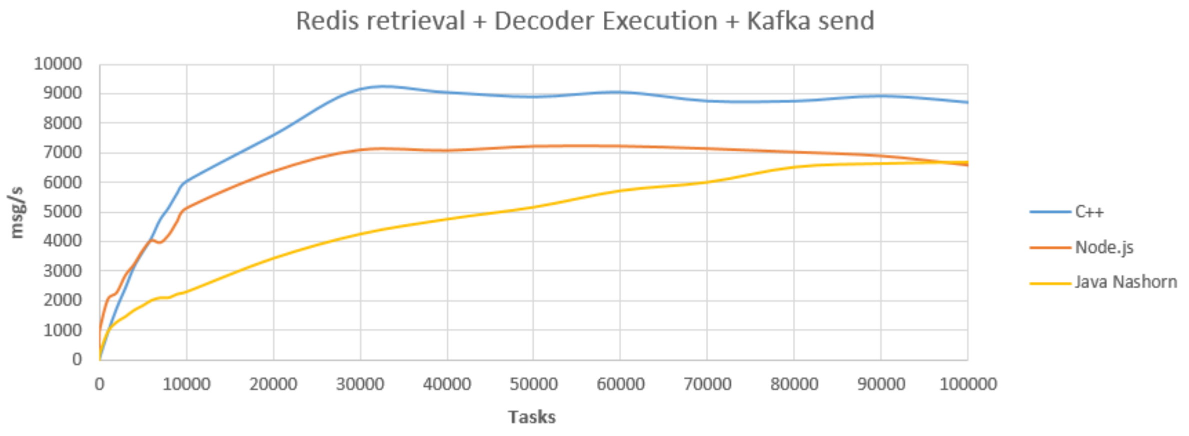


Fig. 6. Second set of tests.

and POSIX C, whereas the Gateway Software includes a framework that enables communication through Bluetooth, ZWave, Modbus, and OPC.

Other IoT platforms such as Ubidots keep a close eye on low-code methods, though not fully-fledged. The user can easily create a new device or even configure an existing one using Device Types that automates the onboarding process. The UbiParser[12] allows users to create their own API, by parsing the data into a Node.JS cloud GET or POST HTTP function. In case of a POST HTTP function, parameters such as a specified token, a device label and a payload in JSON format, as an input, are required.

Another interesting approach is given by The Things Network (TTN) platform[13]. They can manage both devices that support IP-based protocols and non-IP protocols such as LoRaWAN, providing a LoRaWAN Network Server. Notions such as Brokers and Handlers are introduced. Message encryption, decryption and conversion takes place on the Handler side, whereas on the Broker side the range of device addresses to be handled is set and a Handler is found to forward each message to. The Handler is able to decode a payload with the help of the JavaScript payload functions that are included. Even if this approach comes close to ours, TTN is focused more on the communication side, while we cover the end-to-end process, from the sensors up to the applications.

VI. Conclusion

In this paper we introduced a novel end-to-end low-code mechanism for connecting heterogeneous IoT hardware devices to the NETIoT platform and understanding the data generated by them. We provided a frontend solution, together with a backend decoding architecture. We have also analyzed proof-of-concept implementations of a decoding microservice in different technologies and we offered valuable insights on what it means to build such a system and what performance can be achieved under various loads. Future work includes investigating other aspects of the hpaPaaS platforms, such as applications abstraction and other low-code solutions.

[12]https://help.ubidots.com/developer-guides/use-ubifunction-to-ingest-sensor-data-using-http-get-requests

[13]https://www.thethingsnetwork.org/docs/network/architecture.html

Acknowledgment

The research presented in this paper is supported by the CRESCDI (Ctr. 25PFE/17.10.2018), TEL-MONAER (Ctr. 53/SMIS2014+/105976) and SPERO (PN-III-P2-2.1-SOL-2016-03-0046) projects. We would like to thank the reviewers for their time and expertise, constructive comments and valuable insight.

References

[1] D. D. Bowman, "Declining talent in computer related careers." *Journal of Academic Administration in Higher Education*, vol. 14, no. 1, 2018.

[2] W. Yang, L. Zhang, and L. Yu, "An analysis of the talents shortage in the present labor market: A case study of china," *DEStech Transactions on Economics, Business and Management*, no. icssed, 2018.

[3] K. Caspin-Wagner, S. Massini, and A. Y. Lewin, "The changing structure of talent for innovation: on demand online marketplaces," in *International Business in the Information and Digital Age*. Emerald Publishing Limited, 2018, pp. 245–272.

[4] Gartner. (2018) Magic quadrant for enterprise high-productivity application platform as a service. [Online]. Available: https://www.gartner.com/doc/3872957

[5] T. Rogojanu, M. Ghita, V. Stanciu, R.-I. Ciobanu, R.-C. Marin, F. Pop, and C. Dobre, "Netiot: A versatile iot platform integrating sensors and applications," in *2018 Global Internet of Things Summit (GIoTS)*. IEEE, 2018, pp. 1–6.

[6] N. Sornin, M. Luis, T. Eirich, T. Kramp, and O. Hersent, "Lora specification 1.0," *Lora Alliance Standard specification., Jan*, 2015.

[7] N. Blenn and F. Kuipers, "Lorawan in the wild: Measurements from the things network," *arXiv preprint arXiv:1706.03086*, 2017.

[8] S. Forsstrom, U. Jennehag, P. Österberg, V. Kardeby, and J. Lindqvist, "Surveying and identifying the communication platforms of the internet of things," in *2018 Global Internet of Things Summit (GIoTS)*. IEEE, 2018, pp. 1–6.