Doctoral Dissertation
Doctoral Program in Computer and Control Engineering (32.th cycle)

Software Engineering in the IoT Context

Characteristics, challenges, and enabling strategies

Juan Pablo Sáenz Moreno

* * * * * *

Supervisor
Prof. Fulvio Corno

Politecnico di Torino
July 15, 2020

I hereby declare that, the contents and organisation of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

. . . . . .

Juan Pablo Sáenz Moreno

Turin, July 15, 2020

Summary

The Internet of Things (IoT) paradigm has given rise to a programmable world. The idea of embedding computing and communication capabilities into objects of common use has led to the development of a broad range of solutions in several domains. However, from a technical point of view, IoT systems are difficult to implement. They are typically composed of four architectural elements (devices, gateways, cloud services, and applications), and the implementation and orchestration of these architectural elements rely on several enabling technologies and spans across multiple development and execution environments. Consequently, due to the co-existence of various kinds of devices, protocols, architectures, and applications, IoT developers are required to become proficient in various and disparate areas and to consider several dimensions that are unfamiliar to most software developers.

The objective of this dissertation is to gain an understanding of the key characteristics and the most challenging issues of IoT systems development and, consequently, to propose strategies aimed at supporting the developers to overcome the complexity inherent in the development of IoT systems, and in this manner, harnessing the programmable world full potential. To that end, the first part of this thesis presents the results of an *IoT developers survey* aimed at identifying the most challenging development tasks, based on individual and group experience of 40 novice developers that worked developing IoT systems for several years of a university course. Besides, qualitative data about the causes of the identified issues were collected and analyzed. Additionally, the thesis reports a quantitative analysis of a broad set of some of the most popular publicly available IoT *Open Source Software (OSS) repositories* to provide insights into the purpose and characteristics of the code, the behavior of the contributors, and the maturity of the IoT software development ecosystem.

Upon the findings of these approaches, in the second part of the thesis are proposed: *Code Recipes*; a documentation strategy for the IoT aimed at overcoming the lack of documentation understandable by inexperienced IoT developers, and *IoT Notebooks*; an IoT-tailored literate computing approach to support the prototyping of IoT systems by enabling developers to build and share a computational narrative that may span across multiple developments and execution environments.

Acknowledgements

I would like to thank my advisor, Fulvio Corno, for his guidance and constructive criticism during these past three years. Many thanks to Luigi De Russis for his dedicated help, his thoughtful comments, and valuable advice, both personally and academically. I am also grateful to Alberto Monge for his friendship; I shared with him this Ph.D. journey from beginning to end. Furthermore, I thank the three of them most sincerely for having made me feel welcome in the group; I always felt comfortable and motivated despite being far from home.

Special thanks to Darío Correal, my former advisor from my previous university, who encouraged me to pursue a Ph.D. degree, and to do it in Italy. Looking back now, I realize how valuable that advice was.

Le agradezco a mi familia: a mi papá, a mi tío, a mi abuelita, a mis tías y a mis primos. Gracias por el cariño, las oraciones, los buenos deseos, y la buena energía que siempre me estuvieron enviando desde Bogotá. Gracias también a mis amigos que se mantuvieron en contacto y pendientes de mi.

Finalmente, el agradecimiento y el reconocimiento más especial es para mí mamá que desde siempre se ha esmerado por darme lo mejor y por verme feliz: esta etapa que estoy concluyendo es fruto de ese esmero y de su inagotable cariño. Desde un principio, cuando opté por hacer el doctorado fuera de Colombia, ella fue la que más me apoyó a pesar de lo mucho que le entristecía mi partida. Luego, a lo largo de estos años, su cercanía ha sido fundamental para afrontar los retos que se han ido presentando. Gracias de todo corazón: este logro es tan suyo como mío.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context

Nowadays, the Internet of Things (IoT) is a well-established paradigm that has gained prominence in several aspects of our everyday lives [91]. The idea of embedding computing and communication capabilities into objects of common use [65] has given rise to a programmable world, and has encouraged the development of a broad range of solutions in several domains such as smart buildings, smart cities, environmental monitoring, healthcare, logistics, smart business, smart agriculture, and security and surveillance [41, 108, 49]. As illustrated in Figure 1.1, broadly speaking, IoT systems can be characterized by four architectural elements: devices, gateways, cloud services, and applications [94].

- **Devices** are hardware elements with built-in communication capabilities that collect sensor data (*sensing devices*) or perform actions (*acting devices*). *Sensing devices* provide information about the physical entities that they monitor. This information may concern physical entity's identification (tags) or measurable qualities such as temperature, humidity, pressure, luminosity, sound level, location, images, presence, and movement, among others. They might be environmental sensors as well as wearable devices, in which case they tend to measure phisiologycal quantities. *Acting devices* refer to smart devices that cause or trigger changes in the physical environment, such as smart lights, motors, displays, etc. These *acting devices* also encompass push notifications through which end-users are informed about the occurrence of a given event.

- **Gateways or 'edge' devices** collect, preprocess, and forward the data coming from the *sensing devices* to the cloud, and, conversely, route the requests sent from the cloud to the *acting devices*. They may support tasks such as intermediate sensor data storage and preprocessing, gathering the data coming from the sensors and performing computation and reasoning tasks over

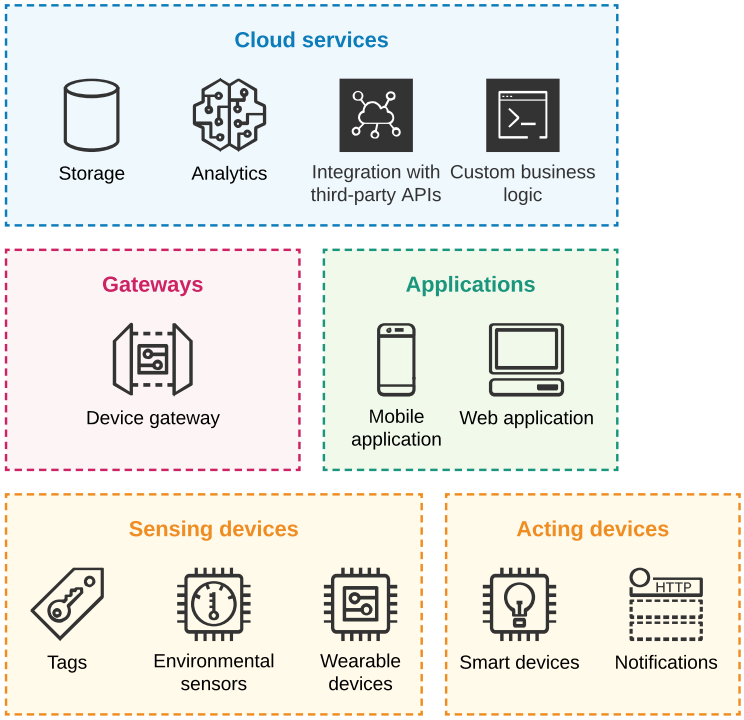


Figure 1.1: Architectural elements in IoT systems

it. If more computing or storage capacity is required, the *gateways* communicate with the *cloud services* and delegate the most demanding tasks. Furthermore, *gateways* also interact with the actuators; they control the *acting devices* based on the outputs from their computations or based on the instructions that they receive from the *cloud services*. Additionally, *gateways* support other tasks such as service discovery, geolocalization, and verification.

- **Cloud services** have three main responsibilities: acquiring and storing the data coming from the *sensing devices*, providing real-time and/or offline data analytics, and managing the *acting devices*. Data acquisition and storage concerns harvesting and storing a large amount of data collected by sensing devices for further processing and analysis. Providing real-time and offline data analytics refers to examining, cross-connecting, and transforming acquired sensor data to discover useful information, able to support decision making. Machine learning and data mining technologies and algorithms are important in this regard. Managing the *acting devices* refers to generating and delivering remote notifications as well as interfacing with third-party APIs through which certain *acting devices* can be reached and managed.

- **Applications** range from web-based dashboards to domain-specific web and mobile applications [97], and represent the mean by which end-users can visualize the device's data and status, visualize the analysis' results, and interact with the system.

The implementation and orchestration of these architectural elements relies on several enabling technologies and spans across multiple development and execution environments. These enabling technologies can be classified into identification, sensing and communication technologies, middleware components, end-user software applications, services composition, service management, and object abstraction [6]. While identification, sensing and communication technologies mainly concern hardware components, the other enabling technologies rely on *software* to address diverse features that IoT systems expose [65].

1.2 Motivation

Naturally, as IoT systems continue to gain prominence in several aspects of our everyday lives [91], so does the interest of academia and industry towards the need of supporting developers [71] and preparing different stakeholders[1] [23] to shape the future directions of IoT.

However, from the software point of view, the fact that each of the four architectural elements has a precise set of computing resources, technologies, and communication protocols, makes the development of IoT systems complex and different from the development of mobile and web applications. Indeed, two main characteristics can be identified in IoT systems from a software engineering viewpoint: their distribution over a large range of processing nodes, and the high heterogeneity of the processing nodes and the protocols used between them [66].

Consequently, IoT developers are now required to consider several dimensions that are unfamiliar to most software developers. Namely, the multi-device programming, the reactive, always-on nature of the application, and the distributed, highly dynamic, and potentially migratory nature of the software [94].

Similarly, the co-existence of various kinds of devices, protocols, architectures, and programming languages requires knowledge of various and disparate areas. Database design, mobile development, web development, embedded system development, authentication mechanisms, Application Programming Interface (API) design, and application-level protocols are some examples of the areas that are typically involved in the implementation of IoT systems. Additionally, since the implementation of IoT systems involves an unusually broad spectrum of development

[1] defined as in software engineering, i.e., people who are involved in any phases of the software development process.

technologies [97] and spans across multiple developments and execution environments, IoT developers, besides focusing exclusively on the code, are also required to deal with the hardware implementation and distributed computing concepts.

Linked to this IoT systems development inherent complexity, is the fact that no consolidated set of software engineering best practices for the IoT has emerged yet and, on the Larrucea *et al.* [60] words, *"IoT landscape resembles the wild west, with programmers putting together IoT systems in ad hoc fashion"*. These authors, in particular, remark the need for a new generation of development environments and the training of the new generation of IoT software developers.

In the same line, Colakovic *et al.* [19] hold that IoT software architectures and frameworks are necessary to overcome the inherent complexity of IoT systems and to provide an environment for services composition. In their opinion, IoT software platforms should be created as an Open Application Platform to enable modular design as well as providing an open API that would easily integrate sensors and other devices.

Furthermore, Patel *et al.* [71] draw attention to the lack of a software engineering methodology to support the entire IoT application development life-cycle, which results in highly difficult to maintain, reuse, and platform-dependent design. Finally, on the basis that IoT applications have been based on fragmented software implementations for specific systems and use cases, Weyrich *et al.* [103] propose the use of reference architectures as a mean to facilitate interoperability, simplify development, and ease implementation.

In this scenario, harnessing the programmable world full power requires to understand the peculiarities and the most challenging issues that the implementation of IoT systems pose to the developers, and accordingly, envision new software engineering and development technologies, processes, methodologies, and tools [94].

The **goals** of the research work presented in this thesis are:

1. Gaining a quantitative, as well as qualitative, **understanding** of the most challenging issues that IoT programmers face, especially novice ones, as well as the emerging characteristics and peculiarities of IoT systems development, especially in the Open Source Software (OSS) domain.

2. **Proposing** documentation, programming, and prototyping tools, consistent with the characteristics of IoT systems, and aimed at lowering the learning curve and easing the development of IoT systems.

1.3 Contribution

An accurate understanding of the characteristics of IoT systems and the issues faced by IoT developers is fundamental to envision strategies aimed at effectively lowering the learning curve and easing the development of this kind of system. As

illustrated in Figure 1.2, the contribution of this thesis is structured according to the research goals. The first phase is aimed at understanding and characterizing the IoT software development from the developers' experience perspective as well as from the analysis publicly available code. The second phase concerns proposing enabling strategies to support the development process of IoT applications based on the findings of the first phase.

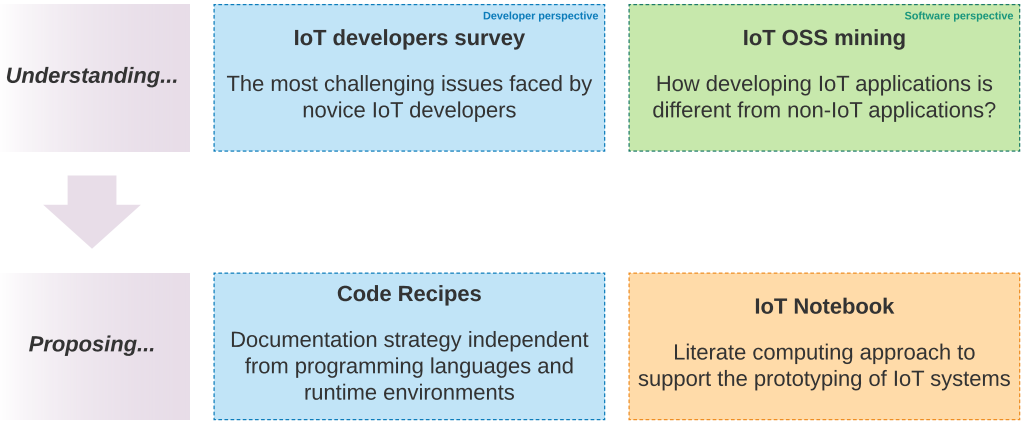


Figure 1.2: Thesis main contributions

In more detail, the main contributions are:

IoT developers survey aimed at gaining an accurate understanding, from the developers' experience, regarding the most challenging issues faced by them when developing IoT systems. This survey focused on novice developers (i.e., students or developers new to the IoT) because they are required to deal with areas in which they do not have deep knowledge, and the complexity of implementing an IoT system is even higher for them. Consequently, the outcomes from the survey can inform the design and development of adequate methodologies and improved tools to ease the development of IoT systems in the academic scenario [26, 29] (Chapter 3).

Mining IoT Open Source Software repositories to identify and provide evidence, from a practical point of view, about the IoT software development peculiarities, in the context of the most popular open-source IoT projects publicly available. This work provides insights into the purpose and characteristics of the code, the behavior of the contributors, and the maturity of the IoT software development ecosystem [28] (Chapter 4).

Code Recipes are a documentation strategy for the IoT, independent from programming languages or run-time environments, and aimed at overcoming one

of the challenging issues identified in the IoT developers survey: the lack of documentation understandable by inexperienced developers from both conceptual and technical perspectives [25] (Chapter 5).

IoT Notebook represent an IoT-tailored literate computing approach in the form of a computational notebook to the prototyping of IoT systems. Given the prominence that computational notebooks have been gaining due to their capability to consolidate text, executable code, and visualizations, this approach aims at assessing to what extent they are suitable to support the prototyping of IoT systems, even if it involves several steps and spans across multiple development and execution environments [27] (Chapter 6).

1.4 Organization

The remainder of this thesis is structured as follows:

Chapter 2 describes prior work regarding, on the one hand, the challenging issues that developers face in areas of software development that concern enabling technologies in the implementation of IoT systems, and on the other hand, the research efforts aimed at easing the development of IoT systems. Additionally, since many contemporary IoT applications employ several cloud-based advanced services, the last section of the chapter includes an overview of the services offered by some popular IoT cloud platform providers currently available in the market. Nevertheless, it should be stressed that the research presented in this thesis addresses the implementation and deployment of IoT systems as generically as possible, without tying to specific services provided by a particular cloud provider.

Chapter 3 presents the design and execution of a survey aimed at gaining an accurate understanding of the most challenging issues faced by non-experienced developers implementing IoT systems during several years of a university course. Based on their own experiences, the most challenging development tasks were identified and prioritized over a common architecture, in terms of difficulty level and efforts. Additionally, qualitative data about the causes of these issues was collected and analyzed

Chapter 4 presents a quantitative analysis aimed at understanding from a practical point of view, how developing IoT applications is different from developing non-IoT applications in the OSS context. To that end, the 60 most popular publicly available IoT and non-IoT projects on GitHub were compared and characterized according to the characteristics of the code, the behavior of the contributors, and the maturity of the IoT software development ecosystem.

Chapter 5 presents Code Recipes, a proposal to overcome the lack of proper documentation and examples identified in the previously described survey. It consists of summarized and well-structured documentation modules, independent from

programming languages or run-time environments, by which non-expert programmers can smoothly become familiar with source code, written by other developers that faced similar issues.

In the light of the architectural elements present in IoT systems and the features provided by current computational notebooks, **Chapter 6** presents the design, implementation and preliminary assessment of an IoT-tailored notebook aimed at helping developers to build and share a computational narrative around the prototyping of IoT systems.

Chapter 7 concludes the thesis and provides insight into future research opportunities.

Chapter 2

Related works

In accordance with the two research goals outlined in the previous section, the related works presented in this Chapter are structured in this manner:

- Section 2.1 provides context to the first research goal, and specifically to the IoT developers survey presented in Chapter 3. The works described in this section, on the one hand, aim at identifying the challenging issues that developers face in those areas of software development that are commonly involved as enabling technologies in the implementation of IoT systems, such as mobile development and APIs development. On the other hand, the last work evaluates the effects that the choice of programming language, problem-solving training, and the use of formative assessment have on learning to program, in general.

- Additionally, since the IoT developer survey relied on novice programmers to identify the most challenging issues that IoT development poses, Section 2.2 encompasses existing research works regarding the experience of novice developers implementing IoT systems in academic settings.

- In line with the research presented in Chapter 4, intended to provide insights into the peculiarities of IoT development in the OSS context, Section 2.3 describes research works related to the needs and challenges of software engineering in the IoT context and Software Mining research in other fields different from IoT.

- Section 2.4 provides context on the Code Recipes (Chapter 5) and IoT notebook (Chapter 6) by presenting research works where the authors discuss the issues and technical challenges that the software development of IoT systems pose, and propose methodologies, frameworks, and architectures to address them.

- Section 2.5 describes research works concerning the characteristics, current use, and opportunities of Computational Notebooks.
- Finally, Section 2.5 provides an overview of some popular IoT cloud platform providers currently available in the market, in terms of the computing, storage, and communication services that they offer. Such overview corresponds to part of the work that I published in [22].

2.1 Identifying programmers issues

This section presents a set of related works that relied on interviews, surveys, and controlled studies with software developers to identify the challenging issues present in mobile applications development, APIs usage, and learning to program. These related works concern areas of software development that are commonly involved in the implementation of IoT systems. For instance, mobile applications are generally the means by which end users interact and configure the whole system; similarly, the integration between subsystems is typically achieved through RESTful APIs; and naturally, the implementation of the system may require programming expertise in more than one programming language.

Ahmad et al. [4] aimed at identifying the challenges that can undermine the successful development of native, web, and hybrid mobile applications. First, the authors identified the challenges through a systematic literature review, and then they validated the challenges through interviews with practitioners. From the systematic literature review, nine challenges emerged, and from the interviews, four additional challenges were identified. In these interviews, 34 mobile developers with 2-5 years of experience were recruited and instructed to rate each challenge on a Likert scale. Interestingly, the distinction between the three types of mobile applications (native, web, and hybrid), that I also considered in the survey that I conducted (Chapter 3), enabled the authors to accurately identify the most critical challenges on each type. Indeed, after comparing the development challenges of these three types of mobile applications, the authors determined that there are statistically significant differences among them. Concretely, fragmentation and change management are more critical in native, the user experience is more critical on the web, and compatibility is more severe on the web. As will be described later in Chapter 3, the results of my survey are consistent with the fact that challenges vary according to the type of mobile application.

Joorabchi et al. [51] aimed at gaining an understanding of the main challenges developers face in practice when they build apps for different mobile devices. To that end, they first conducted a qualitative study consisting of interviews with 12 expert mobile developers, and then, they carried out a semi-structured survey with 188 respondents from the mobile development community at large. Authors identified the existence of multiple mobile platforms as a major challenge for developing

mobile apps; developers are required to learn more languages and APIs for the various platforms and, at the same time, remain up to date with the frequent changes of each Software Development Kit (SDK). Additionally, due to this fragmentation, developers have to keep checking the correctness and consistency of the app across different platforms. Developers also indicated that testing tools and emulators (at the time in which this study was conducted) were not able to sufficiently support important features and scenarios such as mobility (changing network connectivity), location services, sensors, or various gestures and inputs. This lack of tools makes analysis and testing even more challenging. Finally, concerning usability, the study results suggested that the implementation of a reusable user interface is challenging due to the trade-offs that developers are required to achieve between maintaining consistency and adhering to each platform's standards. In this respect, the results obtained from the IoT developers survey (Chapter 3) confirm that the configuration of the development environment (involving dependencies, SDKs, and run-time platforms) is perceived as challenging in the mobile application development.

Sohan et al. [90] conducted a controlled study with 26 experienced software engineers to understand the issues that REST API client developers face while using an API without examples. To that end, participants were divided into two groups and given the same set of 6 API tasks to complete. While one group was given the official REST API documentation, the other group was given an enhanced version of the official documentation where three usage examples were added. From the analysis of 539 API calls, 385 from the first group and 152 from the second, authors determined that, without examples, REST API client developers struggle with using the right data types, data formats, and required HTTP requests headers.

Similarly, Robillard et al. [80] conducted a study aimed at identifying some of the most severe obstacles faced by experienced developers, with an average of 9.8 years of professional experience, when learning new APIs. Such study involved 440 professional developers and was structured around: (i) an exploratory survey to broadly identify *what makes APIs hard to learn*; (ii) a set of qualitative interviews to understand API learning obstacles in detail; and (iii) follow-up survey to confirm the general findings and collect additional demographic data that would help to explain API learning obstacles. The study identified inadequate API documentation as the most severe obstacle facing developers learning a new API. For this reason, based on the qualitative analysis, the authors elicited a set of important factors to consider when designing API documentation. Among their various observations, they stated that *small examples that nevertheless demonstrate API usage patterns involving more than one method call will be more useful than single-call examples.* Furthermore, they determined that *a central challenge when learning APIs is discovering how to match scenarios with the API elements that support this scenario.*

Uddin et al. [100] conducted two surveys about API documentation quality involving 323 software professionals. In the first survey (exploratory), authors aimed at collecting good and bad examples of API documentation. The respondents were

asked to provide examples of good or bad documentation, based on the last development task that they completed, in which they had to consult API documentation. In the context of this exploratory study, there was no standard definition for an API; it could be a library, a framework component, or even a Web API. In the exploratory survey participants were asked to: (i) describe their last development task they had completed that required them to consult API documentation; (ii) provide up to three examples of API documentation that they found useful their corresponding justification; (iii) provide up to three examples of API documentation that they did not find useful and their justification. From the analysis of the results, ten common documentation problems emerged, and they were categorized by the authors into content and presentation problems. Namely, content problems comprised incompleteness, ambiguity, unexplained examples, obsoleteness, inconsistency, and incorrectness. Presentation problems, for their part, concerned bloat, fragmentation, an excess of structural information, and tangled information. In the second survey (validation), the authors assessed the frequency and severity of the previously identified problems. This validation survey was conducted with a different group of participants, and they were asked to (i) rate, for each one of problems, how frequently they were experienced and how severe they were when completing the participants' development tasks; and (ii) to identify the three most painful problems to be prioritized. In this manner the authors analyzed the problems' frequency, their severity, and the necessity to solve them. Ambiguity and incompleteness were identified as the most critical problems, and together with incorrectness, they were into the top three priorities for improving documentation. The major finding of the surveys was that the most frequent and common problems had to do with content. In fact, all content problems were prioritized over presentation problems. Additionally, the hardest problems with API documentation were also the ones requiring the most technical expertise to solve. Completing, clarifying, and correcting documentation require deep, authoritative knowledge of the API implementation. Finally, the authors envisioned recommendation systems as a mean to reduce as much of the administrative overhead of documentation writing as possible, enabling experts to focus exclusively on the value-producing parts of the development tasks.

Koulouri et al. [59] assessed the effect of three factors on learning to program, namely: choice of programming language, problem-solving training, and the use of formative assessment. To that end, the authors conducted a study that adopted an iterative approach and was carried out across four consecutive years involving four experimental groups of CS1 students. These groups corresponded to distinct full student cohorts and were organized in this manner: a control group that was taught using Java (157 students), a group that was taught using Python (195 students), a group that received formative feedback (193 students), and a group that received initial problem-solving training (216 students). From the iterative process, the following outcomes emerged: (i) the choice of programming language seems to

affect student learning, a simpler syntax could have a greater impact because it makes loops easier to use, and the underlying concept easier to understand; (ii) introducing problem-solving concepts before teaching more specific programming aspects has an impact on how students learn to program, it helps students to develop an ability to both break down complex problems into subtasks and produce the correct sequence of actions while accelerating the consolidation of concepts, such as data and control structures, introduced later in the course; (iii) Formative feedback may not be necessarily and effective as expected unless students are ready to have a proactive role in seeking and responding to feedback, it is advised that for formative feedback to yield observable benefits on their performance, novice programming students may need to be externally motivated and guided.

2.2 Novice Programmers in the IoT

Literature on *novice programmers* in the IoT mainly consists of experience reports from college or university-level courses, in which teachers and instructors recognize the needs and challenges brought by the IoT and intervene either with new methodologies or with dedicated frameworks. However, a systematic collection and description of pain points and issues encountered by these novice developers was not performed in any of these works, to my knowledge. Four of the most representative works are reported below.

To provide its students with the systems-level skills needed to understand and develop complete IoT systems, in 2014 Politecnico di Torino, in Italy, initiated a course named "Ambient Intelligence." In this project-based course, a teamwork and design-driven *methodology* is applied to teaching IoT system design [23]; core student skills acquired in previous courses are exploited in a multidisciplinary project work. The main topic of the course is the design and the implementation of prototype Ambient Intelligence (AmI) systems [20], a field closely related to the IoT. This entails a strong focus on the application and on user needs. From the beginning of the course, students form three- to four-people teams and are guided to define the requirements for a system, and then to design and implement a limited but working prototype. Every year, a theme is chosen for the projects. The theme is wide enough to generate around 20 projects, but sufficiently well-defined to determine whether a project fits. After teacher's approval, the teams develop their ideas according to the proposed design methodology, which follows four main steps: vision and goal definition; functional and nonfunctional requirements elicitation; system architecture design and component selection; and practical realization of the prototypical system. Projects cannot be mobile-only, software-only, or hardware-only solutions. Instead, they must exploit different platforms and mix hardware with software and user interaction, as typical IoT systems do. The resulting system and the "deliverables" produced throughout the semester are the focus of the course

exam, which also includes a presentation and demo of the team projects and an oral discussion. The authors, in their paper, provide positive qualitative and quantitative results about the students' ability to understand and design IoT systems; the usage of required languages, frameworks, and protocols; and employed communication, collaboration, and management skills. Moreover, they present a series of "lessons learned" that may allow other instructors to design IoT-related courses by following a similar methodology. Indeed, the subjects involved in the IoT developers survey that I conducted are a subset of students of the Ambient Intelligence course.

In a similar way, Kortuem et al. [58] describe the experience of the Open University, in the United Kingdom, delivering an online course whose purpose was to "place the IoT at the core of the first-year computing curriculum and to prime students from the beginning to meet the coming changes in society and technology". Among the concepts that the course designers identified as fundamental for the IoT and essential for the course, they list: the merging of the physical and digital realms; the huge increase in the number of Internet-connected devices, objects, sensors, and actuators; and the emergence of novel embedded-device platforms below the level of personal mobile devices. A key goal of the course was to empower novices and to make IoT technologies accessible to students with no prior programming skills. One of the most challenging issues faced when designing and delivering the course was that most embedded device technologies require an understanding of software and hardware that cannot be expected from first-year undergraduates. To overcome this issue, an embedded networked sensor was custom-designed for this course, as well as a newly developed visual programming language and environment, and a cloud infrastructure that connected the embedded networked sensors of all students together. Authors determined, based on programming assignments and tests with prospective users during the design stage of the course, that new users can produce a working program in less than 20 minutes during their first session with the custom embedded networked sensor. Furthermore, after a few sessions, novices with no exposure to programming before the course, could understand and modify given programs and develop new ones on their own. Moreover, their proposed programming language and environment help novices to quickly develop an understanding of the principles of programming simple IoT applications. To gain an understanding of the common issues that novices experience, authors conducted a preliminary analysis based on the activity of the help forum. However, the article just provides a very general description of the issues that concerned their proposed IoT teaching infrastructure.

Dobrilovic et al. [37] built a platform to teach communication systems, and designed a second one [36] to be used in university curricula for teaching IoT. The first platform was built to be used within the curricula of Information Technology and Software Engineering, where a strong background in electronics is not expected. The basis of the platform was built upon an Arduino micro-controller

and includes Zigbee expansion shields, different types of sensors, and a packet sniffer specially developed for analyzing ZigBee, Bluetooth Low Energy, and IEEE 802.15.4 traffic. Moreover, the authors described three scenarios for the usage of this platform: a system for temperature monitoring, a Radio-Frequency Identification (RFID)/ZigBee network for tracking human resources, and a smart agriculture and air pollution monitoring system. Starting from the architecture proposed for each scenario, students were asked to deploy such scenarios from the beginning, and develop applications on top of them. Specifically, the platform was used by a group of three students that were able to deploy the temperature monitoring scenario by adding more sensors and developing an application to gather real-time data from an Arduino micro-controller. The second platform [36] is proposed upon an open-source architecture for teaching IoT. It consists of a set of low-cost open-source hardware components (*IoT education kit*), along with the list of software components required to develop IoT custom applications, and the network protocols required to establish the communication between the layers of the proposed IoT teaching architecture. However, this platform was not used during laboratory exercises. Instead, it was presented to students as a part of the lectures, aiming at explaining the functionality and implementation of each layer of the architecture. Therefore, students' feedback was not collected in a formal questionnaire nor analyzed. The platform acceptation and effectiveness was assessed based on the positive comments that the students made. According to the authors, students accepted the platform with good attention and interest to work with it.

2.3 IoT development in the OSS context

The study of IoT OSS, through the data analysis described in Chapter 4, lies in the software engineering domain and is intended to provide insights into the peculiarities of IoT development in the OSS context. Although various authors have pointed out the need for research on software engineering for IoT systems, given the several challenges that the development of such systems poses, to the best of my knowledge, no other research aimed at exploring and analyzing how developers work within several OSS IoT projects. In the following the related works are approached from two areas: the needs and challenges of software engineering in the IoT context, and Software Mining research in other fields different from IoT.

According to Morin *et al.* [66], IoT applications have two main characteristics from a software engineering viewpoint. The first is their distribution over a large range of processing nodes. The second is high heterogeneity of the processing nodes and the protocols used between them. To deal with these characteristics, authors introduce a modeling language aligned with UML, an advanced multiplatform code generation framework, and a methodology specifying the development processes and tools used by both IoT service developers and platform experts.

Similarly, Colakovic *et al.* [19] hold that IoT software architectures and frameworks are necessary to overcome the inherent complexity of IoT systems and to provide an environment for services composition. In their opinion, IoT software platforms should be created as an Open Application Platform to enable modular design as well as providing an open API that would easily integrate sensors and other devices.

According to Larrucea *et al.* [60], no consolidated set of software engineering best practices for the IoT has emerged yet. On the author's words, *"IoT landscape resembles the wild west, with programmers putting together IoT systems in ad hoc fashion"*. They consider that industry needs guidance to engineer the new generation of scalable, highly reactive, often resource-constrained software systems characteristic of the IoT. Among such guidance, authors remark the need for a new generation of development environments and the training of the new generation of IoT software developers.

Regarding IoT projects in OSS, Taivalsaari *et al.* [97] hold that nowadays nearly all the component areas of a typical IoT cloud back-end architecture can be constructed from open source technologies. On their opinion, given the availability and maturity of open source components, the role of back-end developers today could be characterized more as software composition or orchestration instead of traditional software development.

Concerning sotfware mining, as mentioned before, the methodology followed in this work took inspiration from the work of Pascarella *et al.* [70], in the video games OSS context. The authors conducted a study on 60 projects, and their results confirmed the existence of significant differences between game and non-game development, in terms of how project resources are organized and in the diversity of developers specializations. Another source of inspiration was the work of Ray *et al.* [78]: they performed a large scale study on GitHub about the of programming languages type and use on software quality. They examined the interactions of language, domain, and defect type through a combination of regression modeling, text analytics, and visualization. Their results suggested that strong typing is modestly better than weak typing, and among functional languages, static typing is also somewhat better than dynamic typing. However, authors point out that effects arising from language design are overwhelmingly dominated by the process factors such as project size, team size, and commit size. Additionally, they determined that the defect proneness of languages, in general, is not associated with software domains.

2.4 Easing the Development of IoT Systems

Taivalsaari et al. [94] present a roadmap from today's cloud-centric, data-centric IoT systems to a world in which everyday use objects are connected and the network's edge is programmable. On the basis of the authors' experience, they highlight issues and technical challenges that the *Programmable World* poses to software developers. In their opinion, the average mobile or client-side web application developer is not well equipped to cope with the challenges of IoT systems development. Moreover, today's development methods, languages, and tools are poorly suited to the emergence of millions of programmable things. In particular, IoT developers must consider several dimensions that are unfamiliar to mobile and client-side web application developers, namely: multidevice programming; the reactive, always-on nature of the system; heterogeneity and diversity; the distributed, highly dynamic, and potentially migratory nature of software; and the need to write software in a fault-tolerant and defensive manner. Among all the statements discussed in that article, two of them are of special relevance:

- educating software developers to realize that IoT development truly differs from mobile and client-side web application development;
- to harness the Programmable World's full power, we will need new software engineering and development technologies, processes, methodologies, and tools.

Patel and Cassou [71] tackle the challenges brought by application development in the IoT by proposing an "high-level" development methodology that separates IoT application development into different concerns and provides a conceptual framework to develop an application. They recognize that software development in the IoT presents various challenges, and they list four of them: *a)* lack of division of roles, *b)* heterogeneity (of devices), *c)* scale (of IoT systems), and *d)* different life cycle phases. However, these challenges were formulated starting from the authors' unstructured analysis of example applications and from related work in closely related fields like Wireless Sensor Networks and Ubiquitous Computing [101]. The main goal of the work of Patel and Cassou is, indeed, to make IoT application development easy for stakeholders by taking inspiration from the Model-Driven Design (MDD) approach and building upon work in sensor network macroprogramming, thus reducing development efforts.

Datta and Bonnet [34], similarly, start from their own experience (i.e., from the IoT data cycle presented in [35]) to propose a list of the top 8 requirements for building an IoT application framework: interoperability, open source framework, strong security by design, etc. Then, they introduce DataTweet, a framework that decouples application logic from common IoT functionalities. This allows IoT stakeholders to focus on the application logic and use open source, standardized APIs for

the latter. An example with an automotive IoT application for an Advanced Driver Assistance System was developed with the framework and its operational phases were highlighted in the paper. The framework aimed at simplifying the development process, hiding the complexities of programming and security mechanisms from developers, and reducing time to market for industries.

According to Weyrich and Ebert [103], too, software engineering for the IoT poses challenges in light of new applications, devices, and services. Moreover, such new and diverse applications, devices, and services "*pose specific challenges for specifying software requirements and developing reliable, safe software*" [103]. Weyrich and Ebert, in their paper, state that reference architectures may help developers meet those challenges. They focus on two major architectures from an industry standpoint: the *Internet of Things - Architecture* (IoT-A)[1] and *Industrial Internet Reference Architecture* (IIRA)[2]. IoT-A delivered a detailed architecture and model from the functional and information perspectives, while IIRA was delivered by the Industrial Internet Consortium (founded by AT&T, Cisco, General Electric, IBM, and Intel) for a broad consideration and discussion. Such architectures can serve as an overall and generic guideline, and not all domain applications will require every component for real-life development. While such reference architectures are not equal and a "standard" architecture did not yet prevail, they form a superset of functions, information structures, and mechanisms that could provide developers with a more complete view of the IoT system they will implement.

As will be described in Chapter 3, in the IoT developers survey respondents were referred to a "generic architecture", which shares most of the functionalities and building blocks with the two aforementioned architectures and was simplified and customized to the type of IoT projects they had experience on. Moreover, it includes contributions coming from other IoT reference architectures, namely the Intel IoT Platform Reference Architecture [48], the IBM IoT Reference Architecture[3], and the Microsoft Azure IoT Reference Architecture [64]. As already highlighted by Weyrich and Ebert [103], all these architectures are not equal but they present some common features. The purpose of using this generic architecture was to provide respondents with a common understanding about the software components involved in an IoT system.

[1] https://cordis.europa.eu/project/rcn/95713_en.html, last visited on May 24, 2019.

[2] https://www.iiconsortium.org/IIRA.htm, last visited on May 24, 2019.

[3] https://www.ibm.com/cloud/garage/architectures/iotArchitecture/reference-architecture/, last visited on May 24, 2019.

2.5 Characteristics and opportunities of Computational Notebooks

The computational notebook approach, described in Chapter 6, lies in the software engineering domain and is intended to provide insights about the suitability of a computational narrative approach to document, execute, and share the steps involved in IoT prototyping, especially for novice programmers. In the following, the related work is addessed from the perspective of (i) exploring and analyzing the current use of notebooks, and (ii) customizing them to fit into a particular context.

Rule *et al.* [84] assessed the current use of computational notebooks through quantitative analysis of over 1 million notebooks shared online, qualitative analysis of over 200 academic computational notebooks, and interviews with 15 academic data analysts. These analyses demonstrated a tension between exploration and explanation that complicates construction and sharing of computational notebooks. In the context of data analysis, the exploratory process tends to produce messy notebooks. The authors determined that a key challenge concerns the development of tools aimed at augmenting analysts' workflows and facilitating organization and annotation without much additional effort. Furthermore, they claim that a notebook "clean up" tool that moves the imports to the beginning, as well as rewriting reusable code as functions, would improve maintainability and legibility in the long run.

Pimentel *et al.* [75] present an analysis of the notebook characteristics that impact reproducibility. Authors propose a set of best practices that can improve the rate of reproducibility and discuss open challenges that require further research and development. Among these best practices authors suggest to declare the dependencies in requirement files and pin the versions of all packages; use a clean environment for testing the dependencies to check if all of them are declared; put imports at the beginning of notebooks; use relative paths for accessing data in a repository.

Head *et al.* [43] present a collection of code gathering tools, extensions to computational notebooks that help analysts find, clean, recover, and compare versions of code in cluttered inconsistent notebooks. Additionally, the authors conducted a qualitative usability study with 12 professional analysts and found that this kind of tools was considered useful for cleaning notebooks and generating personal documentation and light-weight versioning.

Yin *et al.* [106] describes CyberGIS-Jupyter, a framework for achieving data-intensive, reproducible, and scalable geospatial analytics using the Jupyter Notebook based on ROGER, the first CyberGIS supercomputer. With this proposal, the authors aimed at achieving agility and reproducibility in the field of geospatial

analytics. On one hand, agility concerns the use of a Jupyter notebook as a Graphical User Interface (GUI) development platform for CyberGIS instead of developing customized and web-based GUI interfaces that require professional skills that geospatial researchers do not possess. To do so, they developed a set of utilities to support common CyberGIS operations, using a Jupyter Interactive Widgets library. On other hand, concerning the reproducibility, the authors relied on container virtualization technologies to record and reproduce computational environments with the exact versions of all external libraries. Hence, CyberGIS-Jupyter is deployed inside cloud infrastructure and, for each user instance, their Jupyter notebooks are hosted inside one dedicated container. In this manner, the framework enables researchers to share and build on each other's work to innovate large-scale geospatial analytics cumulatively in a collaborative fashion for team-based development.

Merino *et al.* [63] developed a language parametric notebook generator for Domain-Specific Languages (DSL) in the context of the Jupyter framework. Authors aimed at enabling language engineers to easily implement Jupyter language kernels for their DSLs by reusing, as much as possible, existing language components, such as parsers, code generators, and Read-Eval-Print Loops (REPLs). Since developing a language kernel from scratch requires a lot of effort and communication with Jupyter's low-level wire protocol, authors aimed at hiding the low-level complexity of Jupyter's wire protocol by providing generic hooks for registering language services. In this manner, obtaining a notebook interface for a DSL becomes a matter of writing a few lines of code.

Azzara *et al.* [8] present PyoT, a macro-programming framework for the IoT aimed at simplifying the development of IoT applications. PyoT manages IoT-based Wireless Sensor Networks (WSN), letting programmers focus on the application goal. This framework allows developers to (i) automatically discover available resources; (ii) monitor sensor data; (iii) handle its storage; (iv) control actuators; (v) define events and the actions to be performed when they are detected; and (vi) interact with resources using a scripting language (macro-programming). PyoT hides the complexity of the network by abstracting it as a set of software objects, each of which represents a physical resource (actuators or sensors), its available operations, and its location. Furthermore, since the macro-programming uses Python, authors relied on the Jupyter computational notebook as the web-based user interface through which users can interact with the resources executing basic operations such as resource listing, sensor monitoring, actuator control, event detection and reaction, and access to historical data.

The website [83] shows the setup and use of a custom kernel to run CircuitPython code directly from a Jupyter interactive notebook. CircuitPython is a programming language designed to simplify experimenting and learning to code on low-cost microcontroller boards [18]. Specifically, the CircuitPython kernel is a wrapper that allows CircuitPython's REPL to communicate with Jupyter's code

cells. Using this kernel enables the code to be developed and hosted in the web-browser and executed on the CircuitPython hardware through a serial Universal Serial Bus (USB) link.

2.6 IoT cloud platforms

Most of the contemporary IoT applications employ several *advanced services*, typically cloud-based, that go beyond the communication requirements: e.g., different storage capabilities or notification and alerting functionality. Nowadays, developers rely on such centralized cloud providers (over-the-top players) for their IoT applications. In this context, given the relevance that IoT cloud platforms have gained in the development of IoT applications, this section aims at presenting an overview of the most popular ones, in terms of the computing, storage, and notification services that they offer. Concretely, such services were compared according to a vendor-neutral set of criteria.

The choice of the most widespread IoT platforms included in this overview was made according to market and technology benchmarks. In particular, I selected the leader platforms indicated by the IDC MarketScape report [30] and by the CXP Group IoT Platforms vendor benchmark [32]; the 11 selected platforms are listed in Table 2.1.

Table 2.1: IoT cloud platforms

Arrayent	Arrayent IoT Cloud Services
Amazon	AWS IoT Core
Bosch	Bosch IoT Suite
General Electrics	General Electrics Predix
Google	Google Cloud IoT Platform
IBM	IBM Watson IoT Platform
Microsoft	Microsoft Azure IoT
Oracle	Oracle IoT Cloud Service
SAP	SAP IoT Platform
thinger.io	thinger.io
Xively	Xively

The first step of the overview was to define a set of macro-categories (criteria) regarding the services, to enable a fair and transparent comparison of the platforms. Then, for each platform, it was determined whether a given service is offered and which product of the platform provides it. Among the identified criteria, three of them (Data Storage, Push Notifications, and Virtual Devices) were decomposed into smaller features so that the analysis could be more specific and accurate.

2.6.1 Comparison Criteria

Hereinafter are described the details of the comparison criteria, and the main observations that emerged from the comparison itself. Eight criteria were derived for the IoT services to perform a well-balanced comparison between the considered IoT cloud platforms. They encompass a wide variety of domains, from data storage to notification systems and Software Development Kit (SDK). The criteria are:

1. **Data Storage**. Whether and how the platform provides storage capabilities. In IoT cloud platforms these services are managed differently depending on the source, the format, and mainly, the purpose of the data. Most analyzed platforms offer the following services: *Disk storage* for I/O-intensive applications with low latency and high throughput (e.g., Amazon Elastic Block Store); *NoSQL database* storage for semi-structured data (e.g., Google Cloud Datastore); *BLOB storage* that consists of massively-scalable object storage for unstructured data like images, videos, and audio (e.g., Azure Blob Storage); *File storage*, corresponding to cross-platform file system (e.g., Amazon Elastic File System); *Relational database* management (e.g., Google Cloud SQL).

2. **Devices SDK**. Whether the platform provides a SDK to connect and (remotely) manage IoT devices.

3. **Mobile SDK**. Whether the platform provides a SDK to enable the interaction of mobile apps with IoT devices.

4. **Push Notifications**. Whether the platform provides a push notification or real-time alert mechanism. Common features involve the management of topics, which are communication channels to send messages and subscribe to notifications. The features are: *Register a device as an endpoint*, meaning that it will be receiving notifications; *Creates a topic* to which notifications can be published; *Delete a topic* along with all the endpoints subscribed to that topic; *Subscribe an endpoint device to a topic* so that the concerned endpoint is enabled to receive all the messages published to that topic; *Removing an endpoint device from a topic* to delete a subscription; *Send a notification to a single device* using some specific identifier; *Send a notification* to all the devices that have been subscribed to a given topic. Moreover, it is also possible to *Integrate with the custom notification service of various OS*, i.e., send push notification messages to mobile devices by using their supported push notification services (e.g., Apple Push Notification Service, or Google Cloud Messaging).

5. **REST APIs**. Whether the platform provides REST APIs to enable the integration with software applications.

6. **Supported protocols**. Which protocols can be used to communicate between the IoT devices and the platform.

7. **Virtual devices**. Whether and how the state of a IoT device is stored in the platform, so that the physical device can be remotely controlled. To this end, a virtual representation of the physical device (also called *device twin*) is registered and controlled through the platform. The set of features commonly provided by the IoT platforms are: *Create virtual devices*; *Retrieve virtual device by id*; *Update a virtual device*; *Replace the device properties*, which means to overwrite the properties of the device through its twin; *Define the structure of the device twins metadata* according to the physical device capabilities, and *Remotely assign jobs to the virtual device* executes the deployment of a given function.

8. **Analytics**. Whether the platform provides a graphical interface (e.g., a dashboard) through which users can visualize and manage the deployed IoT devices.

2.6.2 Results

Table 2.2 shows the selected platforms (first column), along with the comparison criteria (first row), where cells indicate whether the platform offers the given service (✘), or even the name of the platform's product providing it.

It can be noted that most cloud platforms overlap in the provided services. In particular, AWS IoT Core, Google Cloud IoT, IBM Watson IoT, Microsoft Azure IoT, Oracle IoT Cloud Service, and SAP IoT rely on a wide catalog of generic cloud services, not strictly related to the IoT ecosystem. They provide, broadly speaking, the same set of services. Furthermore, push notifications are mostly managed through the MQTT publish-subscribe messaging protocol.

The Bosch IoT Suite and the Arrayent IoT Cloud Services, conversely, are the most "limited" platforms. They do not provide any data storage support, SDK for other applications, nor support for virtual devices. Additionally, a small separation in the supported functionality could be observed: while "traditional" ICT vendors (e.g., Google, Microsoft, Oracle) support different facets of the development and deployment of IoT applications, vendors that come from other domains (e.g., Bosch, General Electrics) expose a narrower and more focused set of functions. Finally, most provided services are tuned for cloud-based applications, not specifically for the IoT: long term data storage or analytics are two important examples.

Table 2.2: Service Comparison for IoT Platforms

Platform	Data storage	Devices SDK	Mobile SDK	Push notifications	REST APIs	Supported protocols	Virtual devices	Analytics
Arrayent			Android, iOS	Realtime Alerts	EcoAdaptor framework	HTTPS, WebSockets		✘
Amazon	S3	AWS Greengrass	Android, iOS	Amazon SNS	✘	HTTP, MQTT, WebSocket	✘	AWS Console
Bosch		IoT Remote Manager		Remote Event Push	Java client or HTTP API	HTTP, MQTT, LWM2M, mPRM	✘	IoT Developer Console
General Electrics	Blobstore (S3)	Predix Machine	Predix SDK for Hybrid		Asset Services	HTTPS, WebSockets	Mobile gateway	
Google	Cloud Storage	✘	✘	Cloud Pub/-Sub	Google Cloud IoT API	MQTT, HTTP	Cloud Pub/-Sub (7 days)	Google Data Studio
IBM	Bluemix Storage	Edge Analytics SDK	Android	Bluemix Push Notifications	✘	MQTT, HTTP	MQTT	Watson IoT dashboard
Microsoft	Azure Storage	Device Provisioning	Android, iOS	Notification Hubs	✘	MQTT, HTTPS, AMQP	IoT Edge	✘
Oracle	✘	Endpoint Management	Java, iOS	✘	✘	MQTT, HTTPs	✘	✘
SAP	✘	✘	Cloud Platform, iOS	Apple Push Notification Service	✘		Thing Registry	
thinger.io	Data Bucket	Arduino, Sigfox or Linux	Android appplication		Server API	HTTPs	✘	Cloud Console
Xively		✘	Template mobile apps	Alerting and monitoring	✘	MQTT, HTTP	MQTT	✘

Legend: empty = not supported; ✘ = supported; other = supported with product name

Chapter 3

On the challenging issues faced by IoT novice developers

This Chapter describes the design and application of a survey among 40 novice developers that worked in groups developing IoT systems during several years of a university course. Based on their own experiences, individually and as a group, the most challenging development tasks were identified and prioritized over a common architecture, in terms of difficulty level and efforts. Besides, qualitative data about the causes of these issues was collected and analyzed.

Part of the work described in this chapter has been previously published in two different papers. The initial pilot survey aimed at validating the pertinence and completeness of the questionnaire through a preliminary study with a small group of participants was describedin [26], while the application of the final survey was presented in [29].

3.1 Motivation

As previously discussed, the co-existence of various kinds of devices, protocols, architectures, and applications make IoT systems complex to develop, even for experienced programmers. When novice programmers are learning to implement these systems, they are required to deal conceptually and technically with areas in which they do not have deep knowledge nor previous experience. Additionally, besides becoming proficient in these areas separately, they should integrate them and build a system whose components are heterogeneous from both software and hardware perspectives. In sum, the inherent complexity of such IoT systems raises particular concerns if we focus on novice programmers and how to effectively and easily allow them to design and develop these systems. Indeed, although our survey is not focused on them, these concerns are partially shared by programmers that come from another background and are new to the IoT. For this reason, an accurate

understanding of the most challenging issues that novices experience is fundamental to envision strategies to enable a smoother development of IoT systems. To my knowledge, no previous work assessed which were these issues.

Prior work on the topic of easing the development of IoT systems has focused on providing suitable *methodologies* and *frameworks* to *professional* developers [17, 34], starting from a few challenges extracted from unstructured analysis of IoT applications, at best [71].

In an effort to understand which challenges *novice developers* face when building IoT systems, in late 2017 I conducted a survey involving 40 novice programmers coming from an engineering background. Such developers were recruited among the former students of an undergraduate course, during which they worked in groups to develop different IoT systems. Naturally, the systems taken into account in this work are not large-scale systems but prototypes with didactic purposes. In particular, the research questions addressed by the survey were:

RQ3.1: How complex, in terms of time spent and difficulty, are the software development tasks needed to build an IoT system?

RQ3.2: Which are the software development tasks that are perceived as the most challenging to complete?

RQ3.3: Why are these tasks perceived as the most challenging?

This survey contributes to the body of research on easing the development of IoT systems, with a focus on novice programmers. Here, "IoT systems" is meant broadly and encompasses several application domains (e.g., healthcare, smart home, ...) as well as different IoT devices and technologies. For this reason, in the survey, a generic architecture for IoT systems was used as a reference, for providing a common vocabulary to the respondents. The survey results present insights about novices' experiences when working on IoT systems, and reveal that the integration of those parts of an IoT system that require over-the-network communications is one of the most challenging tasks. The outcomes also reveal that the interaction and interfacing with third-party cloud services is perceived as an important issue to be tackled, mainly for the lack of proper documentation and examples. Overall, the results can inform the design and development of adequate methodologies and improved tools to ease the development of IoT systems in general, and for novice programmers in particular. Furthermore, I consider that the results of this survey, obtained from respondents belonging to the academic setting, might be partially valid as well to software companies, in particular by considering the work by Salman et al. [86], about how well students represent professionals in software engineering experiments. According to this work, a major differentiating factor affecting the results might be subject's experience levels rather than the experiment setting (classroom or industry).

3.2 Survey design and methods

The goal of the survey was to identify, based on the personal experience of novice developers, the most complex issues that they faced when developing an IoT system as well as the principal causes of such complexity. It was advertised and conducted online, and both quantitative and qualitative data were gathered. The following is a detailed description of the survey methodology.

3.2.1 Instrument development

As already mentioned, the term "IoT systems" in the context of this study is meant broadly and encompasses several application domains, as well as diverse IoT devices and technologies. For this reason, in the development of the survey, it was imperative to provide the respondents with a common understanding about the software components involved in an IoT system. Hence, the survey was structured around a generic IoT systems architecture proposed by me along with a predefined set of software development tasks compatible with such architecture.

Such generic architecture was built, firstly, by taking into account the functionalities and building blocks suggested in the IoT reference architectures mentioned in the Related Works (Section 2.4), and secondly, by analyzing the architectures used in the development of prototype IoT systems during several years of the Ambient Intelligence course (described in Section 2.2, concerning the work of Corno et al. [23]), whose students were the subject of this study. In this manner, I made sure that the resulting common architecture would be understandable and familiar to the IoT novices participating in the study.

Consequently, five interconnected subsystems were characterized, as illustrated in Fig. 3.1. The resulting subsystems are:

Sensors monitor the End-user activities and detect changes in the environment by measuring variables such as temperature, humidity, and occupation, among others. They generally refer to wearable devices and environmental sensors.

Gateways gather the data coming from the Sensors and perform computation and reasoning tasks over it. If more computing or storage capacity is required, the Gateways communicate with the Back-End subsystem and delegate the most demanding tasks. Furthermore, Gateways also interact with the actuators. They control the acting devices based on the outputs from their own computations or based on the instructions that they receive from the Back-End subsystem. In the projects developed by the novice IoT developers, this subsystem typically consisted of single-board computers such as Raspberry Pis.

Back-end groups third-party services APIs, the main application server, and the persistence component. The functionalities provided by the application server

and the persistence component are typically exposed to the Gateways subsystem through RESTful web services. Finally, third-party service APIs are commonly used to interact with the wearable devices belonging to the Sensors subsystem.

Actuators span actuating devices that trigger changes in the physical environment. However, they also encompass push notifications through which end-users are informed about the occurrence of a given event. Acting devices are generally controlled by gateway devices via Bluetooth or Wi-Fi, while push notifications are commonly generated in the Back-End subsystem through the Android and iOS push notifications platforms APIs.

End-user refers to the interfaces with which the End-users are enabled to interact with the IoT system. These interfaces typically consist of mobile and web applications through which user preferences can be configured, Actuators can be activated or deactivated, and Sensors can be monitored and managed.

An important component that was decided not to represent in the generic architecture is **Security**. This choice was mainly made because it was outside the course syllabus. Therefore, survey respondents were not exposed to the issues and possible pain points that could be generated from security-related operations.

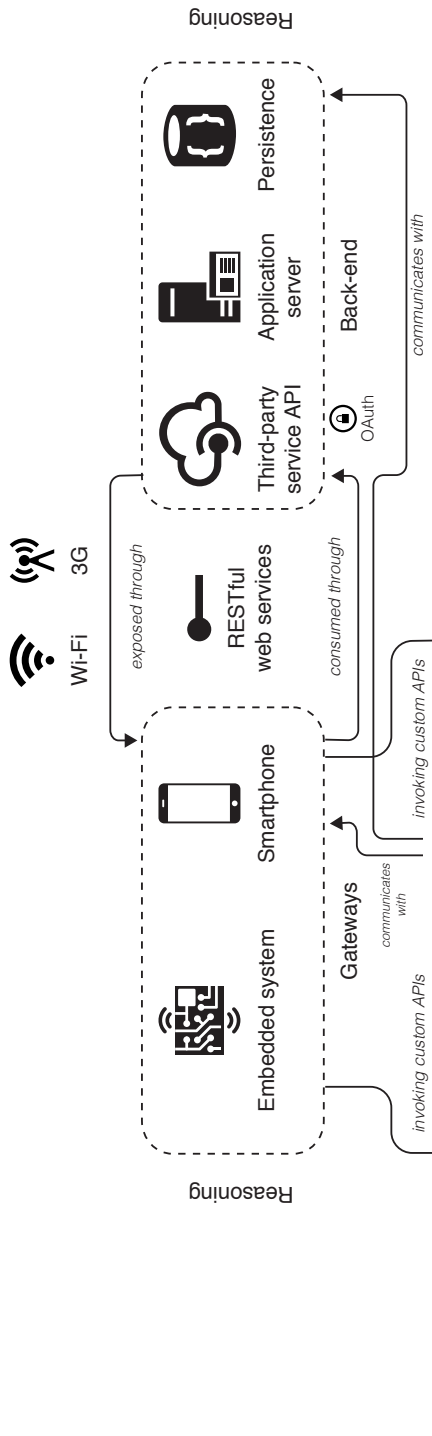


Figure 3.1: Proposed IoT systems reference architecture

However, since the proposed research questions concern the software development perspective of IoT systems, only the subsystems whose implementation and integration with other subsystems relied mainly on software development activities, were considered in this research. These software-intensive subsystems were: End-user, Gateways, and Back-end (we therefore excluded sensors and actuators, that were considered off-the-shelf or external resources). Next, these three subsystems were "decomposed" into a list of software development tasks required for their implementation (e.g.,*Develop a native end-user mobile application*). These tasks, in turn, were decomposed into very punctual, unambiguous sub-tasks (e.g.,*Become familiar with the mobile application platform-specific programming language*).

As Gateways and Back-end subsystems resulted in a quite large number of tasks, these subsystems were split up in two: the first one for the subsystem development tasks, and the second one for the subsystem integration tasks. As a result, the survey was developed starting from the generic architecture and the set of tasks and sub-tasks that emerged from each identified subsystem.

3.2.2 Initial generation of the questionnaire structure and content

The survey was defined in line with the research questions. The subsystems were mapped to a set of sections in the questionnaire, as shown in Table 3.1. Five sections were therefore defined, namely: End-user subsystem (Section A), Gateways subsystem development (Section B), Gateways subsystem integration (Section C), Back-end subsystem development (Section D), and Back-end subsystem integration (Section E). Since the sections belonging to the End-user subsystem were mutually exclusive as they corresponded to the three types of mobile applications, the survey was structured in 5 sections, with 24 tasks, and 67 sub-tasks.

Table 3.1: Survey structure

Subsystems	**Sections**	**# Tasks**	**# Sub-Tasks**
End-user	Section A1: Native mobile application	9	10
	Section A2: Hybrid mobile application	9	10
	Section A3: Web responsive mobile application	9	9
Gateways	Section B: Gateways development	5	10
	Section C: Gateways integration	3	10
Back-end	Section D: Back-end development	4	10
	Section E: Back-end integration	3	8
Total		24	67

As not all respondents participated in the development of every subsystem of the IoT system, the research questions were addressed at the subsystem level of detail. In fact, inside each subsection, they were instructed to skip the rating of the sub-tasks in which they did not personally participate. Therefore, for each section of the questionnaire, respondents were asked to:

- **Rate** the complexity of each sub-task in which they participated according to their difficulty level and the time spent completing them. Two Likert scales ranging from 1 to 5 were included in the questionnaire for each sub-task, as shown in Figure 3.2. This **rating** aims at answering the **RQ3.1**: How complex, in terms of time spent and difficulty, are the software development tasks needed to build an IoT system?

- **Rank** the sub-tasks of the concerned subsystem identifying the ones perceived as the most challenging to complete. In this case, such sub-tasks had to be ranked as the first, second and third most difficult task in the subsystem. In Figure 3.2, the field at the left of the sub-tasks is intended to the rank the three most challenging sub-tasks. This **ranking** aims at answering the **RQ3.2**: Which are the software development tasks that are perceived as the most challenging to complete?

- Assess the **perception** of the respondents about the reasons behind the ranking choice on each subsystem. This perception is captured through an open question, where besides their justification, respondents could also mention any other task that they found complex to achieve, even if it was not in the set of suggested sub-tasks. The qualitative **perceptions** of the participants are intended to answer the **RQ3.3**: Why are these tasks perceived as the most challenging?

RQ3.1 (rate) was measured at the sub-task level, while RQ3.2 (rank) and RQ3.3 (perception) were measured at each section level.

3.2.3 Initial pilot survey

To validate the pertinence and completeness of the resulting survey, a preliminary study [26] was conducted and documented with a small group of participants (6). These participants belonged to the 2016 cohort of the previously mentioned Ambient Intelligence course. In this version of the course 18 projects related to Health and Well-Being were developed. The participants chosen for the pilot survey were the members of two groups whose final projects obtained outstanding grades, and whose implementation relied mainly on software development activities. The first group was composed by 4 students and the second group was composed by 3 students. However, one of the members of the second group was an international

Rank	Section A: End-user	Difficulty					Time spent				
Develop a native end-user mobile application											
	Become familiar with the mobile application platform-specific programming language	1	2	3	4	5	1	2	3	4	5
	Configure the development environment	1	2	3	4	5	1	2	3	4	5
	Develop the models' classes	1	2	3	4	5	1	2	3	4	5
	Develop the controllers' classes	1	2	3	4	5	1	2	3	4	5
	Develop the user interface (views)	1	2	3	4	5	1	2	3	4	5
	Connect the push notification module with the platform notification service	1	2	3	4	5	1	2	3	4	5
	Handle the notifications received in the end-user's smartphone	1	2	3	4	5	1	2	3	4	5

Figure 3.2: Example of a task decomposed in sub-tasks

student who returned to her home university, therefore a total of 6 students were involved in the preliminary study.

The pilot was conducted by inviting the respondents to 2 interview sessions, with one group invited per each session. Each session consisted of three phases: an introduction, the questionnaire, and a discussion. The sessions were conducted by two researchers, and were held in English. In the introduction, one researcher briefly explained the objective of the study, the structure of the questionnaire and the general organization of the session. It was clarified that the questionnaire had to be filled individually from the personal point of view (each participant should respond to those activities in which they were directly involved, only), while the following discussion would involve their evaluation as a group.

The questionnaire was filled out on paper, and as described before, participants were asked to rate the sub-tasks according to their difficulty level and the time spent completing them, rank the three most difficult tasks per each section of the questionnaire, and justify their ranking choice with an open question, where they could also mention any other tasks that were not listed but resulted complex to achieve.

After all participants completed the questionnaire, a final discussion was held to identify, as a group, the most complex and painful tasks. The respondents were free to discuss among themselves, and with the researchers. The completion of the questionnaire took each participant, in average, approximately 30 minutes, while the later discussion about the most painful issues and the feedback about the completeness of the questionnaire took around 20 minutes.

Besides providing some preliminary insights about the most painful issues when developing IoT systems, this study provided valuable feedback regarding the pertinence and completeness of the proposed generic architecture, the identified subsystems, and their tasks and sub-tasks. The participants of this preliminary study

did not suggest any modification to the architecture nor the addition of sub-tasks that the questionnaire could have overlooked.

3.2.4 Survey instrument

After the initial pilot survey, the design of the questionnaire was concluded. The final structure in terms of sections and their tasks is shown in Table 3.2, which also reports the number of sub-tasks defined per each task.

3.2.5 Administration and population

The survey was managed through the Lime Survey [61] platform, and invitations were sent by email to former students of the three cohorts of the Ambient Intelligence course between the years 2014 and 2016. Since some former students of the course were in Erasmus, when possible, the invitations were sent to both institutional and personal email addresses. Moreover, with the aim of motivating the participation, the draw of a Sonos wireless speaker among the people who completed the survey was announced. Recipients were free to participate if they chose and their ratings and opinions would be anonymous. Due to the draw of the wireless speaker, the survey had an explicit closing date (April 23, 2017).

The first invitation was sent on February 8, 2017, and two reminders were sent before the closing date. The number of potential recipients of the survey invitation was 150: 45 of them partially completed the survey (they were not taken into account in the survey results), while 40 completed the whole survey. Therefore, the estimated response rate was approximately 27% (40/150). The platform enabled participants to save partially finished surveys.

To make sure that there was not a substantial difference concerning the characteristics of the survey respondents and the non-respondents former students of the course (*response bias*), I decided to compare their final grades obtained at the end of the course. Table 3.3 presents the main statistics about the grades obtained by former students of the course that did not participate (Non-respondents) in the survey and those who did participate (Respondents)[1]. As it may be observed, the grades do not differ greatly between the two groups.

Another possible bias factor would be an item bias: some respondents might have rushed through several sections of the questionnaire intentionally to quickly complete the survey and participate in the draw. However, as shown in Table 3.4, the time spent completing the online survey was generally consistent with the number of subsystems they worked on. On average, respondents who worked on one

[1]The survey was anonymous, and the responses could not be associated to each student. However, the list of students who responded was available.

Table 3.2: Subsystems and tasks in the questionnaire

Section A: End-user subsystem	**# Sub-Tasks**
Develop a native end-user mobile application	7
Develop a hybrid end-user mobile application	8
Develop a web responsive end-user mobile application	6
Develop the integration between the end-user application and the gateways [computation node, smartphone]	2
Deploy the end-user mobile application into the smartphone	1
Section B: Gateways subsystem (Development)	
Configure the development environment	2
Develop the business logic of the gateway device [computation node, smartphone] application	2
Configure the OAuth authentication between the gateway device [computation node, smartphone] and third-party services APIs	3
Develop the module for generating notifications to be displayed on the end-user application	2
Deploy the software into the gateway devices [computation node, smartphone]	1
Section C: Gateways subsystem (Integration)	
Develop the integration between the gateway device [computation node, smartphone] and the sensors [wearable devices, static sensors]	4
Develop the integration between the gateway device [computation node, smartphone] and the back-end [third-party service API, application server, persistence] by consuming these last ones' custom APIs	4
Develop the integration between the gateway device and the actuators responsible for changes in environment	2
Section D: Back-end subsystem (Development)	
Configure the development environment	2
Design and develop the persistence component	3
Develop the business logic on the application server	2
Develop the RESTful web services	3
Section E: Back-end subsystem (Integration)	
Develop the integration between the application server and third-party services	2
Configure OAuth between the application server and third-party services	3
Develop the integration between the application server and the persistence component	3

Table 3.3: Comparison between the grades (over a scale of 18 to 30, or 31 for a grade with honors) of the former students and the respondents

	Non-respondents (N=110)	**Respondents (N=40)**
minimum	18.0	19.0
maximum	31.0	31.0
mean	26.3	28.3
SD	4.1	3.1

subsystem took 16 minutes, two subsystems 21 minutes, and three subsystems 25 minutes. Moreover, these data show that 16% of the respondents were involved in the development all the subsystems.

Table 3.4: Time spent completing the online survey by number of subsystems on which they answered questions

	1 Subsystem	**2 Subsystems**	**3 Subsystems**
minimum	0:04:53	0:08:33	0:19:29
maximum	0:42:26	0:44:44	0:46:31
mean	0:16:37	0:21:27	0:25:56
respondents percentage	50.0%	34.0%	16.0%

As the respondents were asked to answer the survey just for the sub-tasks that they completed in the development of their IoT systems, Table 3.5 shows the percentage of completion for each subsystem. The subsystem with a higher average percentage of completeness was the End-user subsystem (79%), followed by the Back-end (75%), and finally the Gateways (60%).

Table 3.5: Percentage of sub-tasks rated on each subsystem

	End-user	**Gateways**	**Back end**
minimum	44.4%	15.0%	37.5%
maximum	100%	100%	100.0%
mean	78.9%	59.7%	75.2%
respondents percentage	63.2%	50.0%	50.0%

3.3 Results

This section presents the results from the survey according to the proposed research questions. These results are described as detailed as possible in terms of particular software development tasks. Concretely, Subsection 3.3.1 provides a brief description of the demographics of the respondents of the survey, while Subsection 3.3.2 introduces the outcomes that emerged from the survey and their connection to the research questions. Subsection 3.3.3 concerns the first research question, Subsection 3.3.4 regards the second research question, and Subsection 3.3.5 addresses the third research question.

3.3.1 Demographics

At the beginning of the survey questionnaire, several questions were included to characterize the demographics of the respondents. As shown in Table 3.6, a vast majority of the respondents were male (87.5%), and their ages were mainly from 22 to 24 years old (80%), which is consistent with the student population. Furthermore, a marked majority of respondents belonged to Computer Engineering (70%), followed by those that belonged to Electronic Engineering (20% of them), as illustrated in Figure 3.3.

At the beginning of the course, students were asked to complete an online questionnaire concerning their prior knowledge on a set of technical skills and programming languages. Specifically, they graded such skills and programming languages on a 5-point Likert scale. This grading is reported in Table 3.7, where the ratings in the Likert scale are categorized into low (1 and 2), medium (3), and high (4 and 5). It is observed that the prior knowledge declared by the students of the Ambient Intelligence course was low in most of the topics and programming languages. The only exceptions were programming (in general) and the C language, since nearly all students had taken a basic programming course. Similarly, as briefly mentioned in the Introduction, the respondents of the survey are considered to be, to some extent, representative of novice professionals in the context of IoT systems development. This idea is supported based on the observation made by Salman et al. [86], according to which "in an academic setting, we can find students who already possess industrial experience or in a field experiment, we can face novice professionals with regard to a particular technology".

Moreover, when the respondents were developing their course projects, they were not constrained or induced to use a specific Integrated Development Environment (IDE). In particular, regarding web development (typically in HTML, CSS, and JavaScript) and the back-end development (commonly in Python and Java), respondents had the freedom to choose the code editor that they liked the most. However, when developing mobile applications, respondents were inevitably led to the use of the development tool provided by the corresponding mobile operating

system (Android Studio, when implementing Android applications, and XCode, when implementing iOS applications). Nevertheless, all the respondents (belonging to different groups) implemented the code mostly from scratch, without code generation tools or tools that could have hidden the inherent complexity of implementing the IoT system. For this reason, the development tools are not a factor of disparity; they are not considered to affect the results obtained by respondents from different groups.

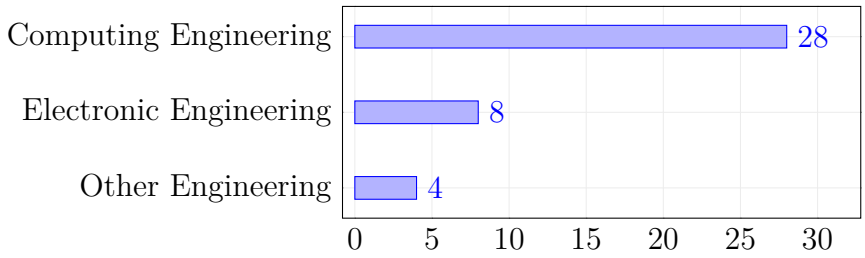
Table 3.6: Age and gender of the respondents

	22-24	**25-27**	**>28**	**Total**
Female	5	1	-	5
Male	27	4	3	33
Total	32	5	3	**40**

Table 3.7: Technical skills of the AmI students before taking the course

Topic	**Low**	**Average**	**High**
Programming (in general)	16.97%	**44.85%**	38.18%
Web architectures	**69.09%**	19.39%	11.52%
Mobile development	**86.06%**	9.09%	4.85%
Source control management	**90.91%**	4.85%	4.24%
Software requirements specification	**75.15%**	15.76%	9.09%
Python	**92.12%**	2.42%	5.45%
HTML/CSS	**81.21%**	9.09%	9.70%
JavaScript	**89.09%**	5.45%	5.45%
Java	**79.39%**	9.09%	11.52%
C	12.12%	27.27%	**60.61%**

Figure 3.3: Bachelor degree of the survey respondents when attending the course



3.3.2 Research questions

Three types of outcomes emerged for each of the subsystems in the survey:

1. The **rating** of each sub-task complexity in terms of its difficulty level and completion time (i.e., to answer RQ3.1).

2. The set of sub-tasks that were **ranked** as the most challenging (i.e., to answer RQ3.2).

3. The **perception** of each respondent about the most challenging tasks (i.e., to answer RQ3.3).

In this section, the three questions will be addressed for each subsystem. Since the **rating** of the difficulty level and time spent on each sub-task was captured on an ordinal scale, the results from this outcome were analyzed through the median and the correlation between these variables. As shown in Tables 3.8 to 3.10, almost all the development sub-tasks yielded positive correlations between the difficulty level and the time spent. Next, for analyzing the **ranking** of the most complex sub-tasks, they were prioritized according to the number of times they were included in the ranking and their position. To do this prioritization, a weighted sum was calculated for each sub-task and the three sub-tasks with the highest weights are presented below. Lastly, the **perception** about the most complex tasks was analyzed through the comments of the survey respondents, and three main categories were identified: *learning curve* issues, *integration between subsystems* issues, and *configuration and deployment* issues. At the end of this section, will be reported, for each of these categories, some of the most representative comments made by the respondents when they were asked to justify their ranking choice.

3.3.3 RQ3.1. Rating of the sub-tasks

Tables 3.8 to 3.10 present the development sub-tasks of each subsystem, along with two box plot diagrams illustrating the ratings for the difficulty level and time spent. Moreover, next to these diagrams are reported the correlation between the ratings. When the p-value was less than 0.05, the correlation is flagged with a *, meaning that the correlation of the concerned sub-task is statistically significant.

Section A: End-user subsystem

End-user native mobile application (Section A1) The sub-task of *Becoming familiar with the mobile application platform-specific programming language (eunat-1)*, was rated as difficult and time-spending. Concretely, these programming languages are: Java, when developing Android applications, or Swift, when developing iOS applications. On the contrary, although also statistically significant, the

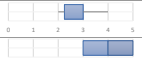
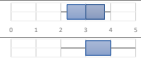
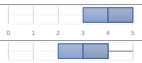
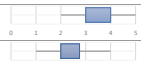
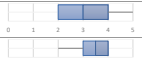
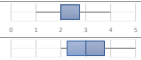
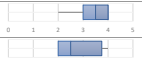
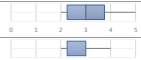
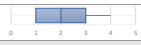
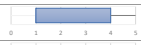
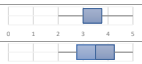
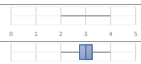
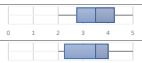
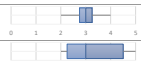
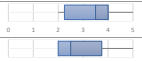
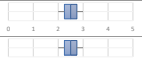
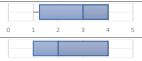
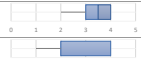
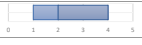
Section A1: End-user native mobile application		Difficulty	Time spent	Corr.
Develop a native end-user mobile application				
eu-nat-1	Become familiar with the mobile application platform-specific programming language			0.91*
eu-nat-2	Configure the development environment			0.93*
eu-nat-3	Develop the models' classes			0.59
eu-nat-4	Develop the controllers' classes			0.88
eu-nat-5	Develop the user interface (views)			-0.19
eu-nat-6	Connect the push notification module with the platform notification service			0.77
eu-nat-7	Handle the notifications received in the end-user's smartphone			0.77
Develop the integration between the end-user application and the gateways				
eu-nat-8	Implement the HTTP asynchronous requests through the RESTful web services exposed by the gateways			0.82
eu-nat-9	Parse and handle the JSON- or XML-formatted response			0.41
Deploy the end-user mobile application into the smartphone				
eu-nat-10	Package the application into a compatible format that might be deployed on the smartphone			0.94*
Section A2: End-user hybrid mobile application		**Difficulty**	**Time spent**	**Corr.**
Develop a hybrid end-user mobile application				
eu-hyb-1	Become familiar with the scripting programming languages			-0.08
eu-hyb-2	Configure the development environment			0.92*
eu-hyb-3	Incorporate into the project all the required plugins on their corresponding versions			0.61
eu-hyb-4	Develop the controllers			0.32
eu-hyb-5	Develop the user interface through HTML and CSS files (views)			0.40
eu-hyb-6	Develop the user interaction through JavaScript files (views)			0.80
eu-hyb-7	Connect the push notification module with the platform notification service			-
eu-hyb-8	Handle the notifications received in the end-user's smartphone			-
Develop the integration between the end-user application and the gateways				
eu-hyb-8	Implement the HTTP asynchronous requests through the RESTful web services exposed by the gateways			0.43
eu-hyb-9	Parse and handle the JSON- or XML-formatted response			0.82
Deploy the end-user mobile application into the smartphone				
eu-hyb-10	Package the application into a compatible format that might be deployed on the smartphone			0.94
Section A3: End-user web responsive mobile application		**Difficulty**	**Time spent**	**Corr.**
Develop a web responsive end-user mobile application				
eu-web-1	Become familiar with the scripting programming languages			0.69*
eu-web-2	Configure the development environment			0.87*
eu-web-3	Develop the controllers			0.70*
eu-web-4	Develop the user interface through HTML and CSS files (views)			0.18
eu-web-5	Develop the user interaction through JavaScript files (views)			0.47
eu-web-6	Deploy the application on the web			1.00
Develop the integration between the end-user application and the gateways				
eu-web-7	Implement the HTTP asynchronous requests through the RESTful web services exposed by the gateways			0.88*
eu-web-8	Parse and handle the JSON- or XML-formatted response			0.75*
Deploy the end-user mobile application into the smartphone				
eu-web-9	Package the application into a compatible format that might be deployed on the smartphone			0.96*

Table 3.8: End-user ratings

sub-tasks of *Configuring the development environment (eu-nat-2)* and *Packaging the application into a compatible format that might be deployed on the smartphone*

(eu-nat-10) were rated as not difficult and not time-spending, plausibly because the "official" IDEs provide the developers enough commands and visual interfaces to do so intuitively. *Developing the user interface (eu-nat-5)* yields a negative correlation, suggesting that it was considered to be more difficult than time spending. Probably because once the first views are developed, the learning curve is overtaken and the process becomes quicker. However, the magnitude (-0.19) and statistical significance of this correlation are weak. *Developing the controller's classes (eu-nat4)* was rated as a very difficult sub-task, and significantly time spending, with a high correlation (0.88) even if not statistically significant, according to the p-value. Effectively, the development of the controllers is critical since they are the bound between the models and the views, and they manage the interaction with external data sources. *Connecting the push notification module with the platform notification service (eu-nat-6)* was also rated as a complex sub-task. This can be due to the parametrization and the functions that have to be properly implemented to generate and manage the notifications.

End-user hybrid mobile application (Section A2) *Configuring the development environment (eu-hyb-2)* was the only sub-task whose correlation between difficulty and time spent was statistically significant. Notwithstanding, the ratings of this task were scattered in both variables, most of them ranging from 1 to 4. Therefore, based on these numbers, it is not possible to determine accurately if this sub-task was perceived as complex or not. Unlike the development of the End-user native application, *Becoming familiar with the scripting programming languages (eu-hyb-1)* is not perceived as complex, and has a very weak, and negative, correlation between difficulty and time spent (-0.08). These results would suggest that an advantage for novice programmers of hybrid mobile applications over the native ones, is the use of scripting languages, which are more common and widespread. However, *Developing the user interface through HTML and CSS files (eu-hyb-5)* appears to be more difficult and time spending in hybrid mobile applications than in the native applications. Since the native applications IDEs are targeted at a specific mobile operating system, they provide a better consistency between the design, development, and deployment of the user interface. Finally, in the sub-task of *Connecting the push notification module with the platform service (eu-hyb-7)*, all the respondents rated the time spent with 2, as well as in *Handle the notifications received in the end-user's smartphone (eu-hyb-8)*, where all the respondents rated the difficulty as 3. In both cases the low number of responses did not allow the correlation to be computed.

End-user responsive mobile application (Section A3) The set of sub-tasks with a significant correlation were rated as not particularly complex. *Parsing and handling the JSON- or XML-formatted responses (eu-web-8)* had exactly the same ratings in the difficulty and time spent variables as well as *Packaging the application into a compatible format that might be deployed on the smartphone (eu-web-9)*. The difficulty when *Developing the controllers (eu-web-3)* had the same

value (3) for the first quartile, the median, and the third quartile. *Implement the HTTP asynchronous requests through the RESTful web services exposed by the gateways (eu-web-7)* was rated as considerably difficult and time spending (median = 3, in both variables). This sub-task might be more complex in this kind of mobile applications due to the lack of libraries or frameworks to manage the connection and the HTTP requests and responses, such as the ones in the native or hybrid mobile applications IDEs. Naturally, the *Deployment of the application on the web (eu-web-6)* was rated as easy and quick. The *Development of the user interface through HTML and CSS files (eu-web-4)* was rated as easy but considerably time spending. Probably all the programmers were familiar with HTML and CSS, or at least could easily become skilled in them. However, the absence of a tool to graphically compose and link the views could affect negatively the time spent.

Sections B and C: Gateways subsystem

Gateways development (Section B) in almost all the sub-tasks the correlation between difficulty and time spent was statistically significant. *Configuring the development environment (gw-dev-1* and *gw-dev-2)* was rated as easy and quick. The ratings for *Developing the business logic of the gateway device application (gw-dev-3* and *gw-dev-4)* were mainly scattered from 2 to 4, meaning that while not extremely difficult, neither they were extremely easy. In the context of the IoT course projects, the concept of 'business logic' basically refers to the way in which the data gathered from the sensors will be used to accomplish the overall system functional requirements. *Setting up the parameters needed to establish the connection with third-party services APIs using OAuth (gw-dev-5)* was rated as very time spending while moderately difficult. In fact, even if this sub-task doesn't require a significant programming effort, it requires a good conceptual and technical understanding of OAuth and the registration of the application in the third-party service platform. Likewise, *Developing the methods or functions required to establish the connection with third-party services APIs using OAuth (gw-dev-7)*, was rated as a difficult and time spending sub-task. It demands to consult several documentation sources and devote a significant amount of time ensuring a successful connection. *Generating the notifications by invoking the platform notification service APIs (gw-dev-9)* had exactly the same ratings in the difficulty and time spent variables.

Gateways integration (Section C) the correlation of all the sub-tasks regarding the *Integration between the gateway device and the sensors (gw-int-1 to gw-int-4)* were statistically significant. In particular, *Developing the methods or functions required to establish the Bluetooth connection with the sensors (gw-int-2)* was rated as easy and quick, and the *Development of the methods or functions to obtain data from the sensors (gw-int-3)* was rated as time-spending. The difficulty in the *Development of a component for receiving real-time streaming data coming*

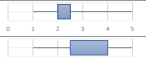
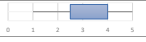
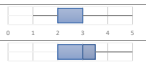
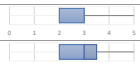
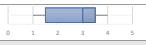
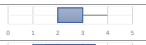
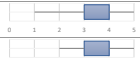
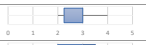
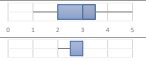
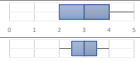
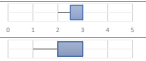
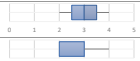
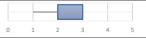
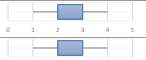
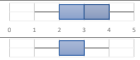
Section B: Gateways development		Difficulty	Time spent	Corr.
Configure the development environment				
gw-dev-1	Install and deploy the operating system			0.64*
gw-dev-2	Install the libraries and dependencies needed to develop on the gateway's controller			0.74*
Develop the business logic of the gateway device application				
gw-dev-3	Define and implement the required set of models			0.82*
gw-dev-4	Develop the methods or functions where the business logic is implemented			0.80*
Configure the OAuth authentication between the gateway device and third-party services APIs				
gw-dev-5	Set up the parameters needed to establish the connection			0.43
gw-dev-6	Install the required set of libraries			0.88*
gw-dev-7	Develop the methods or functions required to establish the connection			0.82*
Develop the module for generating notifications to be displayed on the end-user application				
gw-dev-8	Set up the parameters needed to establish the connection with the platform notification service			0.52
gw-dev-9	Generate the notifications by invoking the platform notification service APIs			0.80*
Deploy the software into the gateway devices				
gw-dev-10	Package the application into a compatible format that might be deployed on the gateway			0.87*
Section C: Gateways integration		**Difficulty**	**Time spent**	**Corr.**
Develop the integration between the gateway device and the sensors				
gw-int-1	Develop the methods or functions required to establish the Wi-Fi connection with the sensors			0.85*
gw-int-2	Develop the methods or functions required to establish the Bluetooth connection with the sensors			0.90*
gw-int-3	Develop the methods or functions required to obtain data from the sensors			0.78*
gw-int-4	Develop a component for receiving real-time streaming data coming from the sensors			0.90*
Develop the integration between the gateway device and the back-end by consuming these last ones' custom APIs				
gw-int-5	Implement the HTTP asynchronous requests through the RESTful web services exposed by third-party services APIs			0.17
gw-int-6	Parse and handle the JSON- or XML-formatted response obtained from third-party services APIs			0.78*
gw-int-7	Implement the HTTP asynchronous requests through the RESTful web services exposed by the application server			0.28
gw-int-8	Parse and handle the JSON- or XML-formatted response obtained from the application server			0.08
Develop the integration between the gateway device and the actuators responsible for changes in environment				
gw-int-9	Develop the methods or functions required to establish the connection			0.88*
gw-int-10	Develop the methods or functions required to handle the actuators behaviour			0.54

Table 3.9: Gateways ratings

from the sensors (gw-int-4) had the first quartile, the median, and the third quartile rated as 3. The rest of the sub-tasks in this subsection did not exhibit a clear trend concerning their low or high complexity.

Sections D and E: Back-end subsystem

Back-end development (Section D) The sub-tasks corresponding to the *Configuration of the development environment (be-dev-1* and *be-dev-2)*, and the *Design and development of the persistence component (be-dev-3* and *be-dev-4)* were not rated as complex. On the contrary, the *Development of the business logic on the application server* was rated as difficult and time spending. Especially the *Development of the methods or functions where the business logic is implemented*

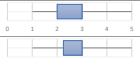
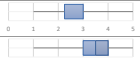
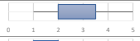
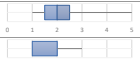
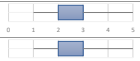
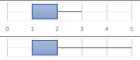
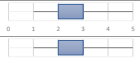
Section D: Back-end development		Difficulty	Time spent	Corr.
Configure the development environment				
be-dev-1	Install and deploy the application server			0.68*
be-dev-2	Install the libraries and dependencies needed for the application server development			0.89*
Design and develop the persistence component				
be-dev-3	Design the entity-relationship model or the corresponding data model, if NoSQL is used			0.80*
be-dev-4	Install the database server			0.92*
be-dev-5	Deploy the database with the corresponding data model			0.49
Develop the business logic on the application server				
be-dev-6	Define the required set of models			0.88*
be-dev-7	Develop the methods or functions where the business logic is implemented			0.86*
Develop the RESTful web services				
be-dev-8	Define the HTTP methods along with their URI and associated operation			0.75*
be-dev-9	Set up the framework required to implement the RESTful web services			0.66*
be-dev-10	Implement the mapping between the business logic models and the exposed RESTful web services			0.41
Section E: Back-end integration		**Difficulty**	**Time spent**	**Corr.**
Develop the integration between the application server and third-party services				
be-int-1	Implement the HTTP asynchronous requests through the RESTful web services exposed by third-party service APIs			0.81*
be-int-2	Parse and handle the JSON- or XML-formatted response			0.74*
Configure the OAuth authentication between the application server and third-party services				
be-int-3	Set up the parameters needed to establish the connection			0.99*
be-int-4	Install the required set of libraries			0.77*
be-int-5	Develop the methods or functions required to establish the connection			0.90*
Develop the integration between the application server and the persistence component				
be-int-6	Set up the connection between the application server and the database			0.80*
be-int-7	Implement the queries to be performed over the database			0.70*
be-int-8	Parse and handle the database response			0.82*

Table 3.10: Back-end ratings

(be-dev-7). Naturally, the complexity of this sub-task has to do with the fact that each group had to program its business logic methods from scratch, and without the guidance of previous implementations. The sub-tasks concerning the *Development of the RESTful web services (be-dev-8* to *be-dev-10)* were not rated as difficult but as moderately time spending.

Back-end integration (Section E) *Parsing and handling the JSON- or XML-formatted response (be-int-2)* was rated as easy and quick. When configuring the OAuth authentication between the application server and third-party services, the *Installation of the required set of libraries (be-int-4)* was rated as easy and quick. Moreover, the *set up of the parameters (be-int-3) and the development of the methods or functions to establish the connection (be-int-5)*, do not exhibit a clear trend about how complex they resulted to the respondents. Lastly, all the sub-tasks concerning the *Development of the integration between the application server and the persistence component (be-int-6, be-int-7, and be-int-8)* were rated as easy and not time spending.

Summary of RQ3.1

RQ3.1: How complex, in terms of time spent and difficulty, are the software development tasks needed to build an IoT system?

Almost all the sub-tasks hold a positive correlation between the two variables that were used to measure the complexity. It means that all the sub-tasks rated as significantly difficult were also rated as considerably time spending. By comparing the ratings of the sub-tasks and the correlation between their two variables, I could preliminarily identify for each subsystem those sub-tasks that stand out as complex. Hereafter is presented a summary of the main findings for each subsection of the questionnaire.

In the **End-user native mobile application**, becoming familiar with the platform-specific programming language (eu-nat-1), and developing the application controllers (eu-nat-4) were rated as difficult and time-spending sub-tasks. In the **End-user hybrid mobile application**, the rating of the sub-tasks and their statistical significance did not allow to identify a clear tendency about the correlation between the difficulty and the time spent. In the **End-user responsive mobile application**, the implementation of the HTTP asynchronous requests through the RESTful web services exposed by the gateways (eu-web-7) was rated as considerably difficult and time spending. On the contrary, the development of the user interface through HTML and CSS files (eu-web-4) was rated as easy but time spending. In the **Gateways development**, implementing the methods or functions to establish the connection with third-party services APIs using OAuth (gw-dev-7), was rated as difficult and time-spending. In the **Gateways integration**, the correlation of the sub-tasks concerning the integration between the gateways and the sensors (gw-int-1 to gw-int-4) were statistically significant. While the methods or functions to establish the connection with the sensors was rated as easy and quick (gw-int-2), the methods or functions to gather information from those sensors were rated as time-spending (gw-int-3). In the **Back-end development** the implementation of the business logic on the application server was rated as difficult and time spending (be-dev-7). On the contrary, the sub-tasks concerning the development of the RESTful web services were not rated as difficult but as moderately time spending (be-dev-8 to be-dev-10). In the **Back-end integration** subsystem no sub-task was rated as particularly difficult or time spending.

3.3.4 RQ3.2. Ranking of the sub-tasks

Table 3.11 presents, for each subsystem, the three sub-tasks that the respondents placed in the first position of the ranking they were asked to do in the survey, as the most complex ones. In the End-user subsystem, these sub-tasks are listed depending on the kind of End-user mobile application developed whether it was a native, hybrid or web-responsive mobile application. Next to each sub-task there

is a triplet of numbers representing the number of times in which the concerned sub-task was ranked in the first, second, and third place, respectively. As expected, this ranking matches with the **rating** of the sub-tasks in terms of difficulty level and time spent.

According to the sub-tasks ranked as the most complex, it can be observed that despite the kind of End-user mobile application implemented (native, hybrid or web-responsive), the development of the user interface was perceived as complex (eu-nat-5, eu-hyb-5, eu-web-4). This observation is somehow surprising, particularly concerning native mobile applications, where one may presume that the IDE would ease the design of the user interface views and their connection with the business logic of the application. Moreover, the configuration of the development environment was perceived as complex both in the native mobile applications as well as in the web responsive mobile application (eu-nat-2, eu-web-2). Once again, it is striking that even with specialized IDEs as the ones used to develop native mobile applications, the configuration of the development environment was ranked as complex. Finally, concerning the End-user subsystem, the development of the controllers (which can be understood as the binding between the views and the business logic), was ranked as one of the most complex sub-task both in the hybrid mobile applications and in the web-responsive mobile applications.

In the Gateways development and the Gateways integration subsystems there is a clear correspondence between the *development of the methods or functions to establish the connection between the gateway device and the third party services APIs using OAuth*, in the Gateways development (gw-dev-7), and the *parsing and handling the JSON or XML-formatted response obtained from the third-party services APIs*, in the Gateways integration (gw-int-6).

Similarly, in the Back-end subsystem, there is a clear mapping between the *deployment of the database with the corresponding data model*, in the Back-end development (be-dev-5), and the *parsing and handling of the database response in the application server*, in the Back-end integration (be-int-8).

However, between Gateways and Back-end subsystems, there are also some coincidences. For instance, the *development of the functions or methods where the business logic is implemented* was ranked in both subsystems as a complex sub-task (gw-dev-4 and be-dev-7). Furthermore, there is another correspondence between the *implementation of the HTTP asynchronous request through the RESTful web services exposed by the application server*, in the Gateways integration (gw-int-7), and the *implementation of the mapping between the business logic models and the exposed RESTful web services*, in the Back-end development (be-dev-10).

Table 3.11: Sub-tasks ranked as the most complex (ranking 1st-2nd-3rd)

Section A1: End-user native mobile application		
eu-nat-1	Become familiar with the mobile application platform-specific programming language	3-3-0
eu-nat-2	Configure the development environment	1-1-1
eu-nat-5	Develop the user interface (views)	1-0-2
Section A2: End-user hybrid mobile application		
eu-hyb-7	Connect the push notification module with the platform notification service	2-0-0
eu-hyb-5	Develop the user interface through HTML and CSS files (views)	1-1-0
eu-hyb-4	Develop the controllers	1-0-0
Section A3: End-user web responsive mobile application		
eu-web-2	Configure the development environment	3-0-0
eu-web-3	Develop the controllers	2-1-1
eu-web-4	Develop the user interface through HTML and CSS files (views)	2-0-1
Section B: Gateways development		
gw-dev-4	Develop the methods or functions where the business logic is implemented	5-2-1
gw-dev-7	Develop the methods or functions to establish the connection (between the gateway device and third-party services APIs using OAuth)	5-0-1
gw-dev-5	Set up the OAuth parameters needed to establish the connection with the third-party services APIs	2-3-1
Section C: Gateways integration		
gw-int-3	Develop the methods or functions required to obtain data from the sensors	3-2-1
gw-int-6	Parse and handle the JSON- or XML-formatted response obtained from the third-party services APIs	3-1-1
gw-int-7	Implement the HTTP asynchronous requests through the RESTful web services exposed by the application server	2-3-1
Section D: Back-end development		
be-dev-5	Deploy the database with the corresponding data model	2-4-1
be-dev-7	Develop the methods or functions where the business logic of the application server is implemented	2-3-1
be-dev-10	Implement the mapping between the business logic models and the exposed RESTful web services	2-2-2
Section E: Back-end integration		
be-int-1	Implement the HTTP asynchronous requests through the RESTful web services exposed by third-party service APIs	9-0-0
be-int-8	Parse and handle the database response (in the application server)	2-3-3
be-int-3	From the server application, set up the OAuth parameters needed to establish the connection with the third-party services	4-0-3

Summary of RQ3.2

RQ3.2: Which are the software development tasks that are perceived as the most complex to complete?

The development tasks that are perceived as the most complex regard aspects

that are common to various subsystems, as might be the development of user interfaces, the configuration of the development environments, and the development of the business logic. Some other aspects, however, are split across various subsystems. Such is the case of the integration between the Gateway devices and the third-party services, the implementation and integration with the persistence component, and the implementation, exposure, and consumption of custom RESTful web services.

Due to the above, besides determining the most complex sub-tasks on each subsystem, the ranking section of the survey helped to identify, firstly, the set of sub-tasks that are common to various subsystems, and secondly, a set of sub-tasks that complement each other in the development of some portions of the IoT system.

In the first set of sub-tasks there are:

- Sub-tasks concerning the development of the user interfaces were ranked as complex regardless the kind of End-User mobile application (eu-nat-5, eu-hyb-5, and eu-web-4).
- The configuration of the development environment was also ranked as complex both in the native mobile application and in the web responsive mobile application.
- The development of the controllers (binding between models and views in the Model-View-Controller [MVC] architecture) was ranked as complex in the hybrid mobile applications and in the web responsive mobile applications.

In the second category there are:

- The development of the methods or functions to establish the connection between the Gateway device and the third party service APIs using OAuth (gw-dev-7), and the parsing and handling of the response obtained from the third party service APIs, in the Gateways Integration subsystem (gw-int-6).
- The design, implementation, and integration of the persistence component, which spans across the Back-end development (be-dev-5) and the Back-end integration (be-int-8).
- The design, implementation and later consumption of the custom RESTful web services. Their implementation and mapping between the business logic and the exposed RESTful web services belong to the Back-end Development subsystem (be-dev-10), while the consumption of those services belongs to the Gateways Integration (gw-int-7).

3.3.5 RQ3.3. Qualitative perception of the survey respondents

Below are presented and analyzed the comments given by the respondents when asked about why did they perceive certain sub-tasks as the most challenging ones. Such comments were analyzed through inductive thematic analysis, that involved two researchers and followed the six-phase framework proposed by Braun and Clarke's [13]. The first step consisted in becoming familiar with the comments of the respondents and was carried out by the two researchers. Secondly, the first researcher categorized the material at the sentence level and generated the initial codes. The second researcher, for his part, discussed and validated them. This discussion was in person and using hard copies of the respondents' comments. Since an open code approach was used, there were not preset codes. Instead, the codes were being developed and modified while advancing through the coding process. Thirdly, the first researcher identified produced a set of themes with their corresponding codes; initially, 18 open codes were used, later grouped into three broader themes. Further on, in the fourth step, the second researcher validated and approved the proposed themes under the rationale that they were supported by the previously generated codes. In this manner, in the fifth step, the themes were jointly analyzed, and in the last step, the researchers' observations around these themes were documented. The thematic analysis revealed three key themes, based also on the commonalities of the comments across all the subsystems: *Learning curve issues*, *Integration between subsystems issues*, and *Configuration and deployment issues*.

Learning curve issues

In the **End-user subsystem**, comments about the Learning curve issues, highlight that most of the survey respondents were developing for the first time a mobile application. Those groups that decided to implement a native End-user mobile application, faced the challenge of learning a new programming language and becoming familiar with the concerned development environment. Respondents expressed: *"I knew almost nothing on Android when I started developing the app, I had to learn everything from scratch"*, *"Becoming familiar with a new programming language in a very little time is very difficult"*. *"I spent a lot of time researching how to complete everything and make it work"*.

In the **Gateways subsystem development**, Learning curve issues concerned how to handle the data gathered from the End-user mobile application, as well as from the sensing devices: *"We worked with the Global Positioning System (GPS), so we had to do some research about playing with coordinates and GPS accuracy."* Moreover, since the communication with End-user and Back-end subsystems is typically achieved through RESTful web services, the understanding of these services

(both the ones that had to be implemented as well as the external ones that had to be consumed), was perceived as challenging by some respondents: *"I had never heard about APIs, and there were tasks in which it was required to work with them."*

Similarly, the **Gateways subsystem integration** required the knowledge about how to adequately implement the *web services* so that, through HTTP requests coming from the other subsystems, the gateway is directed to perform some given business logic function. *"I was a very beginner with no background in 'Implement the HTTP asynchronous requests through the RESTful web services exposed by the third-party services APIs' and 'Parse and handle the JSON- or XML-formatted response obtained from the third-party services APIs'."* Likewise, managing the integration through other transmission protocols was perceived as challenging due to the lack of adequate documentation: *"Bluetooth documentation for Android wasn't clear, and there were not enough examples of how to use it"*, *"There was no documentation for some smart home sensors, or sometimes we found very poorly written documentation"*.

Once again, as occurred in the Gateways subsystem, in the **Back-end subsystem development**, the RESTful web services concept was also challenging to apprehend. *"It took me some time mostly because of the scarce knowledge of Flask, but as I figured things out it all got easier ('Set up the framework required to implement the RESTful web services' and 'Develop the methods or functions where the business logic is implemented')."* In the same way respondents expressed: *"I had to study a lot of things to understand how to implement my logical function in HTTP"*, *"It took some research to understand how to implement it ('Define the HTTP methods along with their URI and associated operation')."* Finally, among the **Back-end subsystem integration**, the learning curve issues concerned how to deal with the data exchanged with the back-end: *"Needed to learn how to 'Parse and handle the database response' properly."*

Integration between subsystems issues

As noted before, across the three types of mobile applications, the integration between the **End-user subsystem** and the Gateways subsystem was among the most challenging issues. Respondents commented *"Interaction between the server and the mobile app was quite difficult for us because we decided to manage it in a 'custom' way."* The integration between the End-user subsystem and the third-party services was particularly challenging: *"We had several problems interfacing the Fitbit APIs with our application"*.

The most painful integration among the **Gateways subsystem development**, according to the respondent's comments, regards the OAuth 2.0 authentication. In fact, this authentication protocol consists of a flow, with a set of roles (*resource owner, resource server, client, and authorization server*) interacting across various steps (*authorization request, access token request, and protected resource request*),

and exchanging several resources (*authorization grant, access token, refresh token, redirect URI [Uniform Resource Identifier]*). Respondents commented *"OAuth 2.0 authentication, to gain access to the Fitbit API, was a mess. There were hardly any tutorials for the method, and it took a lot of time to figure it out"* and *"Authentication is a nightmare."* Moreover, concerning the integration that did not required OAuth, the main reason given by the respondents was the lack of experience: *"I had no experience on how to connect the devices."*

When working on the **Gateways subsystem integration**, dealing with the data coming from the back-end resulted challenging, particularly when receiving streaming data coming in real-time: *"While receiving generic data was easily done, when we stepped up to the real-time stream we spent an enormous amount of time just to figure out how to access it and then how to handle it." "Establishing the connection between front and Back-end is not an easy task! I'd never worked with JSON."*

The integration issues in the **Back-end subsystem development** regarded the communication with the Persistence component, the Gateways, and even the Sensing devices, for those groups that decided to communicate the Sensors subsystem directly with the Back-end subsystem. Finding out how to handle incoming data into the business logic that is implemented in the Application server is perceived as the most challenging issue. Concerning the managing of Persistence component, respondents commented: *"It took a lot of time to develop an error-free function in Python to manage the SQLite database."* When dealing with the data gathered directly from the sensors, respondents commented: *"It was hard to make things run together (i.e. retrieve information from the sensors without stopping the web-application)."* Finally, when incorporating the incoming data into the business logic of the Application server, respondents commented: *"It was necessary to implement a set of models that were compatible with hardware (Arduino) and the web and application server (Flask). The link between these two architectures was not so easy to design."*

According to the respondents comments, the most challenging issues in the **Back-end subsystem integration** had to do with integrating third-party APIs: *"I spent a lot of time because it was the first time I dealt with it ('Implement the HTTP asynchronous requests through the RESTful web services exposed by the third-party service APIs'). I had to change my implementation several times due to the limitations of the commercial third-party APIs. Likewise, it took a while to understand how to use them to achieve our goals. In the final implementation, we used three external APIs'."* As already pointed out in the Gateways subsystem issues, third-party APIs have their own specific protocols, formats, and authentication mechanisms. These specifics imply a higher level of complexity for the novices when integrating third-party services in their projects.

Configuration and deployment issues

Configuration and deployment issues concerning the **End-user subsystem** were mainly related to the development environment and the dependencies required to develop the End-user mobile applications. *"The development environment used throughout the course (Eclipse) is quite simple to use but requires a lot of effort to configure it for a given development project."* Depending whether the mobile application was native, hybrid, or web-responsive, diverse development environments and their dependencies had to be configured: *"I found very hard to configure PhoneGap and Apache Cordova in Windows."* As part of the **Gateways subsystem development**, various libraries have to be installed in order to ease and manage the communication with the End-user and the Back-end subsystem: *"The libraries were always difficult to install, and also very time consuming"* and *"In my personal experience, the OAuth authentication was very difficult to set up for its first use."*

In the **Back-end subsystem development** the sub-tasks aimed at designing, setting up and deploying the Persistence module, were perceived as time-consuming: *"While not difficult per se, these were the tasks that took most of my time. For example, deploying the database, designing the Entity-relationship model, and setting up the hardware."* Moreover, fixing technical details that may affect the development of the Back-end had some minor impact in the development of the Back-end: *"It takes a while to deploy the system because of some port conflicts derived from some issues in the configuration of the development environment."* Once implementing the **Back-end subsystem integration**, and specifically its integration with third-party services, OAuth requires configuring the Back-end from which the authenticated request will be made. Commonly it involves installing libraries, setting up various parameters, and configuring the third-party APIs in their corresponding web platforms. *"I had a lot of problems, and spent a lot of time, properly configuring the OAuth authentication between our application server and the one of Jawbone, it needed a lot of permissions."*

Summary of RQ3.3

RQ3.3: Why are these tasks perceived as the most complex?

By analyzing the comments of the respondents concerning their reasons behind the ranking of the sub-tasks, there might be identified, on the one hand, the lack of adequate documentation that might be understandable by the novices. And on the other hand, the lack of knowledge and expertise required to deal with several protocols, formats, authentication mechanisms, and real-time data. Concretely, their feedback could be categorized into: *Learning curve issues*, *Integration between subsystems issues*, or *Configuration and deployment issues*. Hereinafter I provide a

summary of the most common perceptions on each category.

Learning curve issues: In the **End-user** subsystem, these kinds of issues concerned the fact that most of the survey respondents were developing for the first time a mobile application. They faced the challenge of learning a new programming language and becoming familiar with the concerned IDE. When developing the **Gateways**, respondents struggled with understanding conceptually the RESTful web services before dealing with their implementation and consumption later (when implementing the **Gateways** integration subsystem). Furthermore, also concerning **Gateways** integration, achieving the integration with the **End-user** subsystem using other transmission protocols such as Bluetooth was perceived challenging due to the lack of adequate documentation. In the **Back-end** subsystem, the RESTful web service concept was mentioned once again as difficult to apprehend, particularly regarding the mapping between business logic methods and the web service endpoints.

Integration between subsystems issues: Without a doubt, the issues regarding the integration between subsystems were the most common according to the respondent's comments. The integration with both, self-implemented software components as well as with third-party services APIs, was perceived as complex. In the End-user subsystem, the integration with third-party services was particularly challenging. In the Gateways subsystem development, dealing with the OAuth authentication was perceived as complex given the flow, roles, and steps of this authentication mechanism. According to respondent's comments, the inherent complexity of OAuth was worsened by the lack of documentation understandable by novices. In the Gateways integration, dealing with streaming data coming in real-time from the Back-end was perceived both as extremely difficult and time-spending. In the Back-end, the main issues have to do with appropriately handling the data coming from the other subsystems into the application server business logic. As pointed out in the Gateways subsystem issues, dealing with the particular protocols, formats, and authentication mechanisms of each third-party service APIs was perceived as very challenging when implementing the Back-end integration.

Configuration and deployment issues: In the **End-user** subsystem, these kinds of issues concerned the proper configuration of the development environment and the dependencies required to implement the mobile applications. When developing the **Gateways** subsystem, the installation, and configuration of several libraries to manage the communication with the other subsystems were perceived as difficult and time spending, especially when the communication involved OAuth. In the **Back-end** development, the design,

setting up, and deployment of the persistence component (typically a relational database) was perceived as time-consuming. Also, the deployment of the Back-end itself (application server) was not trivial. Same as the Gateways integration, in the **Back-end** integration, dealing with third-party services requires several configuration and parametrizations so that authenticated request could be sent to the third-party services APIs.

Finally, from a system-level view, the sub-tasks that were perceived as the most challenging, are concerned with the integration between the subsystems. Whether classified as *Learning curve issues*, *Integration between subsystems issues*, or *Configuration and deployment issues*, many of the comments stressed the complexity inherent to the integration of heterogeneous subsystems. On the other hand, the Learning curve issues may be explained basically with the lack of knowledge of the respondents, or with the lack of adequate documentation that might be understandable for them.

3.4 Discussion

The *rating* of the sub-tasks provided a first perspective about how difficult did IoT novice developers find the implementation of concrete development sub-tasks. These sub-tasks were presented as detailed as possible without linking them to a specific programming language, framework or run-time environment. Differentiating difficulty level from time spent helped to understand which sub-tasks are complex just from the practical point of view (such as the time spending sub-tasks) and which other sub-tasks are also complex from the conceptual and learning curve perspective (such as the difficult and also time spending sub-tasks).

Later, aiming at clearly identifying the most complex sub-tasks, it was necessary to ask respondents to prioritize the three most complex sub-tasks they faced on each subsystem. In this way, from the *ranking* of the sub-tasks, it was possible to get a perspective of the most painful development issues. From this perspective, there were identified complex sub-tasks common to many subsystems, and complex sub-tasks that complement between them across various subsystems.

Thirdly, to understand the previous ranking, it was essential to capture qualitatively the *perceptions* of the respondents about the choice they made and their reasons behind that selection. These impressions led to the identification and categorization of the sources of complexity present in the implementation of particular software development tasks in the context of an IoT system. These categories were: *Learning curve* issues, *Integration between subsystems* issues, and *Configuration and deployment* issues.

By combining *rating*, the *ranking* and the *perceptions*, some specific programming areas were recurrently mentioned as particularly hard and painful. In particular, the integration of different subsystems, that require over-the-network communication protocols, and their debugging resulted considerably difficult, due to the diversity of client and server environments and the difficulty of tracing the remote calls. This was worsened by the fact that some third-party services are proprietary, give little visibility over their behavior, and each of them requires to follow different approaches and programming patterns. The integration with third-party services is particularly painful, whether it is the push notification service or the APIs that require OAuth authentication.

From the ranking of the sub-tasks, the development of the user interfaces in the End-user subsystem was among the most complex sub-tasks regardless of the kind of mobile application implemented (native, hybrid or web-responsive). This fact is somehow surprising if considering that the development of native mobile applications relies on specialized IDEs that in theory would ease the implementation of the views. These IDEs typically provide drag and drop features through which placing the user interaction components on the views should be easy enough. Notwithstanding, based on the comments of the respondents in the last section of the survey, I consider that the difficulties experienced by the novice IoT developers concern the binding between the business logic and the events generated at the user interface of the mobile application. In fact, the development of the controllers was also ranked as a complex sub-task in the implementation of hybrid and web responsive mobile applications. Relatively in line with the complexity experienced in the views and controllers development, the configuration of the development environment was also ranked as complex in native and hybrid mobile applications implementation. Once again, it is unexpected since the IDEs of such kind of applications usually have features to deal with external dependencies, as well as to debug and simulate the application execution.

Furthermore, also concerning the ranking of the sub-tasks, it was observed that sub-tasks aimed at integrating were among the most complex in several subsystems. Moreover, a complementarity between these sub-tasks, even if belonging to different subsystems, was identified. This complementarity involves:

- The design, implementation, and consumption of custom RESTful web services, that span across Back-end Development and Gateways Integration subsystem.

- The development of methods to achieve the connection between the Gateway devices and the third party services APIs, that spans across Gateways Development and Gateways Integration subsystems.

- The design, implementation, and integration of the persistence component, that spans across the Back-end development and the Back-end integration.

This observation emerged from the ranking and confirmed by the comments of the respondents, reinforce my initial perception of the significant complexity around the integration of heterogeneous software components. Both when consuming external services as well as when implementing custom integrations based on each project business logic. As outlined in the literature, dealing with the required level of interoperability is considerably painful to novices when developing IoT systems.

Furthermore, the lack of proper documentation and examples about how to integrate the subsystems is one of the main reasons why integration tasks are perceived as challenging. The difficulty to find well-structured documentation that might be understood by a novice was repeatedly mentioned by the respondents of the survey. For this reason, in all the subsystems the sub-tasks regarding integration were ranked among the first three.

The results from the survey point to the need to support novice IoT developers in dealing with several protocols, formats, authentication mechanisms, and streaming data coming in real-time. However, this support must be addressed both from the conceptual and technical perspectives. On the one hand, providing documentation that may be easily comprehended by non-expert IoT developers, and on the other, tools that would ease the configuration and development of certain software components integration.

From my knowledge of the university course that the survey respondents attended, I observed that the implementation of the integration between software components is similar across different projects developed by the respondents (especially when third-party services are involved). For this reason, I envisioned that, if documented, the code developed by the novices might provide some guidance to other programmers that are in the process of overcoming the same learning issues. In fact, being able to observe how someone else coded, what others paid attention to, and how they solved problems all support learning better ways to code and access to superior knowledge [33]. In this regard, from this observation emerged the *Code Recipes* [25], i.e., summarized and well-defined documentation modules, independent from programming languages or run-time environments. They are specified through a set of metadata and consist of multiple code fragments along with documentation and links to ease the understanding of such code, in order to implement a given integration between subsystems of an IoT system. This proposal is described in detail in Chapter 5.

Another possible approach might consider the automation of the sub-tasks that were rated as the most time spending. Both code generation and its deployment across several devices could be automated in such a way that novices just have to specify some custom parameters. Software Product Lines [76][1] might be a viable solution in this direction as long as it deals with the heterogeneity of IoT devices, protocols, and programming languages. Finally, it would be worth focusing on the lack of tools to debug the communication between devices and software components across the subsystems, giving some insight to the novices about possible failures in

the data exchange process.

In this work a reference architecture as generic as possible was proposed, taking distance from any technological stack. For this reason, I consider that the survey structured around this architecture could be useful to a wide range of IoT developers. Nevertheless, it must be said that since the current research questions concerned the software development perspective, my findings do not take into consideration the physical configuration and deployment of IoT systems. Future work could address the identification of the most complex issues concerning the Security implementation from the software perspective.

3.4.1 Implications

The survey presented in this work aimed at gaining an understanding of the challenges that novice programmers face when developing an IoT system. The first research question enabled us to weight all development tasks, identifying their difficulty level and the time that novices spent completing them. In the second research question, it could be determined that some of the development tasks perceived as complex concern aspects that are common across various subsystems. Examples are: the development of user interfaces, the configuration of development environments, and the development of the business logic. On the other hand, there are development tasks that concern aspects that are split across various subsystems. Examples are: the methods or functions to integrate the Gateway with the third-party service APIs, the implementation and integration of the persistence component, and the design, implementation, and consumption of the RESTful web services. The third research question provided insights about the causes behind the challenging issues faced by the novices. Learning curve issues, integration between subsystems issues, and configuration and deployment issues were identified. The most frequently reported issues concerned: the difficulty to find well-structured documentation that might be understood by a novice, the complexity inherent to the integration of the subsystems, and the integration with third-party services.

I consider that identifying these challenges might have an impact both in academic and industrial contexts. The findings of this survey could be used to ease the learning curve in the teaching scenario, and to make the IoT systems development more efficient in the software industry by improving on-boarding time estimations, hiring criteria, and human resource management within the projects. Special attention should be given to the integration of different subsystems, taking into account the various protocols, formats, authentication mechanisms (specially OAuth), and real-time data streaming. Among these integrations, the integration with third-party services resulted particularly painful. Therefore I suggest that research efforts should envision automation or debugging tools, as well as improved documentation strategies. Finally, more empirical studies are required to validate these findings with a diverse set of practitioners.

To my knowledge, there were no previous works assessing which are the most challenging issues faced by IoT novice programmers according to a concrete set of software development activities. The reported related works base their approaches on the authors' expertise, mainly. Additionally, the majority of the frameworks and toolkits for easing the development of IoT systems are constrained to a particular technological stack [17, 71, 34, 93, 31]. Instead, supported by my proposed generic architecture, the IoT developers survey (Chapter 3) aimed at gaining a better understanding of such issues independently from the projects, the architectural decisions, and the technology stack. Moreover, these issues are expressed as concrete software development tasks that belong to a specific part of the architecture.

3.5 Validity of results

This section examines the threats to the validity according to the classification schema proposed by Cook and Campbell [21]. In that schema are defined four types of threats to validity: conclusion validity, internal validity, construct validity, and external validity. The following is a detailed description of each type of threat to validity, as well as some considerations about the repeatability of this work (Subsection 3.5.5).

3.5.1 Internal validity

Threats to internal validity regard issues that may indicate a causal relationship, although there is none [105]. In this survey, threats to internal validity concerned the *instrumentation*, the *subjects selection*, and the *maturation*.

With regard to the *instrumentation*, the pilot survey (Section 3.3) enabled me to inspect and determine to what extent the generic architecture, its subsystems, and the corresponding sections, tasks, and sub-tasks were understandable, pertinent and complete. In this preliminary study, the pilot students could give their impressions on the questionnaire and point out any other tasks that were not listed but resulted complex to achieve. This iteration with the initial pilot survey contributed to avoid possible threats concerning the *instrumentation* due to poor question wording, unclear documentation, or bad instrument layout.

Concerning the *subjects selection*, students took part in the survey voluntarily. Nevertheless, the draw of a wireless speaker was used to motivate their participation. However, as already mentioned in the Survey Design and Methods section, the consistency between the time spent and the number of sections answered, indicates that respondents did not skip several sections or sub-tasks of the questionnaire intentionally to participate in the draw. Consequently, it can be assumed that the voluntary basis and the motivation draw did not influence the obtained results

significantly. Furthermore, as stated in Section 3.5 and shown in Table 3.3, the distribution of exam scores was checked to exclude the risk that only the best students were motivated to participate.

Finally, to avoid the *maturation* threats due to the fatigue or boredom, the participants were allowed to save partially finished surveys, so they had the chance to complete it in different moments. Nevertheless, as shown in Table 3.4, all the participants completed the survey answering it in their first *attempt*, and completed it within 46 minutes.

3.5.2 External validity

Threats to external validity are conditions that limit the ability to generalize the experiment results outside the experiment settings [105]. In this survey, external threats to validity had to do with the *representativeness of the subject population* and the *representativeness of the experimental setting*.

In what refers to the *representativeness of the subject population*, in the survey what happened was that, although most of the respondents belonged to Computer Engineering and Electronic Engineering, the invitations to participate in the survey were addressed to students belonging to 7 different engineering degrees. Additionally, as mentioned in Subsection 3.2.5, some of them were foreign students (Erasmus or other student exchange programs). In fact, on average, the course hosts a cohort (20%-25%) of students from foreign universities [24]. This implies that they received education at different universities and under a different curriculum. Additionally, the ages of the participants ranged from 22 to 39 years old. Finally, no participants were on the basis of their final grades.

Consequently, the *subject population*, given their heterogeneity regarding the bachelor degree, the home university, the age, and the performance in the course, although not statistically representative, is a mix that is frequently encountered among IoT novice developers. However, it would be desirable to have had more women participating, as well as a higher percentage of students from other disciplines, apart from Computing Engineering and Electronic Engineering. Nevertheless, given the demographics of the course itself, such a scenario was not likely in the survey.

On the other hand, in pursuit of an *experimental setting representative of the studied context*, I decided to design the survey based on the current IoT state of the art, considering the industrial perspective and using it to analyze the academical projects that the novices implemented. Concerning the industrial perspective, I started from the analysis of various reference IoT architectures that were developed by some of the most influential industry actors in the IoT landscape. Later, concerning the academical perspective, I faced the challenge to make the reference IoT architectures understandable to the students based on the experience that they

acquired when developing their projects. To that end, the architectures of the Ambient Intelligence course projects developed over the years were analyzed, identified commonalities, and mapped their components to the building blocks of the industry IoT reference architectures. Based on these analyses, from the reference architectures and the course projects' architectures, a generic architecture was developed to structure the survey. The purpose of structuring the survey upon this generic architecture was, on one hand, to provide respondents with a common understanding about the software components involved in an IoT system, and on the other hand, to be in line with the industry state of the art, notwithstanding the specificities of the projects that were developed in the course. Similarly, I tried to avoid as much as possible to tie the generic architecture to a specific architectural pattern or technology stack. In this manner, I wanted to guarantee that the proposed generic architecture and the survey would be useful in other scenarios, independently from the software stack, to assess the most painful issues that novices experience while developing IoT projects.

However, it must be pointed out that, as already described in Section 3.2.1, the Security component was not represented in the generic architecture because it was outside the course syllabus. Therefore, if this survey would be applied in an IoT-related course in which novice programmers are exposed to the security issues that emerge from security-related operations, a new component must be added to the generic architecture as well as its associated tasks and sub-tasks. Nonetheless, according to the experience reports that were described in the Related Works Chapter (Section 2.2), it is very unlikely to introduce security concerns in a course targeted at IoT novice programmers. In the same vein, the focus of this survey was on the software development perspective of IoT systems. If a later study aims at getting understanding about the most painful issues concerning the deployment of the hardware, I consider that the generic architecture is still valid, but the Sensors and Actuators subsystems, which are already represented in the generic architecture, must be defined in terms of a new set of tasks and subtasks. Similarly, on every subsystem, new tasks and subtasks must be added to the ones that were already defined in this work.

Additionally, I consider that the findings of this survey might be partially extended to more experienced developers. Nevertheless, it should be kept in mind that the novice developers who participated in this survey were unfamiliar with many of the software development tasks that they faced. Most of them were approaching for the first time to these tasks, and they were not using any advanced IDE or development tool. For this reason, the perception of the challenging issues that they faced was not biased by their expertise on a given software development area nor by the mastery of a given software stack that would ease the completion of those tasks. Experienced developers, on the contrary, are specialized on a given software development area, and are proficient with a set of advanced development tools. For instance, an experienced mobile developer would rate such development

tasks as simple and not time-spending, and his perception would not be comparable with the rating of a non-expert mobile developer. Consequently, the main obstacle of extending the findings of the survey to more experienced developers is the disparity of expertise, development resources, and background skills.

3.5.3 Construct validity

Threats to construct validity refer to the extent to which the experiment setting actually reflects the construct under study [105]. In this survey, construct threats to validity concerned the *inadequate preoperational explication of constructs*, the *mono-method bias*, and, in the field of social threats, the *evaluation apprehension*.

To avoid the *inadequate preoperational explication of constructs*, when formulating the first research question, it was decided to study the complexity, which is an ambiguous concept by itself, in terms of well-defined constructs such as difficulty-level and time-spent. This way, before translating these constructs into measures, I made sure that the criteria to quantify the complexity was clear enough to the respondents. Furthermore, to avoid *confounding constructs and levels of constructs*, these difficulty-level and time-spent constructs were translated into a 5-point ordinal scale, taking into account that more than their absence or presence, it was the level of difficulty and time-spent which is of importance to the outcome.

Furthermore, to avoid *mono-method bias*, the research questions were addressed using different kinds of measures. Namely, the rating of the sub-tasks (measured using a Likert scale) was complemented with the ranking of the most complex ones and cross-checked with the justification of the participants about their ranking choice (captured through an open question). Thus, by including quantitative and qualitative measures, the goal was to draw conclusions free from measurement bias.

Regarding the social threats, I opted to conduct the survey anonymously, and only after the participants completed with success the course. In this manner, it was attempted to avoid *evaluation apprehension* so that the participants were guaranteed that they could be sincere about all the issues that they may have experienced without being afraid of some negative impact in their course grade, nor expecting some bonus. Furthermore, it was also explained very clearly to the participants that the study was not aimed at evaluating their skills or performance but to identify the most challenging issues that they experienced during the development of their projects.

3.5.4 Conclusion validity

Threats to conclusion validity regard issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment [105]. In this survey, conclusion threats to validity regarded the *low*

statistical power, the *reliability of measures*, the *reliability of treatment implementation*, and the *random heterogeneity of subjects*.

Concerning the *low statistical power*, I would have liked to have a larger sample size to achieve higher statistical power and afford better generalizability. Nevertheless, the length of the survey hampered higher participation (for every completed survey, there were 1.12 abandoned surveys, i.e., students who started but did not complete the survey) and, on the other hand, a shorter version would have covered only a few topics. However, in this regard, it must be clarified that, more than attempting to infer a property of a population, the survey was trying to explore the variety of challenges that IoT novice developers face. Besides, the open questions could contribute to detecting a mismatch between the interpretation of the data and respondents' experience.

The validity of an experiment is highly dependent on the *reliability of the measures*, and they can be negatively affected by factors such as poor question wording, bad instrumentation, or bad instrument layout. In this sense, it can be assumed that, as previously mentioned in the internal validity threats, the pilot study (Section 3.3) enabled us to avoid these instrumentation-related factors. Furthermore, the *reliability of treatment implementation* describes the risk that the implementation is not similar between different persons applying the treatment or across different occasions. Since the survey was applied online with the same platform, and under the same conditions for all the participants, I consider that the implementation is rather similar between the respondents. Finally, as previously stated in the external validity threats, the subject population has a reasonable degree of heterogeneity but framed in the context of novice IoT programmers. Hence, such heterogeneity is not considered to lead to *random heterogeneity of subjects*, which is the risk that the variation due to individual differences is larger than the variation due to the treatment [105].

3.5.5 Repeatability

The generic architecture proposed in this work, all its tasks, and their subtasks, as well as the structure of the survey, and the followed methodology are available from the authors (Section 3.2). Therefore, concerning the artifacts used for the survey execution, the repeatability of the results of this study is considered to be good. Nevertheless, three limits to repeatability were identified: although the architecture was proposed to be as generic as possible if it has to be used in specific domains or contexts, new sections must be added; secondly, if the survey is to be used in a different educational scenario, the course to be analyzed has to be project-based, otherways the instrument cannot be applied; finally, if specific technologies, tools or languages are imposed, the proposed set of tasks and subtasks have to be redefined to match the technical specificities of the concerned technology stack, instead of being technology-neutral as in this work.

3.6 Conclusion

In this work, a survey was conducted to identify the most complex issues experienced by novice programmers when developing IoT systems. The survey was framed into a generic interpretation framework in which the architecture, subsystems, and software development tasks of a significant subset of these kind of systems were abstracted. This survey was conducted among 40 undergraduate students that developed an IoT project during three editions of a university course. The most complex issues were identified on the basis of the rating of software development tasks according to their difficulty level and completion time; the ranking of the most complex tasks; and the qualitative perception of each respondent about such complexity. Through a generic interpretation framework, a system-level view of the main issues was achieved and presented. To the best of my knowledge, this is the first study to express the complex issues as concrete development tasks that are not dependent on a particular kind of project, its architecture, or its technology stack.

The results that emerged from the application of the survey enabled to determine that the most challenging issues reported by unexperienced IoT developers concerned: the difficulty to find well-structured documentation that might be understood by a novice, the complexity inherent to the integration of the subsystems, and the integration with third-party services. Moreover, it could also be identified, on the one hand, aspects that were perceived as complex across various subsystems (development of user interfaces, the configuration of development environments, and the development of the business logic), and on the other hand, aspects whose complexity is split across various subsystems (integration between the Gateway and the third-party service APIs, the implementation and integration of the persistence component, and the design, implementation, and consumption of the RESTful web services). I consider that the findings enable to ease the learning curve in the teaching of IoT, and might help to improve on-boarding time estimations, hiring criteria, and human resource management within the industry IoT projects. Finally, based on the obtained results, I consider that research efforts should envision automation or debugging tools, as well as improved documentation strategies for the development of IoT systems.

Chapter 4

IoT Development in the context of Open Source Software

The outcomes from the research presented in the previous chapter provided insights into the issues that were perceived as the most challenging ones, by novice programmers, when implementing an IoT system. To complement the perception of the novice developers with the analysis of currently developed IoT systems, and to gain a deeper understanding of how experienced or professional developers contribute to successful and complete IoT applications, a quantitative analysis was conducted among a broad set of the 60 most popular publicly available IoT and non-IoT projects. By comparing how developers contribute to these projects, this quantitative analysis provides insight into the purpose and characteristics of the code, the behavior of the contributors, and the maturity of the IoT software development ecosystem. This chapter describes the differences that were identified between IoT and non-IoT projects in terms of how applications are realized, developers' specializations, and code reusing. Part of the work described in this chapter has been previously published in [28].

4.1 Motivation

From a technical point of view, several definitions have been proposed for the Internet of Things [42] and various enabling technologies are considered to characterize IoT applications. As earlier stated, from the software point of view, the implementation of IoT applications is particularly complex and differs from the development of mobile and web applications. According to Taivalsaari *et al.* [94], for instance, IoT development differs from mainstream mobile app and web application development in several ways, summarized by the authors into a set of *dimensions* that are unfamiliar to most software developers. Multi-device programming, the reactive nature of the application, the distributed nature of the software, and the need

to write fault-tolerant software, are among these dimensions, which IoT developers must consider.

Against this backdrop, the work presented in this chapter relied upon software mining to gain an understanding of how developing IoT applications is different from developing non-IoT applications in the OSS context. To that end, an empirical study mining 60 OSS repositories publicly available on GitHub was conducted. 30 IoT OSS and 30 non-IoT OSS projects were mined to analyze *a)* the way developers contribute to their projects, *b)* the files that they tend to modify the most, and *c)* the specialization and the evolution of these modifications. Finally, the maturity of the IoT software development ecosystem was assessed based on a dependency analysis in the selected projects. Besides leveraging a characterization of IoT OSS projects currently available for IoT developers, this work aimed at providing evidence from a practical point of view about the IoT software development peculiarities that should guide future research efforts to better understand and satisfy software engineering needs in the IoT context.

The remainder of the chapter is structured as follows. Section 4.2 describes the research goal and questions and outlines the selection process. Section 4.3 characterizes the selected OSS projects and describes the quantitative analysis conducted over them as well as the outcome of the analysis. Section 4.4 discusses the results and presents further implications, while threats to validity are outlined in Section 4.5. Finally, section 4.6 presents the conclusion of this research work.

4.2 Research Goal and Questions

The overall goal of the research presented in this chapter was to explore the potential differences between the development practices for IoT and non-IoT projects in the OSS context. In particular, it aimed at identifying (a) the behavior of developers and the diversity of resources they manage, and (b) the reuse of features through the adopted dependencies. These two criteria led to the research questions set out below.

4.2.1 Research Questions

The first research goal was to investigate whether and how developers adopt different programming languages and cover various specializations in IoT vs. non-IoT OSS projects. In particular, concerning:

- how different programming languages are used in the two domains;
- whether IoT developers are more specialized in some programming languages or certain types of files in their project;
- how the usage of such programming languages evolves over time.

Therefore, the corresponding research question was:

RQ4.1: How developers of IoT vs. non-IoT OSS applications contribute to their projects regarding the programming languages that they adopt?

Furthermore, this quantitative investigation exploited OSS repositories focusing on the maturity of the IoT ecosystem. Such maturity was investigated in the selected repositories by analyzing project dependencies, how many they were, and which were the most popular ones. Additionally, focusing exclusively on the IoT OSS projects, there were also investigated which aspects of IoT application development these dependencies addressed and how often they were used by IoT developers. This led to the second research question:

RQ4.2: How developers exploit dependencies to reuse features in IoT vs. non-IoT OSS projects?

4.2.2 Selection of the Analyzed Repositories

To select a prominent widely-known and widely-used set of IoT OSS repositories from GitHub, they were first filtered them by topic, choosing the ones that belong to the `iot` or `internet-of-things` topics on GitHub. Topics are labels to classify a repository based on its intended purpose, subject area, community, or language. They appear on the main page of a repository and repository administrators can add as many topics as they want to a repository.

Once the repositories belonging to the IoT topic were filtered, 4,696 of them were retrieved. Therefore, to prioritize the most popular and well-evaluated ones, they were sorted according to the decreasing number of stars. Stars enable GitHub users to keep track of repositories they find interesting and to discover similar repositories [11], as well as to show appreciation to the repository maintainers for their work[1]. Lastly, the 30 top-starred repositories were taken, provided they were open-source code repositories. In fact, since a large portion of repositories on GitHub are not for software development [52], they were inspected manually to exclude the ones that were not software related (i.e., tutorials, documentation pages, icon-packs, fonts) or without an open-source license.

The same procedure was followed to select the non-IoT repositories. The only difference was that the filter was modified to include repositories belonging to any topic *except* `iot` and `internet-of-things`.

[1] https://help.github.com/articles/about-stars/, last visited on June 6, 2019

The data used in the analyses reported in this study was mined from GitHub in August 2018. Tables 4.1 and 4.2 list the selected IoT and non-IoT repositories along with their salient characteristics. Most of the information about the repositories was gathered through the GitHub GraphQL API v4[2].

[2] https://developer.github.com/v4/, last visited on June 6, 2019

Table 4.1: IoT popular Open Source GitHub repositories

Repository name	Genre	Size (kB)	LOC	Prim. Lang.	# Langs.	# Depend.	Commits	Contributors
netdata	Monitoring agent	24,473	259k	C	14	18	7,321	223
kong	API gateway	9,802	151k	Lua	4	24	4,118	141
home-assistant	Home automation	85,734	562k	Python	4	456	14,773	1,211
johnny-five	Robotics programming framework	92,868	136k	JavaScript	3	338	3,215	152
gun	Graph database engine	29,683	124k	JavaScript	5	139	1,532	66
timescaledb	Time-series database	2,975	159k	C	9	0	833	33
gobot	Programming framework	9,668	179k	Go	4	16	2,507	109
node-serialport	Package to access serial ports	2,688	19k	JavaScript	5	18	1,208	145
emqx	MQTT broker	11,682	29k	Erlang	2	4	3,060	53
cylon	Programming framework	20,046	7k	JavaScript	2	5	1,323	26
urh	Wireless protocols monitoring	43,538	197k	Python	4	4	2,579	13
ArduinoJson	Arduino library	3,242	34k	C++	6	0	984	10
platformio-core	Cross-platform IDE	34,704	10k	Python	5	6	3,626	27
crate	Distributed SQL database	86,570	667k	Java	6	15	8,775	62
RIOT	Operating system	56,065	2.04M	C	10	0	19,368	287
thingsboard	IoT platform	8,785	257k	Java	9	109	1,510	59
rt-thread	Operating system	254,378	13.38M	C	24	0	6,549	226
blynk-library	IoT platform	9,318	33k	C++	7	0	1,691	19
openthread	Thread networking protocol	50,835	1.5M	C++	10	4	2,443	85
mongoose-os	Firmware development framework	44,630	106k	C	10	0	4,212	32
vernemq	MQTT broker	11,426	67k	Erlang	8	21	1,713	22
BerryNet	Deep learning gateway	181	14k	Python	4	14	159	6
PJON	Network protocol	4,529	29k	C++	2	0	1,999	34
zephyr	Operating system	123,632	10.05M	C	12	0	23,231	420
blynk-server	IoT platform	29,763	81k	Java	6	21	4,545	14
paho.mqtt.android	MQTT client library	2,014	15k	Java	3	0	194	20
tock	Operating system	136,163	79k	Rust	8	0	4,085	82
kaa	IoT platform	180,031	775k	Java	16	3	6,691	109
Sming	Programming framework	52,066	195k	C++	16	11	1,236	97
homie-esp8266	MQTT convention	1,305	10k	HTML	5	142	1,714	155

Table 4.2: Non-IoT popular Open Source GitHub repositories

Repository name	Genre	Size (kB)	LOC	Prim. Lang.	# Langs.	# Depend.	Commits	Contributors
bootstrap	Web UI framework	124,291	102k	CSS	6	52	17,950	1,192
vue	Web UI framework	23,784	164k	JavaScript	6	89	2,620	209
react	Web UI framework	137,522	277k	JavaScript	10	78	10,326	1,379
tensorflow	Machine learning framework	189,005	4.5M	C++	21	31	41,273	1,923
d3	Data visualization library	35,963	55k	JavaScript	1	39	4,153	133
oh-my-zsh	Zsh framework	4,730	60k	Shell	6	0	4,785	1,472
react-native	Native apps framework	256,123	602k	JavaScript	17	73	14,743	2,157
electron	Desktop applications framework	40,449	195k	C++	11	829	20,369	911
linux	Linux kernel	2,192,884	26.5M	C	19	3	782,537	19,120
angular.js	MVC web framework	98,788	554k	JavaScript	5	76	8,883	1809
vscode	IDE	149,758	1.12M	TypeScript	33	102	40,831	834
create-react-app	React app setup command	5,718	129k	JavaScript	5	18	1,782	551
animate.css	CSS animations library	713	6k	CSS	2	572	423	101
node	JavaScript runtime engine	399,664	9M	JavaScript	13	12	23,947	2,480
moby	Programming framework	137,525	1.43M	Go	8	4	35,841	2,120
jquery	JavaScript library	27,910	93k	JavaScript	4	39	6,343	349
axios	Promise based HTTP client	2,762	10k	JavaScript	3	37	833	171
atom	Text editor	301,069	223k	JavaScript	7	514	35,684	516
go	Go programming language	182,273	2.44M	Go	16	11	38,051	1,404
laravel	Web framework	9,222	5k	PHP	3	8	5,804	573
swift	Swift programming language	324,175	2.47M	C++	17	0	78,463	779
three.js	JavaScript 3D library	662,910	1.39M	JavaScript	5	399	25,250	1,238
redux	Programming framework	6,562	184k	JavaScript	3	29	2,684	666
socket.io	Real-time application framework	12,264	182k	JavaScript	1	11	1,706	171
webpack	Module bundler	16,478	124k	JavaScript	5	74	7,259	552
Semantic-UI	Web UI framework	110,010	283k	JavaScript	3	508	6,659	232
reveal.js	HTML presentations framework	8,271	33k	JavaScript	3	15	2,200	264
rails	Web framework	165,151	524k	Ruby	7	62	70,368	4,490
meteor	Web framework	76,020	529k	JavaScript	9	12	21,688	474
kubernetes	Container-orchestration system	797,674	4.72M	Go	10	10	73,512	1,951

4.3 OSS Projects Analysis

4.3.1 Projects Characterization

Before diving into the research questions, a characterization of the selected projects is reported to provide a brief but complete overview and to set the stage for the subsequent analysis. Each project was examined individually to understand its purpose and to assign it a genre. The genres aimed at describing the nature of the projects. Then, through the GitHub API, several characteristics were gathered, namely: the topics, their size (kB and lines of code), their primary language, and their total number of programming languages.

As observed in Table 4.1, the **genre** of the IoT OSS projects, that was assigned by me after manually examining each repository, is heterogeneous. They are scattered across operating systems, programming frameworks, libraries, network protocols, databases, IoT platforms, and IDEs. At first glance, no clear trend emerged concerning their purpose or application domain. On the contrary, when analyzing non-IoT projects (Table 4.2), it can be noticed that most of them are related to the web development area, with just 12 exceptions, such as a machine learning framework, a Z shell (Zsh) framework, an operating system kernel, an IDE, a text editor, and a couple of open source programming languages.

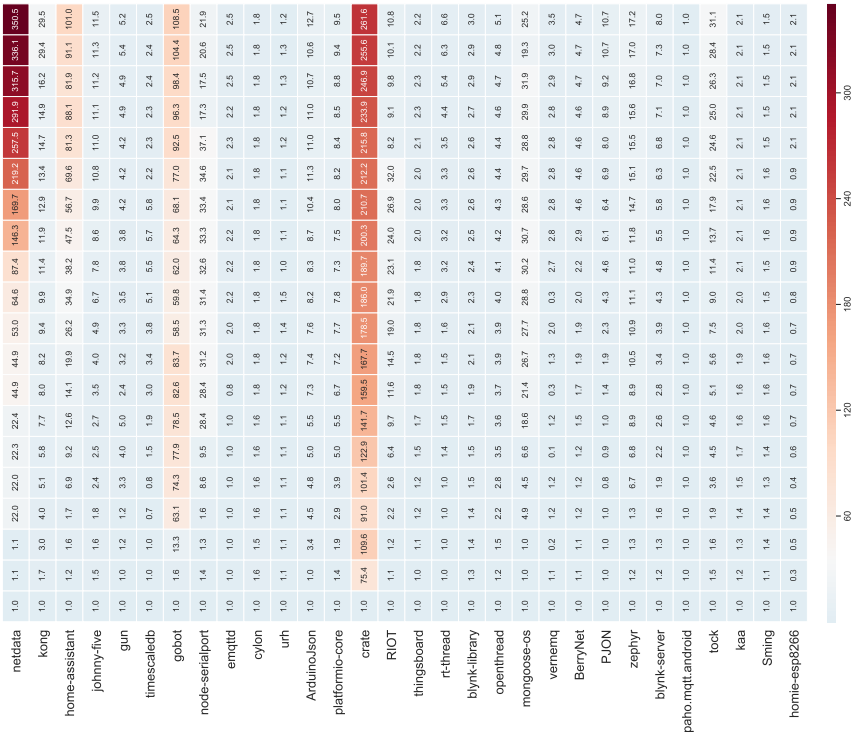
The fifteen most commonly used **topics** across the IoT projects (*mqtt*, *raspberry-pi*, *arduino*, *hardware*, *esp8266*, *esp32*, *embedded*, *robotics*, *javascript*, *java*, *iot-platform*, *i2c*, *home-automation*, *gpio*, *docker*) did not reveal a prevailing technology or application domain. Instead, the 15 topics across the non-IoT projects (*javascript*, *nodejs*, *html*, *framework*, *electron*, *css*, *windows*, *web*, *ui*, *react*, *python*, *macos*, *linux*, *go*, *frontend*) were mostly about web development. This fact indicates that neither in the clasification proposed in Tables 4.1 and 4.2, nor in the labels assigned by the owners to their IoT projects, there is a strong focus towards a particular domain or technology, thus further motivating the research questions. Furthermore, the initial observations regarding the genre and the topics of the projects seemed to be in line with various authors [42, 94, 60], who point out that the development of IoT applications is more complex and requires programmers with skills and expertise in several domains as might be, for instance, mobile and cloud computing, embedded devices, database design, and web development.

Concerning the **size** of the projects (in kB), the average non-IoT project was almost three times larger (4.56x) than a typical IoT project. However, when looking at LOC (Lines Of Code), this difference decreased significantly: on average, non-IoT projects contained 1.9M LOC, while IoT projects 1.0M (1.9x). The largest IoT project, for both kB and LOC, corresponded to `rt-thread`, a real-time IoT operating system for embedded devices. Similarly, the largest non-IoT project was the `Linux` kernel followed far behind by `kubernetes`. The smallest IoT project, in kB, was `BerryNet`, a project to turn edge devices such as Raspberry Pi 3 into

intelligent gateways with deep learning capabilities running locally, on the edge device itself, without the need of an Internet connection. For what concerns LOCs, instead, the smallest IoT project is `cylon`, a JavaScript framework for robots, drones, and the IoT, developed for Arduino and similar boards. As may be observed in these last two projects, achieving a small size is fundamental given the fact that in most cases IoT software components are deployed on constrained devices with low computational and/or storage resources. This same restriction holds for most of the other IoT projects, especially those to be deployed on the gateway architectural element.

Additionally, to put into perspective the comparison of the projects' size, the growth of the source code along the projects' lifetime, is illustrated through the heatmap graphs in Figure 4.1 (for IoT projects) and Figure 4.2 (for non-IoT projects). In these heatmaps, the growth of the projects' source code is expressed as the proportion between the initial size of the programming files and their size along the lifetime of the projects. Therefore, to build these graphs, the period between the first commit in the project and the last commit before August 2018 (the date when the repositories were mined for this analysis), was divided into 21 equally spaced date intervals for each project. Then, on each of these dates, the corresponding version of the project was checked out from GitHub; and the size of its programming files was calculated relying on Linguist; the open-source library that GitHub uses to determine file languages for syntax highlighting, and project statistics[3]. Specifically, through the Ruby API provided by this library, that, given a directory, returns a dictionary with the detected programming languages along with their size.

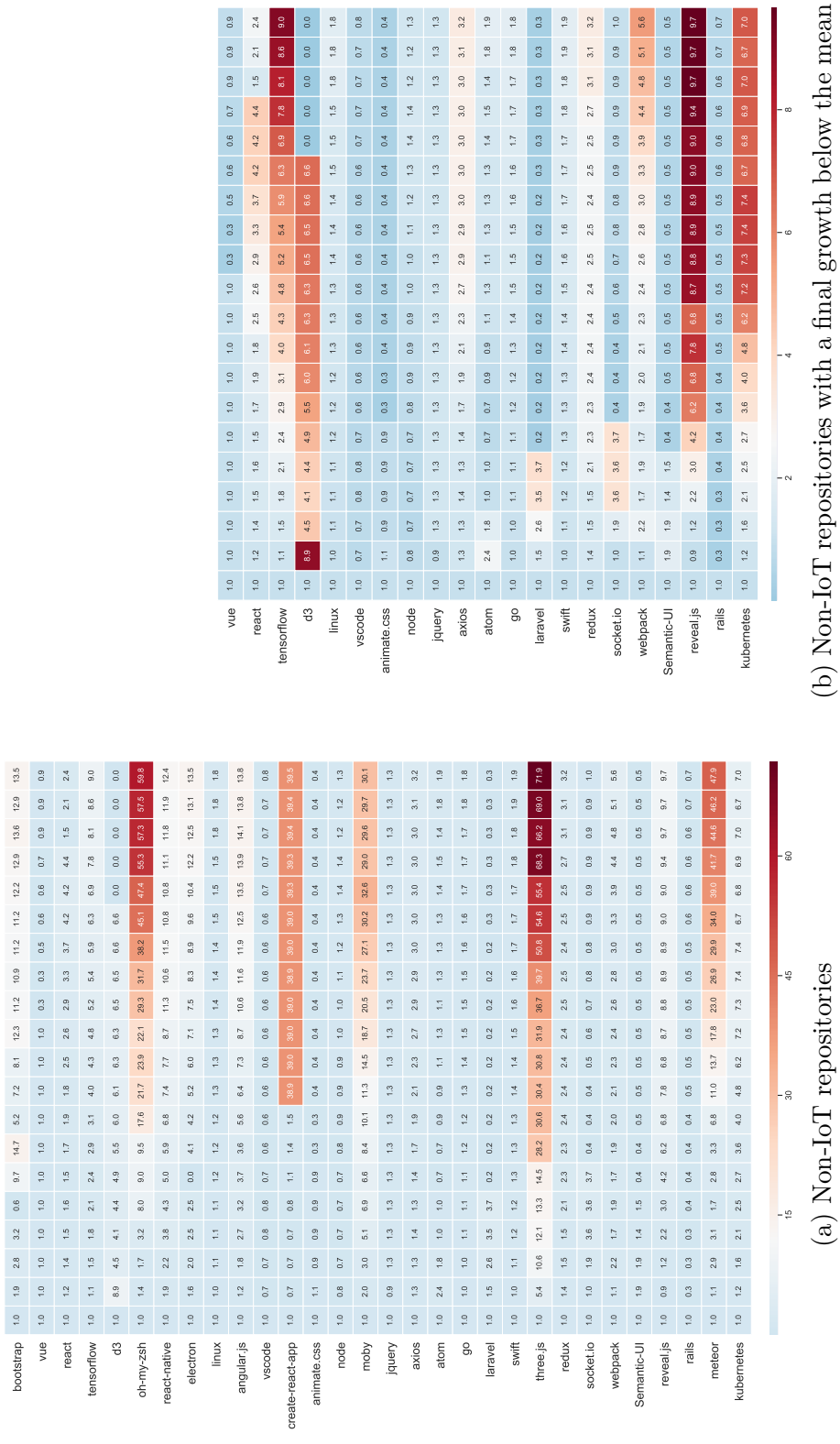
[3] `https://github.com/GitHub/linguist`, last visited on November 26, 2019



(a) IoT repositories

(b) IoT repositories with a final growth below the mean

Figure 4.1: Growth speed of the IoT repositories



(a) Non-IoT repositories

(b) Non-IoT repositories with a final growth below the mean

Figure 4.2: Growth speed of the non-IoT repositories

The growth of the project was calculated by dividing the size of each checked out version of the project by the size of the second checked out version. In fact, to avoid empty projects (without source code) that would have made the calculation impossible or meaningless, the second version was taken instead of the first one. In this manner, the first measure was always 1.0, and the following values represented the variation regarding the initial size of the projects' programming files. Hence, the last measure represents how many times the source code grew in comparison with respect to its initial size.

As can be observed in Figure 4.1a, a subset of four IoT projects grew up hugely. Namely `netdata` (350 times, 24.4 MB, and 7.3k commits), `home-assistant` (101 times, 85.7 MB, and 14.7k commits), `gobot` (108 times, 9.6 MB, and 2.5k commits), and `crate` (261 times, 86.5 MB, and 8.7k commits). Indeed, while the average growth is 35.15 times, the standard deviation is 78.83 times. To improve the readability of the graph for project with less dramatic growth, a second heatmap visualization was generated, considering only the projects whose final growth is below the mean (Figure 4.1b).

Concerning non-IoT projects (Figure 4.2b), five of them grew up significantly, although not as dramatically as the subset of IoT projects that grew above the mean. These repositories were: `oh-my-zsh` (60 times, 4.7 MB, and 4.7k commits), `create-react-app` (39 times, 5.7 MB, and 1.7k commits), `moby` (30 times, 137.5 MB, and 35.8k commits), `three.js` (72 times, 662.9 MB, and 25.2k commits), and `meteor` (47.9 times, 76.0 MB, and 21.6k commits). The average growth in non-IoT projects is 11.88 times, and the standard deviation 18.81 times. As with the IoT projects, Figure 4.2b reports a second heatmap visualization with the IoT projects whose final growth is below the mean.

Among the IoT projects, `paho.mqtt.android` is the one that has remained more stable over time (1.0 times, 2.0 MB, and 194 commits), it consists of an MQTT client library written in Java for developing applications on Android. Nevertheless, the last of its 195 commits was 4ht October 2017, and it has just two releases. After it, the project that remained more stable was `urh` (1.2 times, 43.5 MB, and 2.5k commits), it consists of a tool for analyzing unknown wireless protocols by taking samples from Software Defined Radios and transforming them into binary information. For its part, the non-IoT project whose code growth remained more stable over time is `socket.io` (1.0 times, 12.2 MB, and 1.7k commits), a library that enables real-time, bidirectional and event-based communication between the browser and the server.

4.3.2 RQ4.1: Development Activities

An analysis of the commit history for all the OSS projects was performed to answer RQ4.1. In particular, each repository was cloned locally so that its git

history could be saved into an external text file, and processed later by a custom-developed text mining tool. This tool extracted from each commit the set of files that were modified, the modification date, and the author name. Several classifications and cross-checking analyses over this information allowed us to determine the most widely-modified file formats, and especially the commit history over time of such resources. In addition, complementary information from the GitHub API, was gathered when appropriate.

Distribution of programming languages: Among the information that Linguist provides there is the primary language, which is the most used programming language within a project (Figure 4.3). The most popular primary programming language among non-IoT projects is JavaScript, which is the also the *lead* language since 18 non-IoT projects use it (60%). It is followed far behind by C++ and C (3 and 1 project, respectively). IoT projects, instead, exhibit a more balanced distribution of primary languages, with the most popular languages being C, C++, Java, Python, and JavaScript. All of them are the primary language on almost the same number of projects (from 4 to 6 projects, each).

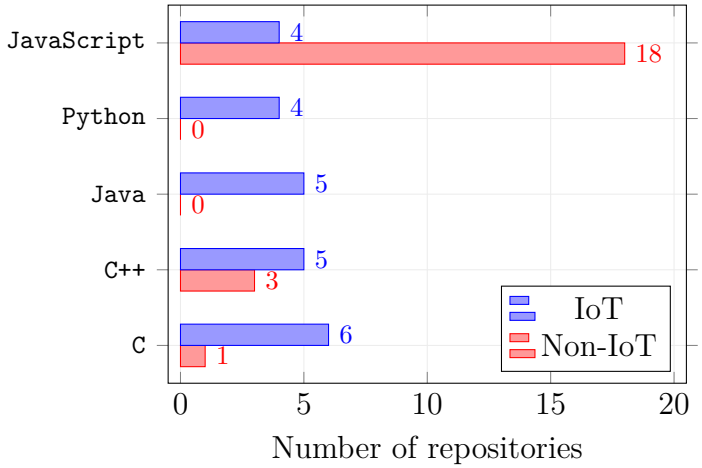


Figure 4.3: Top primary programming languages in IoT and non-IoT repositories

Besides the primary language, there are several other languages on each project: on average, 7.4 different languages for non-IoT projects vs. 8.3 for IoT projects. To gather additional insight on this comparison, since the averages' difference is not statistically significant due to the small size of the sample, the percentage of files written in a given programming language was compared with the number of projects in which that language is present and reported it in Figure 4.4. It illustrates, given a programming language, the number of projects in which it is present, and the average percentage of files on those projects. Regarding this graph, it can be observed that no languages were present on a high number of IoT projects with a significant percentage of files (right-upper quadrant). In most of the IoT projects,

the chart identifies programming languages that are present in many projects with a marginal percentage (right-lower quadrant), as well as programming languages that have a significant percentage of files but just on a few projects (left-upper quadrant). In the first category, Java and Erlang have a significant percentage of files on a few projects. In the second category, C++, C, and Python are present in around half of the projects, with percentages of files ranging from 26% to 32%. Furthermore, several IoT projects have a small portion of Shell scripts (on average, 0.96% of the files in 23 projects).

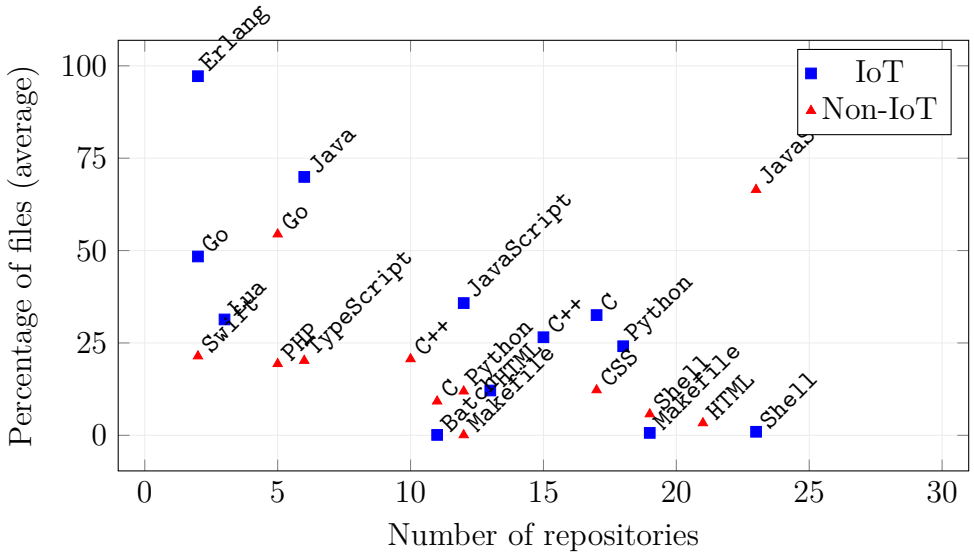
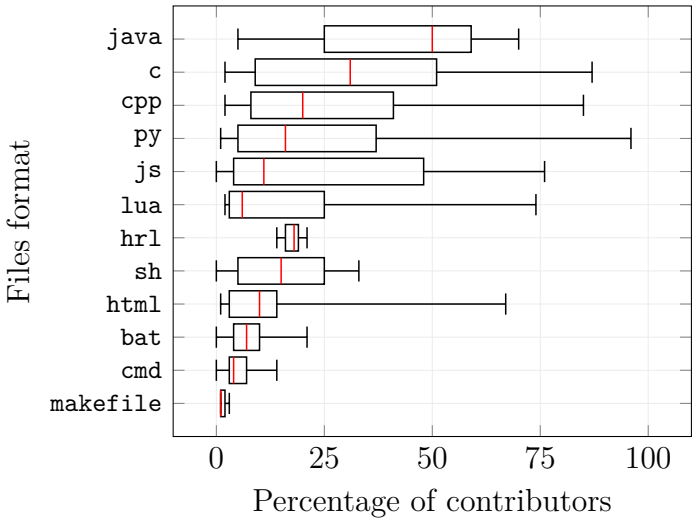


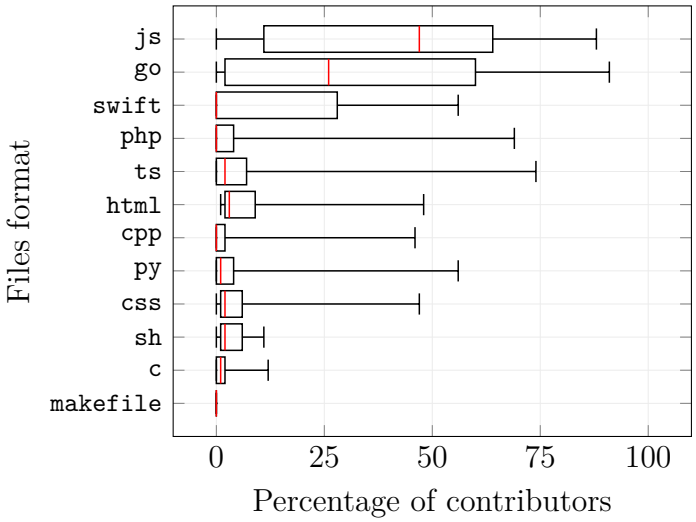
Figure 4.4: Presence of programming languages in IoT and non-IoT projects

For non-IoT projects, JavaScript is still the only programming language with a significant percentage of files on most projects (66.47% on 23 projects). This results gives an initial indication that the programming languages IoT developers deal with are observably different and more varied from those worked on by non-IoT developers, and supports the idea that the development of IoT applications requires programmers with skills and expertise in several domains.

Specialization of contributors by programming language: Figures 4.5a and 4.5b illustrate the average percentage of contributors that modify the files developed in a given programming language, among the projects where it is present. Inside the IoT projects, the files modified by a higher proportion of the contributors are Java, C, C++, Python, and JavaScript. As before, this result indicates that programming languages used by IoT developers are more variegate and diverse than other contexts, with a lower specialization towards a few lead languages. On the contrary, shell executable files, batch files, and command files are manipulated by a percentage that reaches, on average, 15% of the contributors. This percentage suggests a higher level of specialization for shell-oriented languages.



(a) Percentage of contributors that modified certain files formats in IoT repositories



(b) Percentage of contributors that modified certain files formats in non-IoT repositories

Figure 4.5: Percentage of contributors by file format

For what concerns non-IoT projects, the files modified by a higher proportion of the contributors are by far JavaScript and Go. The rest of the files are modified by a dramatically lower proportion of contributors. Moreover, shell-oriented files (e.g., sh files) in non-IoT projects are modified by a significantly lower proportion of contributors, in comparison with IoT projects. However, it must clarified that Figure 4.5 does not represent an overall ranking of the most used programming languages among IoT and non-IoT projects. Instead, it corresponds to the

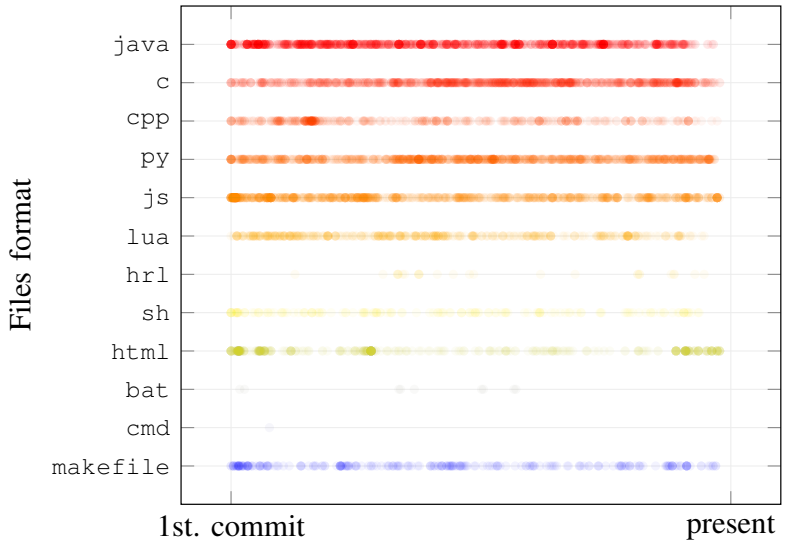
programming language whose files are modified by a higher percentage of contributors, among the repositories that were analyzed. For instance, although Go is the second programming language modified by a high percentage of contributors, it is present in just three IoT projects and five non-IoT projects, in both cases with around half of the files.

Evolution of files by programming languages: Figures 4.6a and 4.6b aim at visualizing the files modified in the commits, grouped by their format. To facilitate the interpretation, the dates of the commits, from all the analyzed projects, were normalized and placed on a common timeline since the first commit to the data extraction date. Moreover, as the modifications to the files from the analyzed projects sum up to approximately 0.6 million diffs in IoT projects, and 3 million in non-IoT projects, and larger projects have a significantly higher number of commits, it was decided to randomly sample 500 modifications, at most, per each project. In this manner, it could be guaranteed that the graph was readable and balanced concerning the represented number of modifications from each project. Otherwise, there would be so many points that it would not be possible to identify the trends, and most of them would belong to the larger projects.

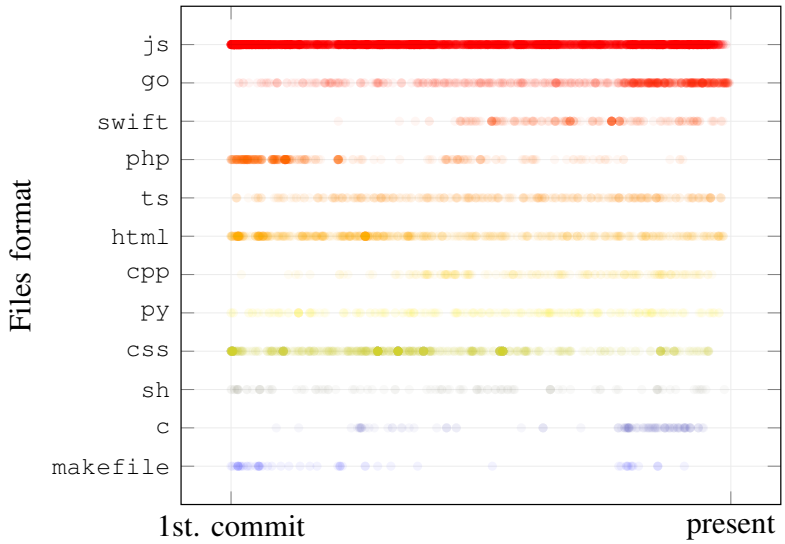
This visualization of the modifications in the commits by files format (where the formats are differentiated by random colors, and each point represents a modification of a file in the given format) allows observable trends concerning the frequency of the changes to be identified. This chart indicates that compiled and interpreted programming languages are continually modified along the IoT projects lifetime, while shell-oriented languages are rarely modified. Thus, the commits over time are consistent with the specialization trends by language (Figure 4.5), the presence of the programming languages and the primary programming languages (Figures 4.4 and 4.3). This shows that developers focus more on source code concerning the business logic of the application rather than the execution scripts.

Regarding non-IoT projects, JavaScript files are evidently the most modified over time, no matter in which project they were used (e.g., user interface frameworks, general purpose libraries, MVC frameworks, runtime engines, programming frameworks). Other types of files evolved equally, with no evident differences, across the various development phases.

RQ4.1: How developers of IoT vs. non-IoT OSS applications contribute to their projects regarding the programming languages that they adopt? IoT projects present contributions in diverse programming languages, without a unique widely used language. In IoT projects, in addition, the files modified by a higher proportion of contributors are Java, C, C++, Python, and JavaScript. Additionally, Shell executable files, Batch files, and Command files are manipulated by a percentage that reaches, on average, 15% of the contributors. The above indicates a more variegate usage of programming languages and a higher level of



(a) IoT repositories



(b) Non-IoT repositories

Figure 4.6: Commit history over time by file format

specialization in shell-oriented languages than in non-IoT projects. Concerning files' evolution over time, compiled and interpreted programming languages are continually modified along the IoT projects lifetime, while shell-oriented languages are rarely modified. This is less visible for non-IoT projects.

4.3.3 RQ4.2: Maturity of the IoT Software Ecosystem

The maturity of the IoT software ecosystem was investigated by exploring the dependencies of each project and identifying how many they are and which ones are present in the various projects. Initially, the GitHub API was used to extract the data about dependencies. However, in this case, the data provided by the API is not completely accurate because GitHub is not able to identify the dependencies of a project if they are not defined in one of the supported manifest file types[4]. Moreover, these manifests are limited to a reduced set of supported languages, namely Java, JavaScript, .NET, Python, and Ruby. For this reason, each project had to be manually explored looking for the files where dependencies are specified along with their versions.

When manually looking for the dependencies, the first step was trying to find the equivalent to the manifest file in the project root directory. If such a manifest did not exist, the content of the files was examined using the GitHub search engine, looking for keywords that could help us to identify the files in which dependencies could have been declared. Concretely, the query keywords were: `dependencies`, `deps`, `dev-deps`, `import`, `include`, `require`. Furthermore, the substring "`github.com/`" was also used as a query keyword to identify the dependency's corresponding repository on GitHub. In that case, the search could highlight the URL within GitHub of the declared dependencies. Unfortunately, this strategy was not always effective, particularly in the largest projects where the query retrieved thousands of source code files, most of which contained the keywords inside documentation blocks. When it was possible to find one or more dependencies, they were added to the data gathered with the GitHub API; otherwise, it was assumed that the project under analysis did not have any explicit dependency.

Afterwards, the API data and the data gathered manually were consolidated, and the analysis was performed taking into account two conditions: (i) dependencies had to correspond to open source software projects so that they could explored and analyzed, (ii) the dependencies declared directly in the analyzed project, only, were included: *dependencies of the dependencies* were excluded from the analysis. Consequently, the number reported in the *# Dependencies* column in Tables 4.1 and 4.2, corresponds to the number of dependencies that could be correctly identified either via the API or manually, and that satisfy the just described conditions. For this reason, it must be clarified that zero dependencies reported in the table do not necessarily imply that, in practice, the concerned project does not have any dependencies at all.

Regarding the number of dependencies, it could be observed that developers

[4] https://help.github.com/articles/listing-the-packages-that-a-repository-depends-on/, last visited on June 6, 2019

of non-IoT projects adopt more dependencies than those working on IoT projects. Specifically, IoT projects exhibited 1,084 dependencies, compared to 1,868 dependencies for non-IoT projects (1.7x). In addition, the number of dependencies shared among different repositories is significantly higher in non-IoT projects. Accordingly, Figure 4.7 shows the percentage of dependencies present in a given number of projects. In both cases, the majority of the dependencies are not shared, but while in the non-IoT projects the percentage of dependencies shared by 2 or more projects is approximately 35%, in IoT projects is around 5%.

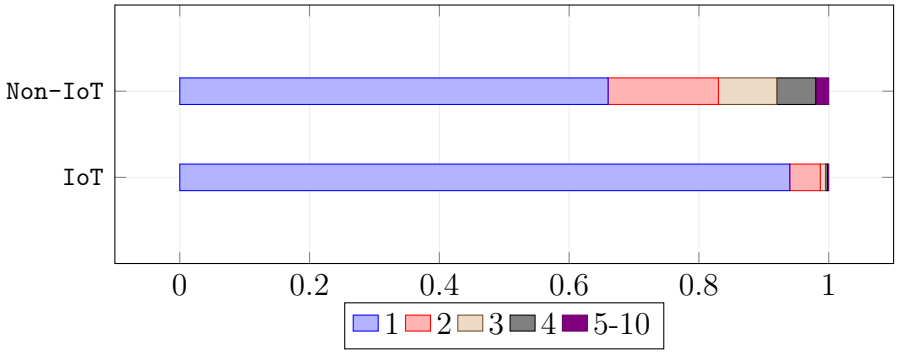


Figure 4.7: Distribution of dependencies present in one or more projects

Finally, Tables 4.3 and 4.4 present the list of the top-15 most popular dependencies among IoT and non-IoT projects, respectively. By analyzing the type of the dependencies, it can be highlighted that most of the dependencies of non-IoT projects correspond to *utilities* aimed at easing code development, such as parsers, test frameworks, beautifiers, and algorithm implementations. In the IoT projects, instead, some of the most popular dependencies concern network protocols client libraries, HTTP requests libraries, a serial port access library, and a test framework. A few dependencies were common across IoT and non-IoT projects, and they are utilities mainly concerning code source code formatting, linting, and testing.

RQ4.2: How developers exploit dependencies to reuse features in IoT vs. non-IoT OSS projects? Non-IoT projects have more dependencies than IoT projects (1.7x). Moreover, the number of shared dependencies is significantly higher for non-IoT projects. Although in both of them, IoT and non-IoT projects, most of the dependencies were not shared among different projects, in non-IoT projects the percentage of dependencies shared by 2 or more projects is approximately 35%, while in IoT projects is around 5%. Finally, the most popular dependencies in the analyzed IoT projects were shared at most by 5 projects, and among these popular dependencies, there were network protocols client libraries, HTTP requests libraries, a serial port access library, and a test framework. Among the most popular

Table 4.3: Most popular dependencies of IoT projects

Dependency	# Repos.	Description
mochajs/mocha	5	Test framework for nodejs
petkaantonov/bluebird	4	Promise library for nodejs
eslint/eslint	4	Linting utility for JavaScript
gruntjs/grunt	3	JavaScript task runner
substack/minimist	3	JavaScript argument parser
substack/node-mkdirp	3	Recursively `mkdir` for nodejs
kelektiv/node-uuid	3	RFC UUIDS generator
request/request	3	HTTP request client
sinonjs/sinon	3	Test framework for JavaScript
shama/gaze	2	File system watcher wrapper
jashkenas/underscore	2	Utility library for JavaScript
eclipse/paho.mqtt.python	2	MQTT Python client library
requests/requests	2	HTTP library for Python
pyserial/pyserial	2	Serial port access library
numpy/numpy	2	Scientific computing package

Table 4.4: Most popular dependencies of non-IoT projects

Dependency	# Repos.	Description
isaacs/node-glob	11	glob implementation in JavaScript
eslint/eslint	10	Linting utility for JavaScript
isaacs/rimraf	10	`rm -rf` utility for nodejs
chalk/chalk	9	Terminal string styling utility
lodash/lodash	9	JavaScript utility library
Microsoft/TypeScript	8	Superset of JavaScript
substack/minimist	8	JavaScript argument parser
substack/node-mkdirp	8	Recursively `mkdir` for nodejs
npm/node-semver	8	Semantic versioner for npm
acornjs/acorn	7	JavaScript parser
rollup/rollup	6	Module bundler for JavaScript
sinonjs/sinon	6	Test framework for JavaScript
mishoo/UglifyJS2	6	JavaScript beautifier toolkit
mathiasbynens/he	6	HTML entity encoder/decoder
browserify/resolve	6	`require.resolve()` implementation

non-IoT projects, instead, dependencies mainly concerned utilities aimed at easing code development.

4.4 Discussion and Implications

After presenting the results of the analysis, this section focuses on (i) a discussion of the results and on (ii) an analysis of the implication that the current research work has both for researchers and practitioners.

4.4.1 Discussion

The results that emerged from the analysis showed some points to be further highlighted and discussed, detailed below.

The development of IoT applications is different. While the knowledge about an inherent complexity in developing IoT applications was already hinted in the literature (e.g., [42, 94, 60]), in this research this complexity was evaluated in a more quantitative way. It was observed that developers, involved in the creation of IoT vs. non-IoT software applications, are less oriented towards the adoption of a lead programming language, but they work with different programming languages, according to the task at hand or to the specific capability of the infrastructure (e.g., a micro-controller or a cloud service) where the IoT application should be deployed. Furthermore, this heterogeneity of languages is also reflected in the IoT projects' topics, thus unveiling one of the main sources of complexity in IoT applications development, i.e., the co-existence of various kinds of devices, protocols, and architectures within the same application. Therefore, the tools and methodologies to support IoT developers can not be constrained to a given technological stack. They should be language and platform agnostic, or at least open to support or integrate different ecosystems.

Specialization of a few contributors towards command-line scripting. The percentage of contributors that modified specific files and the tracking of the commits over the lifetime of IoT projects showed that a strong majority of the developers are frequently modifying the files written in compiled and interpreted programming languages, where the business logic of the application reside, while a few contributors specialize in shell-oriented languages (e.g., bash), generally related to the configuration and deployment of the software components in a particular execution environment. Indeed, differently from non-IoT projects, shell-oriented languages are present in most of the IoT projects. This result reveals that, in IoT projects, the execution environment is particularly relevant yet problematic for what concerns the different (and often incompatible) target devices.

The way files evolve is different. By observing the evolution of the files during the history of software projects, it could be determined that IoT developers focus more on compiled and interpreted programming languages (i.e., Java, C, C++, Python, and JavaScript) able to fulfill the core business logic of the IoT application. All these files evolved equally across the various development phases, while shell-oriented files are scarcely modified. IoT developers seems not to focus on configuration and deployment scripts, probably immutable once the target platform(s) is chosen. Conversely, non-IoT developers constantly and significantly evolve the JavaScript files of their applications, only, being they user interface frameworks, general purpose libraries, MVC frameworks, runtime engines, or programming frameworks. Other types of files evolved equally, with no evident stops, across the various development phases.

Dependencies are considered differently. Non-IoT projects have more dependencies than IoT projects, and 35% of those dependencies are shared among 2 or more non-IoT projects. IoT developers do not only use less dependencies, but such dependencies are also shared among fewer projects, with only 5% of them shared by two or more repositories. However, dependencies in non-IoT projects mainly represent utilities, while dependencies in IoT projects are more varied and oriented towards software integration tasks. The relatively high number of dependencies used by IoT projects may entail a relatively good maturity of the IoT ecosystem, but the analysis also highlight some issues in sharing the knowledge about the existence of a given dependency.

4.4.2 Implications

The aforementioned findings have a number of implications for researchers and practitioners. Researchers should acknowledge the specificity of this domain, and explicitly consider *IoT-oriented* software engineering as a study branch. More specifically:

IoT-oriented tools and methodologies. Given the wide heterogeneity of IoT applications and adopted programming languages, stemming from both the results and the literature, tools like IDEs and software methodologies to support IoT developers should be language and platform agnostic, and not constrained to any given technological stack. In addition, research could focus on ways to abstract this heterogeneity, to allow developers to more easily share their IoT-related efforts, code, and documentation.

Supporting automation for multiple and diverse deployment targets. The specialization towards shell-oriented languages and their relative immutability, generally related to the configuration and deployment of the software components in a particular execution environment or embedded device, may indicate that execution environments are particularly relevant for IoT development. Research efforts

should consider approaches to deal with this devices heterogeneity and to automate the generation and execution of deployment commands across several, often incompatible, devices.

IoT-specific dependencies sharing mechanisms. The obtained results showed that developers exploit some existing dependencies in their projects, but the same projects do not present common dependencies. Likely, this is due both to the heterogeneity of the IoT projects and to the relatively new and not yet consolidated software community behind those projects. This represents an opportunity for researchers for the definition of novel mechanisms that IoT developers can adopt to make their code more extensible, modular, and reusable, given the peculiarities of the deployment platforms.

Furthermore, practitioners need to find appropriate ways to handle and share dependencies, as well as to create a more focused software community around these topics. Finally, confirming previous insights in the literature (including those presented in Chapter 3), the results from this study suggest that IoT software development requires skills and expertise in several and disparate domains, differently from those required by the development of traditional software. Developers are indeed called to be more creative and able to adapt to different contexts and programming environments. Thus, it would be beneficial for students to have dedicated courses (e.g., similar to the courses reported in [23]) where they could gather these skills to approach the development of IoT applications.

4.5 Threats to Validity

Sample validity: The selection criteria of the analyzed projects aimed to be as neutral as possible from the researchers' appreciations. For this reason, their number of stars was the only criteria to prioritize them and take the 60 top starred ones. Additionally, their IoT and non-IoT nature were determined by the topics that the project owners assigned them. Since tags are freely added by project owners, this might have excluded some potentially interesting IoT projects from the analysis. The only two interventions of my criteria consisted of excluding projects that were not software related or without an open source license. Nevertheless this selection procedure, unintentionally, resulted in a strong shift in the non-IoT projects towards web-related frameworks. However, this selection criteria was kept because, on the one hand, it was replicable and transparent, and on the other hand, it reveals GitHub users trends about their interests.

On the other hand, the inclusion of the most starred projects spontaneously resulted in a significant number of files, commits, and an active contributors community. According to Kalliamvakou *et al.* [52], these variables help to avoid *perils* while performing software engineering research on GitHub. Moreover, inspiration was strongly taken from the methodology adopted by Pascarella *et al.* [70]. Authors

included the same number of projects in their comparative analysis of video games and non-video games OSS projects.

File classification validity: The percentage of programming language on each project was calculated relying on the statistics provided by the GitHub API. As already mentioned, this measure is calculated by GitHub using the open-source Linguist library, which was assumed to provide accurate statistics. However, the accuracy of such statistics could be assessed later when computing the percentage of contributors working on a given programming language. Each project was locally cloned and, with a text mining tool developed internally, all the commits were processed to extract the files modified by each contributor. It emerged that the results delivered by my text-mining tool were consistent with the percentages retrieved through the API.

Dependencies identification: The GitHub API retrieves the number and list of dependencies if they are defined in one of the supported manifest file types, only. These types are only attached to Java, JavaScript, .NET, Python, and Ruby projects. Therefore, to avoid inconsistencies in the analysis of ecosystem maturity, each project had to be explored manually looking for the files where software dependencies and their versions are specified. This manual process, given its complexity, could have led to omissions or mistakes in the identification of the dependencies.

Finally, the higher number of dependencies in the non-IoT projects could depend from the nature of these projects: they are homogeneously distributed in web development and a large number of them have the same primary language (i.e., JavaScript). Given these conditions, it is logical that non-IoT projects share more dependencies among them than IoT projects, which are more heterogeneous.

4.6 Conclusion

IoT software development is known to differ from the development of other kinds of applications. It poses several challenges and requires expertise in various areas due to the diverse features that IoT applications expose. The research presented in this chapter provided empirical insights into the peculiarities of IoT software development through the analysis of OSS projects. This analysis was structured around two criteria: the behavior of the contributors, and the maturity of the IoT software development ecosystem. Specifically, exploratory study was conducted mining 30 popular IoT OSS and 30 popular non-IoT OSS projects available on GitHub. The obtained results are intended to provide evidence about IoT development characteristics (such as the distribution of programming languages, the specialization of contributors, the evolution of the files, and the adopted dependencies), that should be considered by future research efforts aimed at better satisfying software engineering needs in the IoT scenario.

Chapter 5

Code Recipes: a documentation approach for easing IoT development

The IoT developers survey, presented in Chapter 3, focused on identifying the most challenging issues that novice programmers experience when developing IoT systems. The results suggested that the integration of heterogeneous software components resulted one of the most painful issues, mainly due to the lack of documentation understandable by inexperienced developers, from both conceptual and technical perspectives. In fact, novice programmers devote a significant effort looking for documentation and code samples willing to understand them conceptually, or in the worst case, at least to make them work. Driven by the research question: "How do the lessons learned by IoT novice programmers can be captured, so they become an asset for other novice developers?", this Chapter presents the proposal of *Code Recipes*. They consist of summarized and well-defined documentation modules, independent from programming languages or run-time environments, by which non-expert programmers can smoothly become familiar with source code, written by other developers that faced similar issues. Furthermore, a use case is presented to illustrate how *Code Recipes* are a feasible mechanism to support novice IoT programmers in building their IoT systems. Part of the work described in this chapter has been previously published in [25].

5.1 Motivation

As described in Chapter 3, an exploratory study was conducted among Electronic and Computer Engineering undergraduate students of a university course. The goal of the study was to identify the pain points that novice programmers experienced when developing IoT systems [26]

The obtained results from this exploratory study suggested that the integration of heterogeneous software components is one of the most painful issues. It commonly implies dealing with several protocols, formats, and authentication mechanisms, that are usually unknown to the students. Moreover, the lack of clear and complete documentation, or merely, the absence of documentation that can be understood by a novice developer, make this integration issue even more difficult to overcome.

In view of this issue, and looking for solutions to support novice IoT developers in overcoming these integration issues, it was noticed that despite the specificity of each project, implementations of the integration between software components were similar across most of them, especially when third-party services were involved. However, although the source code of the projects from the past years' courses was on GitHub, it was not being reused among groups in later versions of the course. Therefore, the lessons learned by a group when implementing its project was not useful for the next year's groups.

Taking into account the results of the exploratory study and the lack of code reuse between the course groups, it was envisioned that the the solutions found by the students, that were finally included in the working prototype built at the end of the course, could become a valuable asset for the novices that are about to start implementing their projects. The source code of these prototypes reveals architectural decisions and strategies adopted by other groups to achieve the integration of diverse software components. This code should, therefore, provide some guidance to other programmers that are in the process of overcoming the same learning curve issues. Moreover, if documented, this code would be a solution to the reported lack of documentation understandable by inexperienced developers [102]. In fact, being able to observe how someone else coded, what others paid attention to, and how they solved problems all support learning better ways to code and access to superior knowledge [33].

Driven by the research question: "How do the lessons learned by IoT novice programmers can be captured, so they become an asset for other novice developers?", the current proposal aimed at easing the learning curve to IoT novice developers, not by automating code reusing and hiding the code from the developers, but instead, by enabling non-expert programmers to easily become familiar with source code, written by other developers that faced similar issues.

5.2 Use Case

To illustrate my proposed solution, I selected a use case representative of the difficulties identified in the IoT developers survey (Chapter 3). The survey suggested that among the most challenging issues novices face when developing IoT systems, the integration with other software components was perceived by many

students as the most painful issue. In particular, the integration with third-party APIs that require OAuth 2.0 authentication was a time-consuming and difficult task. The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. Broadly speaking, this authentication protocol consists of a flow, with a set of roles (*resource owner, resource server, client, and authorization server*) interacting across various steps (*authorization request, access token request, and protected resource request*), and exchanging several resources (*authorization grant, access token, refresh token, redirect URI*).

In the development of IoT systems, OAuth authentication protocol becomes fundamental since most of the third party service APIs use it. The integration with the Fitbit activity tracker[1] is a concrete example of the OAuth protocol usage. In order to gather the data captured by this wearable device, the third party application (i.e., the one developed by the novices) must obtain users authorization through the OAuth protocol.

However, due to the roles, steps, and resources that the protocol comprises, the adoption of the OAuth authentication is not trivial. The appropriate implementation of this protocol requires a clear understanding of the various steps, both from the conceptual and the technical perspective. Novice programmers struggle considerably with the adoption of OAuth, mainly due to the lack of documentation that might be understandable by non-expert programmers.

Fitbit, for instance, has a documentation website[2] that provides guidance about the Web API for accessing data from Fitbit activity trackers. Although the developer's site has an API explorer built in Swagger, and an API debug tool, it does not provide a fully implemented functional source code sample. Moreover, despite the clarity, readability and good overall structure of the documentation, it is targeted at experienced programmers, as with most of the developer's documentation.

In this scenario, novice programmers are required to search code samples, willing to understand them conceptually, or in the worst case, at least to make them work. Typically, this involves the reference of the Google OAuth Client Library documentation, the Fitbit developers website, several posts published in Stack Overflow, and various code samples available on GitHub. Hence, from the experience of the novice programmers adopting the OAuth protocol, it could be observed that: (i) a significant amount of effort is devoted looking for documentation and samples; (ii) just through the source code it is not possible to understand the whole learning process

[1] *Fitbit*, accessed October 6, 2017, https://www.fitbit.com

[2] *Fitbit Web API*, accessed October 6, 2017, https://dev.fitbit.com/reference/web-api/quickstart/

behind it; (iii) the code fragments must be surrounded by summarized, structured and well-defined documentation modules, so they become an asset for other IoT novice programmers.

5.3 Code Recipes

Code Recipes aim at capturing the most important information and documentation about one or more code fragments. They are specified through a set of metadata and consist of multiple code fragments along with documentation and links to ease the understanding of such code, in order to implement a given integration between subsystems of an IoT system. The joint presence of metadata and links allows novice developers to explore alternative solutions and, at their will, deepen their knowledge about a specific IoT subsystem, thus contributing to their learning process.

The current approach lies in the fact that code examples, when used effectively, can be a powerful learning resource [46]. However, while examples are a valuable resource for programmers, the rich context surrounding them is often crucial for adaptation and integration [67]. This proposal enables the integration of several software components through code fragments that might belong to different programming languages and might be deployed across various run-time environments, as it is common in IoT systems. The decoupling between the recipes and the technological stack is fundamental given the heterogeneity of the software components that are involved in an IoT system. *Code Recipes*, therefore, are defined as summarized and well-defined documentation modules, independent from programming languages or run-time environments.

By defining *Code Recipes* as documentation modules structured around code fragments, they can be incorporated in various kinds of tools that might handle them in the learning process, e.g., a wiki-style web application or an IDE extension.

Code Recipes, therefore, expose four features:

- Although the *Recipes* are structured around source code fragments, they are much more than just code. They encompass information that, besides providing technical solutions, includes comments and documentation sources that account for the learning process that other novice IoT developers followed and the decisions they made to reach a solution.

- *Recipes* are not constrained to a specific architecture, programming language or run-time environment. This means, first, that this proposal is aware of the heterogeneous nature of IoT environments, and second, that is suitable to be used in multiple scenarios with IoT novice developers.

- *Recipes* are not isolated from each other, they are cross-linked on the basis of three criteria: alternative versions, other language versions, and related

recipes. This feature enables the sharing of diverse learning experiences with their commonalities and their divergences.

- Technical speaking, a structured representation (e.g., in JSON or XML) of the *Code Recipes* enables the implementation of various kinds of tools that might handle them. For instance, a web application (as shown in Figure 5.1), a web browser extension, or an IDE plugin.

```
1  {
2      "id": "1506954092",
3      "author": [{
4          "name": "Juan Saenz"
5      }],
6      "date": "21.9.2017",
7      "name": "Integration between Fitbit and Java",
8      "description": "Recipe to consume the Fitbit API using OAuth 2.0",
9      "tags": ["fitbit", "java", "oauth 2.0", "api"],
10     "running_environment": "Server application built in Java",
11     "endpoints": ["Fitbit API"],
12     "ingredients" : [{
13         "name": "Fitbit account",
14         "description": "Fitbit accounts set up for read/write API access",
15         "url": "https://dev.fitbit.com/"}],
16     "dependencies" : [{
17         "name": "Maven",
18         "description": "Maven plugin for Eclipse installed",
19         "url": "http://www.eclipse.org/m2e/"}],
20     "code_fragments": [{
21         "programming_language": "Java",
22         "description": "This is the main class",
23         "documentation_urls": ["https://github.com/google-oauth-client"],
24         "name": "FitbitSample",
25         "source_code_url": "./1506954092/FitbitSample.java",
26         "ide": "Eclipse Neon",
27         "parameters": [{
28             "name": "SCOPE",
29             "description": "OAuth 2.0 permission for resources",
30             "data_type": "String",
31             "sample_value": "activity, heartrate, location, nutrition"
32         }]
33     }],
34     "documentation_urls": ["https://stackoverflow.com/quest/9863836"],
35     "rating": "4.6",
36     "alternative_versions": ["1506957773", "1507562564"],
37     "other_languages_versions": ["1496761597"],
38     "related_recipes": ["1507302404"]
39 }
```

Listing 5.1: Code Recipe Sample

Listing 5.1 describes a possible structure of a *Code Recipe* in JSON format. First, each recipe is described through an **id** (timestamp), its **author name**, **publication date**, **name**, **description**, and **tags** (lines 2 to 9). Then, the subsystems that the recipe integrates are specified in the **endpoints** fields (lines 10 and 11). **Ingredients** (line 12) correspond to the requirements of the recipe. They can be technical requirements, such as the deployment of a specific kind of web server, or

data requirements, such as creating a developer account and issuing API client credentials. **Dependencies** (line 16) refers to requirements associated with the source code, which are fundamentally libraries and packages that must be installed.

Most importantly, *Code Recipes* include one or more **code fragments** that can be implemented in different **programming languages** and **IDEs** (lines 20 to 33). Each fragment has a set of **parameters**, which are values specific to each implementation of the recipe. Besides the source code, recipes include the documentation that their authors consulted, both for the whole recipe as well as for its fragments. They can be specified in the **documentation URLs** fields (lines 23 and 34). Finally, *Code Recipes* can be linked to each other in three ways (lines 36 to 38): **alternative versions**, that point to other recipes targeted at implementing the same integration; **other language versions**, that point to implementations of the same recipe in other programming languages; and **related recipes**, that correspond to other recipes that can be used as intermediate steps to implement the concerned recipe.

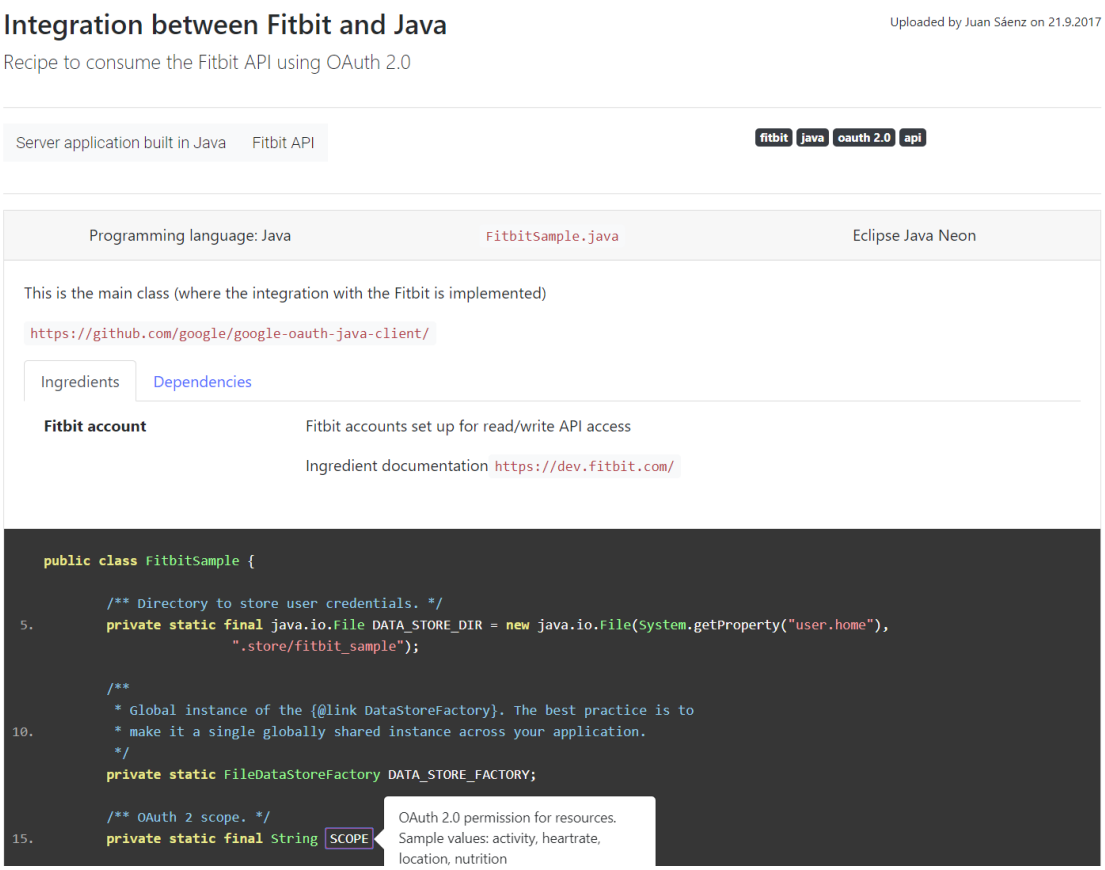


Figure 5.1: Code Recipe visualized in a web interface

5.4 Validation: The Fitbit OAuth Code Recipe

With the use case described in Section 5.2 in mind, a *Code Recipe* was developed to illustrate how this approach might help novices to overcome integration issues through a collaborative approach. To develop this recipe, a simple Java application to gather data from a Fitbit bracelet had to be implemented.

As mentioned before, no sample projects are provided in the Fitbit developers' website. Therefore, the first endeavor was to find a sample project in which the OAuth authentication was implemented using Java. After googling "*OAuth 2.0 Java Sample Code*", the second result took us to the documentation of the Google OAuth Client Library for Java (in the *Code Recipes*, this website would be included in the **`documentation_urls`** field). This website had setup instructions for Maven, the list of libraries that were required (in the *Code Recipes* would correspond to the **`dependencies`** field), the release notes of these libraries, and one sample code of the integration between a Java application and the Dailymotion API[3], using OAuth 2.0.

Once the sample code is downloaded and imported into the IDE, the next step was to install and configure Maven, including the Project Object Model (POM) in which the dependencies of the project were defined. Later, when the Java project was already compilable, the next task was to identify which pieces of the code should be modified to achieve the integration with the Fitbit API (in the *Code Recipes*, these pieces are specified in the **`parameters`** field). Among the new data that had to be inserted into the code as parameters, there were the API key, the API secret, the Callback URL and the Scope. All of this data was obtained after completing the registration as a Fitbit developer (in the *Code Recipes* this registration accounts as an **`ingredient`**)

Afterwards, there was the source code itself. It consisted of three Java classes, two of which had to be parameterized. The explanation of the meaning of every parameter was available in the Fitbit developers website, along with their possible values (in the *Code Recipes*, these **`parameters`** can be documented through a **`description`**, their **`data_type`**, and a set of **`sample_values`**). Since this was the first *Recipe* that was developed, there were no other *Recipes* to link.

Across the whole implementation process, several documentation sources were consulted. The Google OAuth Client Library documentation, the Fitbit developers website, several posts published in Stack Overflow, and various code samples available in GitHub. Notwithstanding the fact that the *Code Recipe* was developed by an experienced programmer, its implementation was not trivial, and many of the issues expressed by the novices in our previous research were highlighted.

[3] *Dailymotion Developers - API*, accessed October 6, 2017, https://developer.dailymotion.com/

Regarding the first research goal of this dissertation, the Code Recipes are a strategy targeted at helping IoT developers to overcome the lack of documentation understandable by novices, especially concerning the integration of heterogeneous subsystems. This research work focused on proposing a mechanism through which source code written by inexperienced developers could become an asset for other novices facing the same issues. To that end, the code fragments were surrounded by a set of metadata fields that allow such code to be explained thoroughly, linked to the documentation sources that account for the learning process, and linked to alternative or related versions of the concerned fragment.

Nevertheless, while the Code Recipes were proposed mainly as a conceptual model, it is critical as a future research work to integrate these documentation modules into a development tool. To some extent, the research that will be presented in the following Chapter (that relies on computational notebooks to satisfy the lack of documentation, and is not tied to a specific architecture, programming language, or run-time environment), represent a viable alternative. However, given that technically speaking, the recipes are a structured representation, they can also be incorporated in various kinds of tools that might handle them.

5.5 Conclusion

In view of the complexity that the development of IoT systems poses, particularly concerning the integration of heterogeneous software components, and taking into account the lack of documentation reported by novice programmers in my previous research, this chapter presented *Code Recipes*. They are summarized and well-defined documentation modules, non-dependent from programming languages or run-time environments, and structured around the code fragments that are required to implement some portions of an IoT system. This approach was aimed at supporting novice IoT programmers, enabling them to easily become familiar with source code written by other developers that faced similar issues.

Since Code Recipes were proposed mainly as a conceptual model, future work will concern (i) the development (or adaptation of a currently available) tool, by which novice programmers can deal with the creation of a *Code Recipes* catalog; and (ii) the subsequent validation in the context of the course. In this regard, as computational notebooks create narratives that consolidate text, executable code, and visualizations in a single document, they represent, as a tool, a feasible alternative to support the Code Recipes conceptual model. Indeed, the research that will be presented in the following Chapter partially constitutes a viable alternative to represent and validate the *Recipes* approach. In particular, the *Code Recipes* catalog might correspond to a set of notebook documents, and the computational notebook might be the web tool through which novice IoT programmers can easily

manage those documentation modules created by and for themselves, and structured around executable code fragments.

Chapter 6

A literate computing approach to support IoT prototyping

Computational notebooks create narratives that consolidate text, executable code, and visualizations in a single document. They are widely used in data science, enabling data scientists to accurately document and execute the steps of their analyses in an exploratory and iterative manner. Prototyping IoT systems is complex as well, because of IoT heterogeneous and interconnected nature. Indeed, IoT system prototyping spans across multiple development and execution environments and developers, besides focusing on the code, are required to configure diverse devices.

With the goal of ascertaining if and how computational notebooks' capabilities might be useful in the IoT scenario, this chapter presents an IoT-tailored notebook approach aimed at helping developers to build and share a computational narrative around the prototyping of IoT systems. Specifically, in this work I propose the concept of "IoT notebook", for which were analyzed its required features and was developed a prototype implementation. Finally, the proposal was evaluated by describing a use case in which a preliminary version of the IoT notebook was used for prototyping a realistic IoT system.

Part of the work described in this chapter has been previously published in [27], where a preliminary assessment of the idea, suitability, and limitations of current computational notebooks to support the development of an IoT system, is presented. Furthermore, an article providing an accurate definition and prioritization of the features, the design of the architecture, and the description of a clean implementation of the top-prioritized features, is ready for submission.

6.1 Motivation

Since the development of IoT systems requires an unusually broad spectrum of design and development technologies and skills [97] that span across multiple development and execution environments, besides focusing exclusively on the code, IoT developers are also required to deal with the hardware implementation and distributed computing concepts. Consequently, due to this inherent complexity, it is common to prototype parts of the IoT system, to explore and validate possible strategies useful to configure, develop, and integrate hardware and software artifacts. However, this prototyping process comprises several steps along which the IoT developer gradually overcomes a learning curve, while iteratively exploring and assessing various alternatives.

In addition, IoT developers struggle with three challenges: first, the programming tools for IoT development are largely the same ones used for mobile and web application development [94], and there is a shortage of software development environments that would allow an IoT developer to write a single IoT application capable of running on various type of devices [88]; second, the absence of documentation written and shared by and for non-expert developers [29, 26]; and finally, among the currently available documentation, the lack of contextual information, such as a textual description of how the code works, crucial for understanding how to configure or adapt this code to the developers' specific needs [67]. Against these three challenges, it was envisioned that IoT developers would greatly benefit from an interactive computing tool to document, execute, and share the configuration and programming steps over diverse execution and development environments, which according to the results of described in Chapter 3, would represent an alternative to the reported lack of documentation understandable by novice IoT developers.

In this regard, computational notebooks are interactive computing tools designed to support the construction and sharing of computational narratives by consolidating text, executable code, and visualizations in a single document [54, 84]. In the light of the architectural elements present in IoT systems and the features provided by current computational notebooks, this work aimed at designing and prototyping an IoT-tailored notebook that would represent a feasible environment to support IoT systems prototyping.

6.2 Literate computing

This section briefly introduces literate programming, a paradigm in which a computer program is given an explanation of its logic in a natural language interspersed with snippets of macros and traditional source code. Later, some basic concepts about computational notebooks are defined.

6.2.1 Computational notebooks

Literate programming originates in 1984 from a paper by Donald Knuth [55]. He suggests software developers that *"instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do"* [55]. In line with this proposal, literate computing tools such as computational notebooks have emerged as a mean to support the construction and sharing of computational narratives by enabling data analysts to arrange code, visualizations, and text in a computational narrative [54, 84]. These cells are linearly arranged but can be reorganized, reshuffled, and executed in any order. Moreover, programmers can pick and choose which code cells they would like to edit and run [53].

These literate computing tools are based on cells containing rich text or executable code that generates results or visualizations. Notwithstanding, notebooks also have limitations: (i) saving application states is difficult, limiting the ability to develop applications from within a notebook, (ii) real-time collaboration is, at best, limited to text editing, and (iii) the behavior of a notebook cannot be reprogrammed or extended from within, limiting its expressive power [53].

Among the currently available computational notebooks, Project Jupyter is one of the most widely used platforms [84], it is a popular open-source computational notebook that relies on open standards and enables users to combine code, visualizations, and text in a single document (a `.ipynb` file) whose underlying structure is JSON [84]. Jupyter Notebook originated from IPython [73] and, in addition to Python, it natively supports a variety of programming languages, such as Julia, R, Javascript, and C [75]. The popularity of Jupyter Notebooks increased since 2015 when GitHub began to natively render them, presenting the `.ipynb` files as fully rendered notebook documents, rather than displaying the underlying JSON[1].

6.2.2 Definitions

Computational notebooks commonly refer at the same time to the interactive literate programming documents and to the software application to execute them [75]. For the sake of clarity, throughout this chapter, they will be referred as *notebook documents* and *computational notebooks*, respectively.

Notebook documents are based on cells, each of which contains rich text or code that can be executed to compute results or generate visualizations. These cells are linearly arranged but can be reorganized, reshuffled, and executed multiple times in any order. Moreover, programmers can choose which code cells they would like to edit and run [53], and their execution does not require cleaning the outputs of

[1]Given its wide adoption and open standards it was taken as a reference upon which to build the current proposal

previous executions. Thus, an executed notebook may contain retrospective data of multiple executions.

Computational notebooks typically consists of a kernel that executes the code cells in a particular programming language and returns the corresponding output to the user, and an interactive computing protocol that standardizes and manages the communication between the *notebook documents* and the kernels. A *kernel* for its part, is a "computational engine" that executes the code contained in the code cells of a Notebook document. When the notebook document is executed, the kernel performs the computation and produces the results [47]. Each kernel executes a given programming language.

6.3 Use Case

This section describes a use case concerning a maker-level IoT system, aimed at providing a common understanding of the four architectural elements present in IoT systems (devices, gateways, cloud services, and applications), as well as at identifying the characteristics of IoT systems.

6.3.1 Controlling Philips Hue Lamps from an Arduino

The use case concerns an IoT system that warns the occupants of a room when a harmful level of carbon dioxide (CO_2) is reached by turning on a Philips Hue lamp. As illustrated in Figure 6.1, this system comprises an air quality sensor, an Arduino single-board microcontroller, a Back-end application, a Philips Hue bridge, and a Philips Hue lamp.

Specifically, the Arduino gathers and evaluates the readings from the air quality sensor. If these measures exceed a certain threshold, meaning that the level of CO_2 has become harmful, the Arduino communicates it to the Back-end application deployed on the cloud, through an HTTP request. For its part, the Back-end application communicates, through an HTTP request, with the Philips Hue Bridge, that sets the lighting color and intensity of the Philips Hue Bulb to red. The communication between the sensing and acting devices with the Application and cloud services is achieved through a set of RESTful web services that the latter expose.

Technically speaking, from the perspective of the architectural elements involved in the system, the air quality sensor is physically attached to the Arduino, and it represents the sensing devices of the system. The *application* architectural element is represented by the Back-End application, which mediates the communication between the sensing and the acting device, that in this project is represented by the Philips Hue bulb, whose lighting color and intensity can be programmed. Finally, the gateway architectural element corresponds to the Philips Hue bridge that,

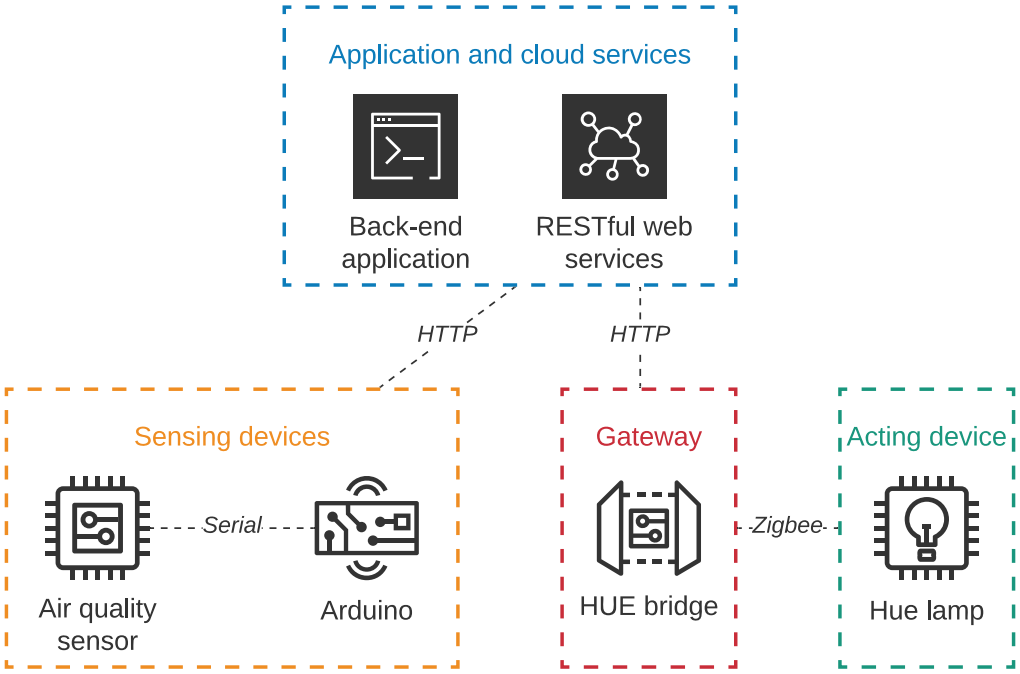


Figure 6.1: IoT architectural elements in the Use Case

through a Zigbee protocol, controls the bulb according to the requests received from the Back-end application.

Regarding the programming and deployment of the system, two heterogeneous software artifacts are present: (i) an Arduino sketch running on the microcontroller, and (ii) the Back-end application, written in Python and deployed on the cloud. Furthermore, the integration between the Arduino and the Back-end is achieved by invoking a set of web services exposed by the latter. These web services were implemented using Flask, a Python web micro-framework.

6.3.2 Characteristics of an IoT system prototype

Regarding the software artifacts, the following characteristics can be identified:

1. **They remain in background execution**. The Arduino script keeps gathering and monitoring the readings from the temperature sensor, the Back-End application keeps mediating the communication and interacting with the sensing and acting devices, and the RESTful web services keep forwarding the requests sent from the Arduino.

2. They are deployed once, and then they keep running in the background simultaneously, while **exchanging data and requests** among them.

3. They are deployed over **heterogeneous run-time environments**. Specifically, while an Arduino script is executed on a computing-resource constrained device without an operating system, the Back-End application and its web services are deployed on the cloud.

4. They involve multiple devices, therefore the prototyping of this system requires to **manage the available devices**, and for each one of them, configure the proper run-time environment, and deploy the executable files on it.

6.4 IoT notebook

This section presents the concept of an IoT-tailored notebook (hereinafter, referred to as IoT notebook) as a tool to support IoT systems prototyping. The first step was to list the set of features that an IoT notebook should offer (Section 6.4.1) based on the characteristics specified in Section 6.3.2. Then, the current IoT notebook proposal was introduced from two perspectives: first, conceptually, outlining which concepts from the identified features should be incorporated into the currently available computational notebooks (Section 6.4.2); and lastly, technically, describing the architecture that supports the implementation of the IoT notebook (Section 6.4.3).

It should be emphasized here that the IoT notebook approach is meant for prototyping, not production. It aims at enabling IoT developers to build and share a computational narrative around the process that they follow when they prototype an IoT system. Hence, IoT notebooks are not intended to support the production deployment of a fully implemented IoT application. For this reason, aspects such as the scalability and testability are out of the scope of the current proposal.

6.4.1 Features of an IoT notebook

The following is a recap of 7 **features** (**FT-1** through **FT-7**) that emerge from the previously identified characteristics.

1. Several software artifacts remain on simultaneous background execution.

    - **FT-1: Various code fragments within the code cells might be able to remain running in background simultaneously**, without blocking the execution of other code cells (as occurs in the currently available computational notebooks).

2. Software artifacts keep exchanging data and requests among them while running simultaneously

- **FT-2:** Being able to control and monitor the embedded devices is critical in IoT systems. For this reason, the IoT notebook must **support real-time communication** with the devices. It means that, just as commands and data are sent from the IoT notebook to be executed on the devices, conversely, the results of such executions should be gathered and displayed in real time in the notebook.

3. Software artifacts are deployed over heterogeneous run-time environments

    - **FT-3:** Additionally, in connection with the previous feature, the notebook should be able to **detect and identify the devices automatically**, in a plug and play manner, once they are connected, physically or over a network, to the terminal where the notebook is running.

4. Before executing the code, **several run-time environments must be configured** according to the specificities of the available devices.

    - **FT-4:** In the context of IoT prototyping, software configuration steps such as the installation of packages, the setup of a given device or the definition of the imported libraries are executed before the software programming steps that concern the development of the system's business logic. Accordingly, the IoT notebook must allow **separating the configuration steps from the development ones**. Besides clearly differentiating the nature of these steps, it would provide consistency to the notebooks.

Other than the features that emerge from the technical characteristics of IoT systems prototyping, three more features concerning the presentation of the notebook documents were identified from the literature. These features are aimed at making notebook documents more understandable and consistent with the structure of IoT systems.

- **FT-5:** To adequately structure and support the heterogeneity of IoT systems, typically involving various devices as well as back-end and end-user services and applications [60, 94, 97], the IoT notebook should **enable the grouping of various notebook documents**, depending on the architectural element to which they belong. For instance, to clearly define and represent such architectural elements, a group of notebook documents concerning the setup and development of an Arduino should be categorized as sensing devices documents, while the group of documents regarding the setup and development of the Flask RESTful web server, should be categorized as cloud service documents.

- **FT-6:** As mentioned before, while current computational notebooks present the cells in a linear top-bottom narrative, a user may choose to execute the cells in a non-linear, arbitrary order [75]. This feature can be useful in the field of data science when checking if changes to a prior analytical step impact later computations [84]. However, hidden states, out-of-order cells, hardcoded paths, and other bad practices also prevent the reproduction of notebooks [75]. Furthermore, if cells appearing at the beginning of notebooks depend on cells that appear later, it would cause several issues to users that try to execute them in the default order [57]. On the contrary, managing the dependencies of notebooks and guaranteeing the linear execution order could improve the reproducibility rate [75].

  When prototyping IoT systems, several imports may be required to link external dependencies. Unlike the iterative nature of data analysis, the IoT development process tends to be incremental. The execution of the initial steps must satisfy the conditions that are required to guarantee the successful completion of the later steps. Consequently, the IoT notebook must **enable the definition of execution order constraints** among the cells within a document, if needed.

- **FT-7:** Notebooks evolve and grow and they often become difficult to navigate or understand, discouraging sharing and reuse [85]. Since the prototyping of IoT systems might comprise large fragments of code in specific architectural components, an IoT notebook should allow this **code to be split across various cells**, so that small pieces or even single lines of code can be accurately documented while maintaining their execution as a single block. This feature would enable the elaboration of accurate and clear narratives, even in the presence of large fragments of code.

Table 6.1: IoT notebook features

ID	Features	Jupyter	Priority
FT-1	Keep executing code cells simultaneously	✖	Medium
FT-2	Support real-time communication	✖	Low
FT-3	Detect and identify the devices	✖	Medium
FT-4	Differentiate configuration and business logic steps	✔	Medium
FT-5	Group several notebook documents	✖	Low
FT-6	Enable the definition of execution order constraints	✖	Top
FT-7	Split code across various cells	✔	Low

Table 6.1 summarizes the just described IoT notebook features. The third column represents whether the Jupyter notebook and its available plugins partially

satisfy (✔) or do not satisfy (✖) the given features. Finally, each feature was assigned a priority based on the importance of the IoT prototyping feature that they support.

6.4.2 IoT notebook Conceptual Model

Figure 6.2 depicts the conceptual model of the IoT notebook. With respect to the concepts involved in the Jupyter notebook, there were introduced the concepts of *Architectural element* and *Section*. The concept of *Architectural element* enables to categorize the Notebook documents depending on whether they belong to the devices, gateways, cloud services, or applications. The concept of *Section*, on his part, enables to distinguish the cells in the *Notebook document* concerning the configuration of the given architectural element from the ones concerning the business logic development. However, apart from these two categories, the users can define any other custom category to structure their *Notebook documents*.

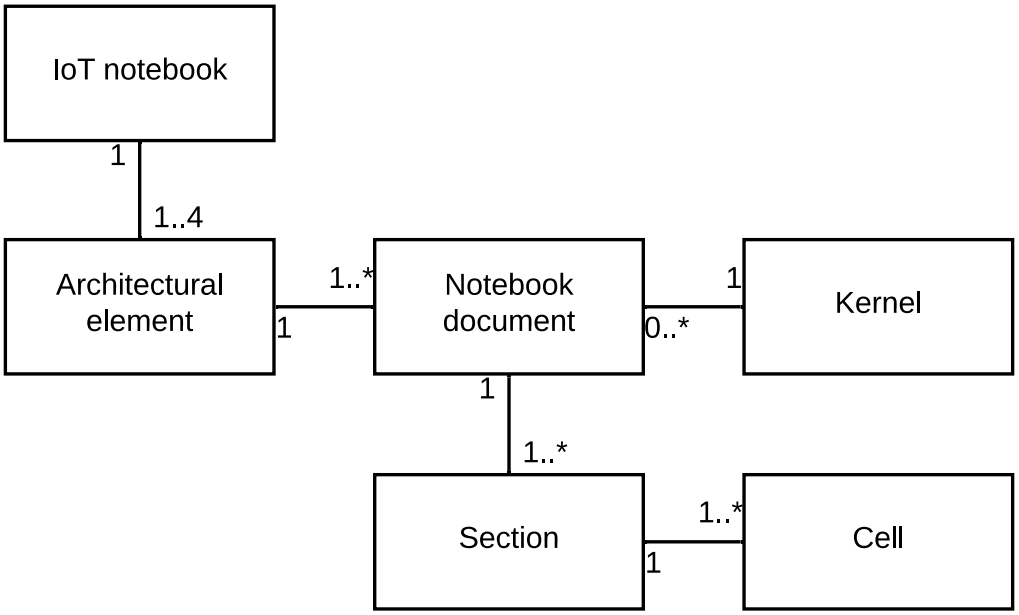


Figure 6.2: IoT notebook Conceptual Model

6.4.3 IoT notebook Architecture

As stated before, the hereby proposed architecture was studied and inspired from the architecture of Jupyter. Figure 6.3 depicts this proposed architecture, the main idea behind it is to integrate the components of a computational notebook with the concept of IoT nodes, that aim at representing and supporting the architectural elements that characterize IoT systems [94]. To that end, it was structured

around five blocks, listed below.

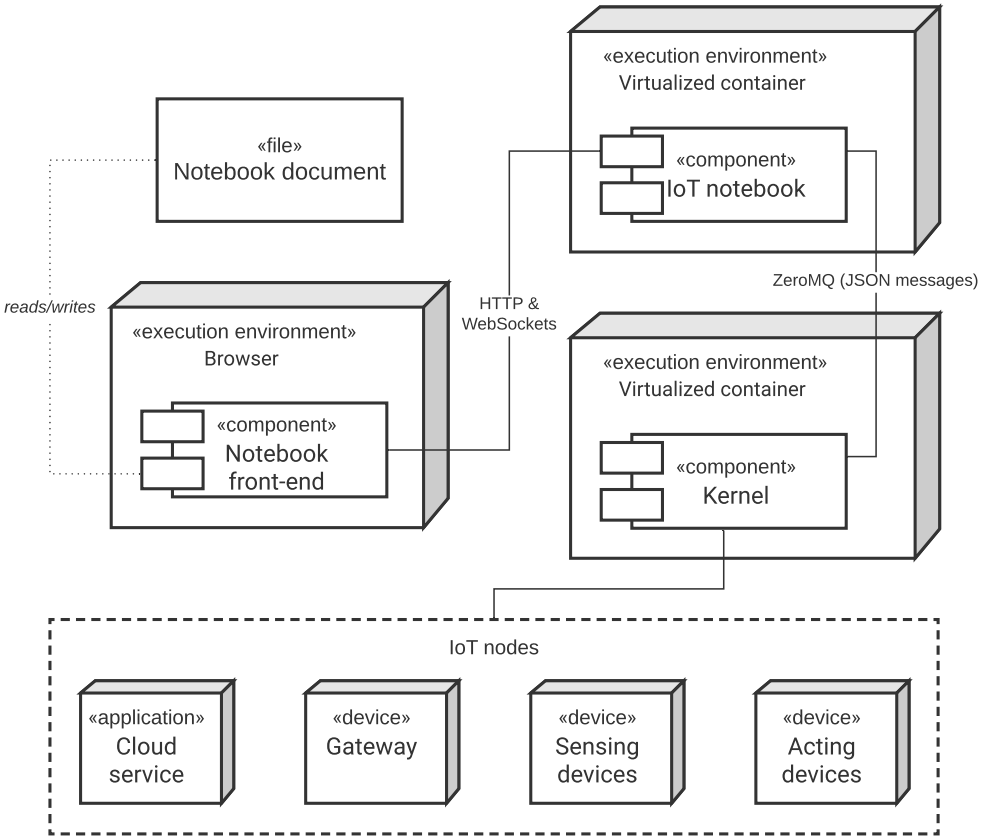


Figure 6.3: IoT notebook Architecture

Notebook documents

A *Notebook document*, technically speaking, consists of a JSON document containing text, source code, rich media output, and metadata. As shown in Listing 6.1, at the highest level, a notebook is a dictionary with a few keys: *metadata (dict)*, *nbformat (int)*, *nbformat_minor (int)*, and *cells (list)*. There are two types of cell types; markdown cells and code cells. The former ones contain source code in the language of the document's associated kernel, and a list of outputs associated with executing that code. They also have an `execution_count`, which must be an integer or null. In short, a *Notebook document* consists of a file with descriptive text cells interleaved with executable code cells.

Aiming at satisfying the previously identified features, the following modifications (highlighted with red text in Listing 6.1) are proposed over the current *Notebook documents* structure:

- Add the `architectural_element` field to enable the grouping of the Notebook documents depending on the architectural element to which they belong (**FT-5**). The four possible string values for this field are: devices, gateways, cloud services or applications.
- Introduce the `sections` list concept. Each section is characterized by its name and encompasses a list of cells. In this manner, the cells within a Notebook document can be separated into configuration and development steps (**FT-4**). Furthermore, since the name of the section can be freely assigned by the user, several sections may be defined within a Notebook document according to his particular needs.
- Add the `id` field to uniquely identify each cell and enable them to be referenced.
- Include the `background_execution` field to determine if the given cell must run in background or if the user should execute it and wait for the output (**FT-1**). The values that this field may take are true or false.
- Insert the `is_prerequisite` field to indicate if the given cell must be executed before all the subsequent cells in the Notebook document (**FT 6**). The values that this field may take are true or false.
- Include the field `linked_group_id` to enable the grouping and execution of the code split across several code cells (**FT-7**). In this manner, several cells are identified as belonging to a given group, and consequently, executed together.

Notebook front-end

The *Notebook document* is visualized, edited, and executed through the *Notebook front-end*, a web application that is accessed by the IoT developers over a browser. Apart from the features that the Jupyter front-end currently provides, the proposed IoT notebook front-end should include new user interface elements aimed at (i) Enabling users to edit and visualize the new fields of the *Notebook document* previously described; (ii) displaying in real-time the data coming from the connected devices via the custom IoT-tailored kernels (**FT-2**); and (iii) displaying the available devices identified by the IoT-tailored kernels, enabling the users to determine in which device they execute a given code cell (**FT-3**).

```
{
  "metadata": {
    "kernel_info": {
      "name": "the name of the kernel"
    },
    "language_info": {
      "name": "the programming language of the kernel",
      "version": "the version of the language",
      "codemirror_mode": "the name of the codemirror mode to use [optional]"
    }
  },
  "nbformat": 4,
  "nbformat_minor": 0,
  "architectural_element": "devices, gateways, cloud services or applications",
  "sections": [
    {
      "name": "the name of the section",
      "cells": [
        {
          "id": "unique id for each cell",
          "cell_type": "code",
          "execution_count": 1,
          "metadata": {},
          "source": "[some multi-line code]",
          "outputs": [{}],
          "background_execution": "true or false",
          "is_prerequisite": "true or false",
          "linked_group_id": "unique id for each group"
        }
      ]
    }
  ]
}
```

Listing 6.1: Notebook document JSON top structure, where the proposed modifications and additions to support IoT requirements are shown in red

IoT notebook

The proposed architecture follows a client-server pattern [9] in which the notebook back-end is deployed remotely in the server execution environment. As shown in Figure 6.3, the IoT notebook component exchanges messages with the notebook front-end. Besides satisfying the presentation client requests, the function of the IoT notebook component is to receive the code execution requests and forward them to the corresponding kernel according to the programming language of the given cell.

Kernel

In the proposed IoT notebook architecture, the IoT-tailored kernel is required to support two of the previously identified features: (i) support the real-time and bi-directional communication between the IoT notebook and the IoT nodes, particularly with the sensing and acting devices (**FT-2**); and (ii) detect the IoT notebook the available sensing and acting devices on which the code cells may be executed (**FT-3**). Additionally, the IoT-tailored kernel is also required to read and execute accordingly the new fields added to the *Notebook documents*.

IoT nodes

IoT nodes correspond to three of the four architectural elements that were previously mentioned (cloud services, gateways, sensing and acting devices).

6.5 Validation

This section reports the description of a prototype implementation of the top-priority and some of the medium-priority features from Table 6.1 (Section 6.5.1). Furthermore, the implementation was validated by showing how this resulting version of the IoT notebook may support the prototyping of the maker-level IoT system that was described in Section 6.3 (Section 6.5.3).

6.5.1 IoT notebook Implementation

Based on the previous architecture, a prototype version of the IoT notebook was implemented, including the top prioritized (**FT-6**), and two medium prioritized features (**FT-3** and **FT-4**). In the following are described the implementation of the selected features from the technical point of view.

FT-6: Define compulsory execution order

The development of this feature involves the Notebook document, the Notebook front-end, and the IoT notebook component. Concerning the Notebook document, as described in the previous section (Listing 6.1), the field `is_prerequisite` was introduced to indicate if the current cell should be executed before all the subsequent cells. This way, every time a code cell is to be executed, the IoT notebook component performs a search from the top of the document until the given cell, looking for cells marked as `is_prerequisite` that have not yet been executed, and executes them first.

Regarding the Notebook front-end, it was developed a custom plugin that places a checkbox below each cell so that the users can indicate if this cell has to be compulsorily executed before the cells further down in the Notebook document. The value field `is_prerequisite` is assigned according to whether the checkbox was clicked or not.

FT-3: Detect and identify the devices

The communication between the IoT notebook component and the IoT nodes requires the implementation of custom Kernels, able to support the execution of the code cells, whether in Cloud services, Gateway devices, or Sensing and Acting devices. As explained below, for the use case implementation, a custom *IoT-tailored* kernel was implemented to enable the communication of the IoT notebook component with an Arduino single-board microcontroller (called Kernelino). Other than supporting the execution of the code cells in the single-board microcontroller, Kernelino allows the detection of the devices physically connected to the computer where the IoT notebook executes (**FT-6**), as shown in Figure 6.5.

FT-4: Differentiate configuration and business logic steps

This feature was fully implemented within the Notebook front-end. Since in the proposed structure for the Notebook documents, the cells are grouped into sections, as shown in Listing 6.1, a custom widget was developed to display the sections and their contents inside a tabbed sidebar. In this manner, the user can decide on which tab to place the text and code cells, and even to add new sections.

The implementation of the preliminary IoT notebook with the top-priority and some of the medium-priority features enabled to "translate" and to consolidate the steps that are required to prototype an IoT system.

6.5.2 Use Case Implementation

Figure 6.4 presents the deployment architecture of the IoT notebook that has been designed to satisfy this use case. First of all, concerning the IoT notebook

documents, the prototyping of the IoT system described in the use case comprised three notebook documents. The first of them contains the configuration and development steps involved in the deployment of RESTful web services using Flask, a lightweight web application framework [69]. This first notebook document belongs to the **application** architectural element and, since the programming language is Python, it uses the default IPython kernel to execute the code cells. The second notebook document corresponds to the implementation of the Arduino, responsible for gathering the reads from the temperature sensor and forwarding them to the Flask application. This notebook document belongs to the **gateway** architectural element, and to support the execution of the code cells in the Arduino programming language it uses the custom-developed Arduino kernel, Kernelino. Similarly, the third notebook document corresponds to the integration of the Back-end application with the Philips Hue bridge. It also belongs to the **gateway** architectural element, is developed with Python, and is supported upon the IPython kernel.

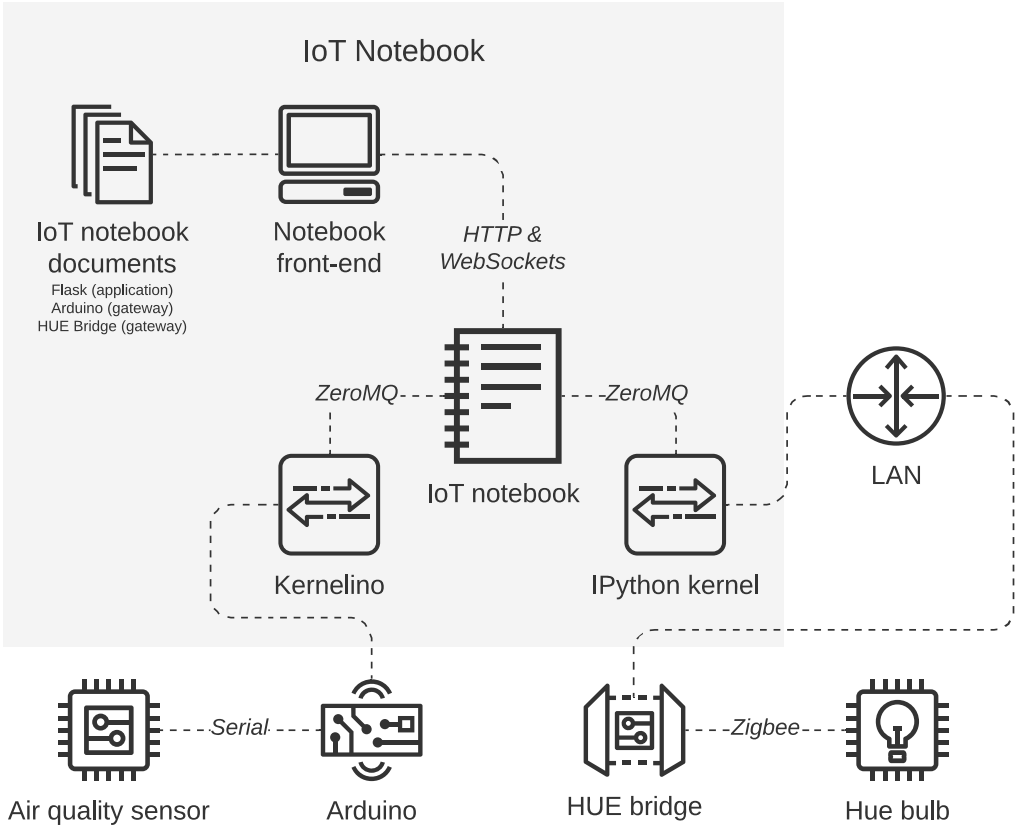


Figure 6.4: IoT notebook Validation use case architecture

The implementation of Kernelino required to integrate the messaging protocols of Jupyter with the Arduino command-line interface [5]. However, since the

messaging protocols are complex, writing a new kernel from scratch is not straightforward [82]. For this reason, an interface provided by Jupyter was used to wrap kernel languages in Python. Specifically, the class `ipykernel.kernelbase.Kernel` was subclassed, implementing the methods and attributes that forward the code from the IoT notebook to the Arduino command-line interface and retrieve the corresponding response [98]. In short, the interface provided by Jupyter handles all the ZeroMQ (a high-performance asynchronous messaging library [7]) sockets and communication mechanisms, making sure that the messages are correctly created and parsed for each type of request between the IoT notebook component and the Kernelino. Additionally, wrapper kernels can implement optional methods, notably for code completion and code inspection.

As illustrated in Figure 6.5, the Kernelino enables the execution of code cells, written in the Arduino programming language, directly from the notebook into the device, which is physically connected to the computer where the notebook is running.

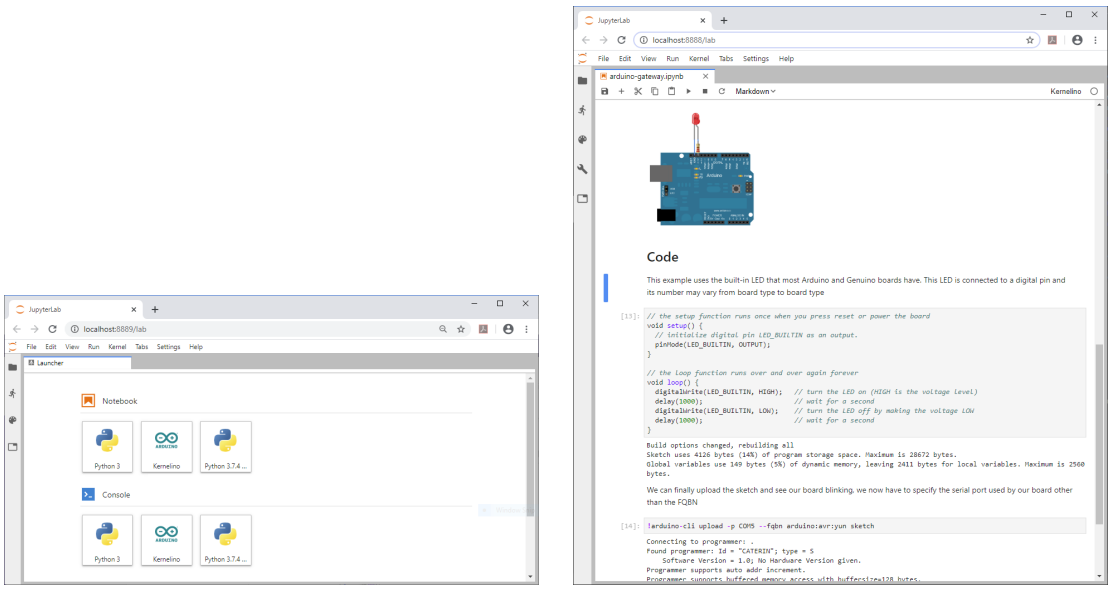


Figure 6.5: Screenshots of the IoT notebook

6.5.3 Results and Limitations

In the validation of the use case, it was possible to create and execute three notebook documents on top of the first implementation of the IoT notebook. These documents concerned two different architectural elements of an IoT system. While the first document regarded the business logic of the **application** element, the second and the third document concerned the **gateway** element and described the integration with a sensing and an acting device, respectively. The notebook

documents corresponding to the implementation of the RESTful web services and the integration of the Back-end application with the Philips Hue bridge were developed in Python and supported by the default IPython kernel. The first one of them was composed of nine code cells and twelve markdown cells, while the second was composed of seven code cells and nine markdown cells. In both of them, the first two cells concerned configuration steps and were prerequisites of all the subsequent cells as they regarded the installation of packages (through console commands supported by Jupyter by default) and the importation of the required modules. The third notebook document, corresponding to the configuration and execution of the Arduino gateway, required the implementation of a custom kernel to support the communication between the IoT notebook and the single-board microcontroller. In all the three documents it was necessary to execute the cells in order. Through the execution of the three notebook documents, the proposed maker-level use case could successfully be developed and deployed.

Two main limitations emerged from this work. On the one hand, to effectively implement the IoT notebook approach, future research efforts have to envision mechanisms to ease the integration between the notebook and several diverse IoT platforms. Although Arduino is a well-known and widely-used prototyping platform among IoT systems, the IoT platforms' and devices' landscape is huge, and the development effort to support just that integration was significant. On the other hand, although the IoT notebook approach was partially inspired by the findings of the previous research work presented in chapters 3 and in this article, it has been validated against a realistic use case, it is still necessary to conduct an evaluation involving IoT developers.

The first stage of such evaluation consists of determining how useful do novice developers find the IoT notebook as a resource to prototype IoT systems, by following the documentation and executing the code of pre-existing notebook documents. To that end, several use cases like the one presented in the previous section must be implemented. The second stage of the evaluation will assess how easy it is for IoT developers to document and share their prototyping process through the IoT notebook. In both stages, a user study must be conducted.

Finally, regarding the IoT OSS repositories analyzed in Chapter 4, just one of them corresponds to a cross-platform IDE: `plataformio-core`. It is a Command-Line Tool upon which is built the Plataformio IDE, a toolset for embedded C and C++ development that has pre-configured settings for the most popular embedded boards, and integration with numerous cloud platforms and web services feeds. The main difference of this toolset with the IoT notebook is the lack of documentation accompanying the source code, and the impossibility to capture and share the steps followed during the development process.

6.6 Conclusion

In this work, I identified the features that, based on a use case, an IoT-tailored literate computing approach should satisfy to support the prototyping of IoT systems. Among the set of identified features, there were prioritized a subset of them and implemented a preliminary version of what was called *IoT notebook* that was validated by prototyping a realistic maker-level IoT system. I consider that the characteristics of the computational notebooks are suitable to support IoT developers in the prototyping and documentation of their IoT systems. However, in line with the specificities of the architectural elements involved in such kind of systems, current computational notebooks have to be adapted. Future research should focus on the implementation of the IoT notebook with all the features and its assessment based on the feedback provided by the developers.

Chapter 7

Conclusion

The previous chapters have outlined the research activity encompassed in the analysis and identification of the characteristics and the most challenging issues of IoT systems development; and the proposition of documentation, programming, and prototyping tools, consistent with those characteristics and aimed at helping the developers to overcome the identified challenging issues. To conclude this thesis now are summarized the key contributions and possible future works.

7.1 IoT developers survey

In this work, a survey was conducted to identify the most complex issues experienced by novice programmers when developing IoT systems. The first contribution of this work is the **generic interpretation framework** in which the survey was framed. In this interpretation framework, the architecture, subsystems, and software development tasks of a significant subset of this kind of system were abstracted. This abstraction, on the one hand, gives the developers a common understanding of the software components involved in IoT systems, even if they work in different projects, with diverse architectures and IoT devices and technologies. On the other hand, it enables the rating and evaluation of the most challenging issues at a software task level of detail while, at the same time, locates them into a system-level view.

The second contribution regards the **identification of the most complex issues** based on the rating of software development tasks according to their difficulty level and completion time; the ranking of the most complex tasks; and the qualitative perception of each respondent about such complexity. The results that emerged from the survey indicated that the most challenging issues reported by unexperienced IoT developers concerned: the difficulty to find well-structured documentation that might be understood by a novice, the complexity inherent to the integration of the subsystems, and the integration with third-party services. Additionally,

there were identified aspects perceived as complex across various subsystems (development of user interfaces, the configuration of development environments, and the development of the business logic), as well as aspects whose complexity is split across various subsystems (integration between the Gateway and the third-party service APIs, the implementation and integration of the persistence component, and the design, implementation, and consumption of the RESTful web services).

To the best of my knowledge, this is the first study to express the complex issues as concrete development tasks that are not dependent on a particular kind of project, its architecture, or its technology stack. Furthermore, these findings might guide future research efforts to ease the learning curve in the teaching of IoT and might help to improve on-boarding time estimations, hiring criteria, and human resource management within the industry IoT projects.

7.2 IoT Open Source Software mining

In this work, an exploratory study was conducted mining and analyzing 30 popular IoT OSS and 30 popular non-IoT OSS projects available on GitHub, to provide empirical insights into the peculiarities of IoT software development. The contributions of this work are: (i) **providing evidence** about IoT development characteristics (the distribution of programming languages, the specialization of contributors, the evolution of the files, and the adopted dependencies); and (ii) **suggesting future research efforts** to satisfy software engineering needs in the IoT scenario.

Concerning the IoT development characteristics, it could be determined, among others, that: (i) IoT developers are less oriented towards the adoption of a lead programming language, but they work with different programming languages, depending on the capability of deployment environment; (ii) a strong majority of the IoT developers are frequently modifying the files written in compiled and interpreted programming languages (where the business logic of the application typically resides), while a few of them specialize in shell-oriented languages (generally used for configuration and deployment tasks); (iii) the files corresponding to compiled and interpreted programming languages evolved equally across the various development phases, while shell-oriented files are scarcely modified; and (iv) IoT projects have significantly fewer dependencies than non-IoT projects and they are also shared among fewer projects, with only 5% of them shared by two or more repositories.

In line with these findings, future research efforts should consider that tools like IDEs and software methodologies to support IoT developers should be language and platform agnostic, and not constrained to any given technological stack. Similarly, research efforts should be targeted at dealing with the devices heterogeneity by automating the generation and execution of deployment commands across several, often incompatible, devices. Additionally, there could be defined novel mechanisms

that IoT developers can adopt to make their code more extensible, modular, and reusable, given the peculiarities of the deployment platforms.

7.3 Code Recipes

In line with the outcomes of the IoT developers survey, this work represented a possible solution to the reported lack of documentation understandable by novice programmers. In particular, documentation concerning the integration of heterogeneous software components. The main contribution of this work was the proposition of *Code Recipes*. They are summarized and well-defined documentation modules, non-dependent from programming languages or run-time environments, and structured around the code fragments that are required to implement some portions of an IoT system. This approach was aimed at supporting novice IoT programmers, enabling them to easily become familiar with source code written by other developers that faced similar issues. This approach is considered to support novice IoT programmers, enabling them to easily become familiar with source code written by other developers that faced similar issues. While the Code Recipes were proposed mainly as a conceptual model, the IoT notebook outlined in Chapter 6, is considered to be a feasible alternative to implement them.

7.4 IoT Notebook

An-IoT tailored notebook approach was proposed to help developers to build and share a computational narrative around the prototyping of IoT systems. The contributions of this work were: (i) identifying the set of features that an IoT notebook should offer based on the characteristics of a maker-level IoT system; (ii) outlining which concepts of the identified features should be incorporated into the currently available computational notebooks; (iii) proposing an architecture and a conceptual model to support the implementation of the IoT notebook; and (iv) implementing and validating a preliminary version of the *IoT notebook* with a set of top-prioritized features. Specifically, the validation consisted of prototyping a realistic maker-level IoT system; three notebook documents were created and executed on top of the first implementation of the IoT notebook.

Future research efforts should be targeted at overcoming the two main limitations of the current implementation. The first limitation has to do with the huge size of the IoT device landscape. The development effort to support the integration with Arduino was significant. However, and despite the popularity of Arduino as prototyping platform among IoT systems, the IoT platforms' and devices' landscape is huge. Consequently, to effectively implement the IoT notebook approach, future research efforts have to envision mechanisms to ease the integration between the notebook and several diverse IoT platforms. The second limitation concerns

the limited experimental evaluation. Although the IoT notebook approach was partially inspired by the findings of our previous research work [29] and it was validated against a realistic use case, it is still necessary to assess developers' perspective on using it. Therefore, the current IoT notebook approach has to be expanded and validated by conducting further studies with IoT developers.

7.5 Current State and Future Work

The research works presented in this dissertation have followed a logical path. The first research efforts were aimed at understanding software development in the context of IoT systems. Such understanding was addressed from two perspectives: the rating and perceptions of novice IoT programmers concerning a generic architecture and a set of specific development tasks, and the quantitative analysis of the most popular open-source IoT applications publicly available. The results from these first research efforts were the identification of the most challenging issues that non-experienced programmers face and their perceptions behind such complexity, and the characteristics of IoT development in terms of the distribution of programming languages, the specialization of contributors, the evolution of the files, and the adopted dependencies.

From the first research efforts, I could determine that novice programmers struggle to find well-structured documentation that might guide them effectively to implement the most complex software development tasks, namely, the integration of heterogeneous subsystems and third-party services. Furthermore, the quantitative analysis provides evidence that IoT developers are less oriented towards the adoption of a lead programming language, just a few developers specialize in shell-oriented languages, and IoT projects have significantly fewer dependencies than non-IoT projects.

Upon the findings of these first research efforts, two correlated proposals emerged. At first, the Code Recipes, as a possible solution to the reported lack of documentation understandable by novice programmers, in which the source code written by novice developers is structured around well-defined documentation modules non-dependent from programming languages or run-time environments. The second proposal, the IoT notebook, in line with the identified lack of documentation and also with the heterogeneous set of development and deployment environments present in IoT systems, rely on computational notebooks to help developers to build and share a computational narrative around the prototyping of IoT systems.

The findings that emerge from the IoT developers survey, and the IoT OSS mining, point out the need for IDEs and software methodologies tailored to the particularities of IoT systems. Therefore, a promising area of future work consists in extending the integration between the notebook and several diverse IoT platforms and conducting further studies to assess the developers' perceptions of using it.

Additionally, it is necessary to evaluate how effective it is this approach to improve the documentation of IoT software artifacts and to prototype IoT systems that span across various development and execution environments.

Appendix A

Publications

Updated: February, 2020.

A.1 International Journals

1. Fulvio Corno, Luigi De Russis, Juan Pablo Sáenz (2020) **How is Open Source Software Development Different in Popular IoT Projects?** in: IEEE Access, IEEE, pages: 12, Volume 8
   ISSN: 2169-3536
   DOI: 10.1109/ACCESS.2020.2972364

2. Fulvio Corno, Luigi De Russis, Juan Pablo Sáenz (2019) **On the Challenges Novice Programmers Experience in Developing IoT Systems: A Survey** in: Journal of Systems and Software, Elsevier, pages: 21, Volume 157
   ISSN: 0164-1212
   DOI: 10.1016/j.jss.2019.07.101

3. Juan Pablo Sáenz, Mónica Paola Novoa, Darío Correal, Bell Raj Eapen (2018) **On Using a Mobile Application to Support Teledermatology: A Case Study in an Underprivileged Area in Colombia** in: International Journal of Telemedicine and Applications, Hindawi, pages: 8, Volume 2018
   ISSN: 1687-6415
   DOI: 10.1155/2018/1496941

A.2 Proceedings

1. Fulvio Corno, Luigi De Russis, Juan Pablo Sáenz (2019) **Towards Computational Notebooks for IoT Development** in: Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems (CHI EA

'19), ACM, Glasgow (UK), May 4-9, 2019, pages: 6
DOI: 10.1145/3290607.3312963

2. Fulvio Corno, Luigi De Russis, Juan Pablo Sáenz (2018) **On The Advanced Services That 5G May Provide To IoT Applications** in: Proceedings of the 2018 IEEE 1st 5G World Forum, IEEE, Santa Clara, CA (USA), July 9-11, 2018, pages: 4
DOI: 10.1109/5GWF.2018.8517038

3. Fulvio Corno, Luigi De Russis, Juan Pablo Sáenz (2018) **Easing IoT Development for Novice Programmers Through Code Recipes** in: Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training, ACM, Gothenburg (Sweden), May 27-June 3, 2018, pages: 4
DOI: 10.1109/5GWF.2018.8517038

4. Fulvio Corno, Luigi De Russis, Juan Pablo Sáenz (2017) **On the Design of an Energy and User Aware Study Room** in: 2017 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe), IEEE, Turin (Italy), September 26-29, 2017, pages: 6
DOI: 10.1109/ISGTEurope.2017.8260192

5. Fulvio Corno, Luigi De Russis, Juan Pablo Sáenz (2017) **Pain Points for Novice Programmers of Ambient Intelligence Systems: An Exploratory Study** in: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), IEEE, Turin (Italy), July 4-8, 2017, pages: 6
DOI: 10.1109/COMPSAC.2017.186

Acronyms

AmI Ambient Intelligence.

AMQP Advanced Message Queuing Protocol.

API Application Programming Interface.

AWS Amazon Web Services.

BLOB Binary Large Object.

CSS Cascading Style Sheets.

DSL Domain-Specific Languages.

GPS Global Positioning System.

GUI Graphical User Interface.

HTML Hypertext Markup Language.

HTTP The Hypertext Transfer Protocol.

I/O Input/Output.

IDE Integrated Development Environments.

IIRA Industrial Internet Reference Architecture.

IoT Internet of Things.

IoT-A Internet of Things - Architecture.

JSON JavaScript Object Notation.

LWM2M Lightweight Machine to Machine Protocol.

MDD Model-Driven Design.

mPRM mPower Remote Manager.

MQTT Message Queuing Telemetry Transport.

MVC Model-View-Controller.

OSS Open Source Software.

POM Project Object Model.

REPLs Read-Eval-Print Loops.

RESTful Web services that conform to the Representational state transfer (REST) architectural style.

RFID Radio-Frequency Identification.

S3 Amazon Simple Cloud Storage Service.

SDK Software Development Kit.

SNS Amazon Simple Notification Service.

SQL Structured Query Language.

URI Uniform Resource Identifier.

USB Universal Serial Bus.

WSN Wireless Sensor Networks.

XML Extensible Markup Language.

Zsh Zeta shell.

Bibliography

[1] A. Abbas et al. "Binary Pattern for Nested Cardinality Constraints for Software Product Line of IoT-Based Feature Models". In: *IEEE Access* 5 (2017), pp. 3971–3980. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2680470.

[2] Rabe Abdalkareem et al. "Why Do Developers Use Trivial Packages? An Empirical Case Study on Npm". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: ACM, 2017, pp. 385–395. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3106267. URL: http://doi.acm.org/10.1145/3106237.3106267.

[3] Alireza Ahadi et al. "Students' Syntactic Mistakes in Writing Seven Different Types of SQL Queries and Its Application to Predicting Students' Success". In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. SIGCSE '16. Memphis, Tennessee, USA: ACM, 2016, pp. 401–406. ISBN: 978-1-4503-3685-7. DOI: 10.1145/2839509.2844640.

[4] A. Ahmad et al. "An Empirical Study of Investigating Mobile Applications Development Challenges". In: *IEEE Access* 6 (2018), pp. 17711–17728. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2818724.

[5] Arduino. *GitHub - arduino/arduino-cli: Arduino command line interface*. https://github.com/arduino/arduino-cli. Online; last accessed September 23, 2019. 2019.

[6] Luigi Atzori, Antonio Iera, and Giacomo Morabito. "The Internet of Things: A Survey". In: *Comput. Netw.* 54.15 (Oct. 2010), pp. 2787–2805. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2010.05.010.

[7] The ZeroMQ authors. *ZeroMQ*. https://https://zeromq.org/. Online; last accessed September 23, 2019. 2019.

[8] A. Azzarà et al. "PyoT, a macroprogramming framework for the Internet of Things". In: *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*. June 2014, pp. 96–103. DOI: 10.1109/SIES.2014.6871193.

[9] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. 3rd. Addison-Wesley Professional, 2012. ISBN: 0321815734, 9780321815736.

[10] J. Beal, D. Pianini, and M. Viroli. "Aggregate Programming for the Internet of Things". In: *Computer* 48.9 (Sept. 2015), pp. 22–30. ISSN: 0018-9162. DOI: 10.1109/MC.2015.261.

[11] Hudson Borges and Marco Tulio Valente. "What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform". In: *Journal of Systems and Software* 146 (2018), pp. 112–129. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2018.09.016.

[12] Eleonora Borgia. "The Internet of Things vision: Key features, applications and open issues". In: *Computer Communications* 54 (2014), pp. 1–31. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2014.09.008. URL: http://www.sciencedirect.com/science/article/pii/S0140366414003168.

[13] Virginia Braun and Victoria Clarke. "Using thematic analysis in psychology". In: *Qualitative Research in Psychology* 3.2 (2006), pp. 77–101. DOI: 10.1191/1478088706qp063oa.

[14] John Seely Brown, Allan Collins, and Paul Duguid. "Situated Cognition and the Culture of Learning". In: *Educational Researcher* 18.1 (1989), pp. 32–42. DOI: 10.3102/0013189X018001032. eprint: https://doi.org/10.3102/0013189X018001032.

[15] Nathan Cassee et al. "How Swift Developers Handle Errors". In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR '18. Gothenburg, Sweden: ACM, 2018, pp. 292–302. ISBN: 978-1-4503-5716-6. DOI: 10.1145/3196398.3196428. URL: http://doi.acm.org/10.1145/3196398.3196428.

[16] Vint Cerf and Max Senges. "Taking the Internet to the Next Physical Level". In: *IEEE Computer* 49.2 (Feb. 2016), pp. 80–86. ISSN: 0018-9162. DOI: 10.1109/MC.2016.51.

[17] S. Chauhan et al. "A Development Framework for Programming Cyber-Physical Systems". In: *2016 IEEE/ACM 2nd International Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS)*. May 2016, pp. 47–53. DOI: 10.1109/SEsCPS.2016.016.

[18] CircuitPython. *CircuitPython*. https://circuitpython.org/. Online; last accessed September 23, 2019. 2019.

[19] Alem Čolaković and Mesud Hadžialić. "Internet of Things (IoT): A review of enabling technologies, challenges, and open research issues". In: *Computer Networks* 144 (2018), pp. 17–39. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2018.07.017.

[20] Diane J. Cook, Juan C. Augusto, and Vikramaditya R. Jakkula. "Ambient intelligence: Technologies, applications, and opportunities". In: *Pervasive and Mobile Computing* 5.4 (2009), pp. 277–298. ISSN: 1574-1192. DOI: 10.1016/j.pmcj.2009.04.001.

[21] Thomas D. Cook and D.T. Campbell. *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin, 1979.

[22] F. Corno, L. De Russis, and J. Pablo Sáenz. "On The Advanced Services That 5G May Provide To IoT Applications". In: *2018 IEEE 5G World Forum (5GWF)*. 2018, pp. 528–531.

[23] Fulvio Corno and Luigi De Russis. "Training Engineers for the Ambient Intelligence Challenge". In: *IEEE Transactions on Education* 60.1 (Feb. 2017), pp. 40–49. ISSN: 0018-9359. DOI: 10.1109/TE.2016.2608785.

[24] Fulvio Corno, Luigi De Russis, and Dario Bonino. "Educating Internet of Things Professionals: The Ambient Intelligence Course". In: *IT Professional* 18.6 (Nov. 2016), pp. 50–57. ISSN: 1520-9202. DOI: 10.1109/MITP.2016.100.

[25] Fulvio Corno, Luigi De Russis, and Juan Pablo Sáenz. "Easing IoT Development for Novice Programmers Through Code Recipes". In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*. ICSE-SEET '18. Gothenburg, Sweden: ACM, 2018, pp. 13–16. ISBN: 978-1-4503-5660-2. DOI: 10.1145/3183377.3183385.

[26] Fulvio Corno, Luigi De Russis, and Juan Pablo Sáenz. "Pain Points for Novice Programmers of Ambient Intelligence Systems: An Exploratory Study". In: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 01. July 2017, pp. 250–255. DOI: 10.1109/COMPSAC.2017.186.

[27] Fulvio Corno, Luigi De Russis, and Juan Pablo Sáenz. "Towards Computational Notebooks for IoT Development". In: *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI EA '19. Glasgow, Scotland Uk: ACM, 2019, LBW0154:1–LBW0154:6. ISBN: 978-1-4503-5971-9. DOI: 10.1145/3290607.3312963. URL: http://doi.acm.org/10.1145/3290607.3312963.

[28] Fulvio Corno, Luigi De Russis, and Juan Pablo Sáenz. "How is Open Source Software Development Different in Popular IoT Projects?" In: *IEEE Access* 8 (2020), pp. 28337–28348. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2972364.

[29] Fulvio Corno, Luigi De Russis, and Juan Pablo Sáenz. "On the challenges novice programmers experience in developing IoT systems: A Survey". In: *Journal of Systems and Software* 157 (2019), p. 110389. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.07.101. URL: http://www.sciencedirect.com/science/article/pii/S0164121219301566.

[30] Stacy Crook, Carrie MacGillivray, and Vernon Turner. *IDC MarketScape: Worldwide IoT Platforms (Software Vendors) 2017 Vendor Assessment*. Tech. rep. US42033517. IDC, 2017. URL: https://www.idc.com/getdoc.jsp?containerId=US42033517.

[31] Irena Pletikosa Cvijikj and Florian Michahelles. "The Toolkit Approach for End-user Participation in the Internet of Things". In: *Architecting the Internet of Things*. Ed. by Dieter Uckelmann, Mark Harrison, and Florian Michahelles. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 65–96. ISBN: 978-3-642-19157-2. DOI: 10.1007/978-3-642-19157-2_4.

[32] CXP Group. *PAC INNOVATION RADAR - IoT Platforms in Europe 2017*. https://www.pac-online.com/iot-platforms-europe-2017-pac-innovation-radar. 2017.

[33] Laura Dabbish et al. "Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository". In: *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*. CSCW '12. Seattle, Washington, USA: ACM, 2012, pp. 1277–1286. ISBN: 978-1-4503-1086-4. DOI: 10.1145/2145204.2145396. URL: http://doi.acm.org/10.1145/2145204.2145396.

[34] S. K. Datta and C. Bonnet. "Easing IoT application development through DataTweet framework". In: *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)*. Dec. 2016, pp. 430–435. DOI: 10.1109/WF-IoT.2016.7845390.

[35] S. K. Datta et al. "DataTweet: An architecture enabling data-centric IoT services". In: *2016 IEEE Region 10 Symposium (TENSYMP)*. May 2016, pp. 343–348. DOI: 10.1109/TENCONSpring.2016.7519430.

[36] D. Dobrilovic and S. Zeljko. "Design of open-source platform for introducing Internet of Things in university curricula". In: *2016 IEEE 11th International Symposium on Applied Computational Intelligence and Informatics (SACI)*. May 2016, pp. 273–276. DOI: 10.1109/SACI.2016.7507384.

[37] D. Dobrilovic et al. "Platform for teaching communication systems based on open-source hardware". In: *2015 IEEE Global Engineering Education Conference (EDUCON)*. Mar. 2015, pp. 737–741. DOI: 10.1109/EDUCON.2015.7096051.

[38] Stefan Endrikat et al. "How Do API Documentation and Static Typing Affect API Usability?" In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 632–642. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568299.

[39] Dave Evans. *The Internet of Things: How the Next Evolution of the Internet Is Changing Everything*. Tech. rep. Cisco Internet Business Solutions Group, 2011.

[40] G. Fortino et al. "Towards a Development Methodology for Smart Object-Oriented IoT Systems: A Metamodel Approach". In: *2015 IEEE International Conference on Systems, Man, and Cybernetics*. Oct. 2015, pp. 1297–1302. DOI: 10.1109/SMC.2015.231.

[41] A. Al-Fuqaha et al. "Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications". In: *IEEE Communications Surveys Tutorials* 17.4 (Fourthquarter 2015), pp. 2347–2376. ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2444095.

[42] Jayavardhana Gubbi et al. "Internet of Things (IoT): A vision, architectural elements, and future directions". In: *Future Generation Computer Systems* 29.7 (2013), pp. 1645–1660. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2013.01.010.

[43] Andrew Head et al. "Managing Messes in Computational Notebooks". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. Glasgow, Scotland Uk: ACM, 2019, 270:1–270:12. ISBN: 978-1-4503-5970-2. DOI: 10.1145/3290605.3300500. URL: http://doi.acm.org/10.1145/3290605.3300500.

[44] S. Hodges et al. "A New Era for Ubicomp Development". In: *IEEE Pervasive Computing* 11.1 (Jan. 2012), pp. 5–9. ISSN: 1536-1268. DOI: 10.1109/MPRV.2012.1.

[45] H. Hsieh et al. "ScriptIoT: A Script Framework for and Internet-of-Things Applications". In: *IEEE Internet of Things Journal* 3.4 (Aug. 2016), pp. 628–636. ISSN: 2327-4662. DOI: 10.1109/JIOT.2015.2483023.

[46] Michelle Ichinco and Caitlin Kelleher. "Exploring novice programmer example use". In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Oct. 2015, pp. 63–71. DOI: 10.1109/VLHCC.2015.7357199.

[47] Antonino Ingargiola. *1. What is the Jupyter Notebook? - Jupyter/IPython Notebook Quick Start Guide 0.1 documentation*. https://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html#kernel. Online; last accessed September 23, 2019. 2019.

[48] Intel. *The Intel IoT Platform*. white paper. 2015. URL: https://www.intel.com/content/www/us/en/internet-of-things/white-papers/iot-platform-reference-architecture-paper.html.

[49] S. M. R. Islam et al. "The Internet of Things for Health Care: A Comprehensive Survey". In: *IEEE Access* 3 (2015), pp. 678–708. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2015.2437951.

[50] Ivar Jacobson, Ian Spence, and Pan-Wei Ng. "Is There a Single Method for the Internet of Things?" In: *Queue* 15.3 (June 2017), 20:25–20:51. ISSN: 1542-7730. DOI: 10.1145/3121437.3123501. URL: http://doi.acm.org/10.1145/3121437.3123501.

[51] M. E. Joorabchi, A. Mesbah, and P. Kruchten. "Real Challenges in Mobile App Development". In: *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. Oct. 2013, pp. 15–24. DOI: 10.1109/ESEM.2013.9.

[52] Eirini Kalliamvakou et al. "The Promises and Perils of Mining GitHub". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: ACM, 2014, pp. 92–101. ISBN: 978-1-4503-2863-0. DOI: 10.1145/2597073.2597074.

[53] Mary Beth Kery et al. "The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: ACM, 2018, 174:1–174:11. ISBN: 978-1-4503-5620-6. DOI: 10.1145/3173574.3173748. URL: http://doi.acm.org/10.1145/3173574.3173748.

[54] Thomas Kluyver et al. "Jupyter Notebooks - a publishing format for reproducible computational workflows". In: *Positioning and Power in Academic Publishing: Players, Agents and Agendas*. IOS Press, 2016, pp. 87–90. URL: https://eprints.soton.ac.uk/403913/.

[55] Donald E. Knuth. "Literate Programming". In: *Comput. J.* 27.2 (May 1984), pp. 97–111. ISSN: 0010-4620. DOI: 10.1093/comjnl/27.2.97. URL: http://dx.doi.org/10.1093/comjnl/27.2.97.

[56] Michael Kölling and Fraser McKay. "Heuristic Evaluation for Novice Programming Systems". In: *Trans. Comput. Educ.* 16.3 (June 2016), 12:1–12:30. ISSN: 1946-6226. DOI: 10.1145/2872521.

[57] David Koop and Jay Patel. "Dataflow Notebooks: Encoding and Tracking Dependencies of Cells". In: *9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2017)*. Seattle, WA: USENIX Association, June 2017.

[58] G. Kortuem et al. “Educating the Internet-of-Things Generation”. In: *Computer* 46.2 (Feb. 2013), pp. 53–61. ISSN: 0018-9162. DOI: 10.1109/MC.2012.390.

[59] Theodora Koulouri, Stanislao Lauria, and Robert D. Macredie. “Teaching Introductory Programming: A Quantitative Evaluation of Different Approaches”. In: *Trans. Comput. Educ.* 14.4 (Dec. 2014), 26:1–26:28. ISSN: 1946-6226. DOI: 10.1145/2662412.

[60] X. Larrucea et al. “Software Engineering for the Internet of Things”. In: *IEEE Software* 34.1 (Jan. 2017), pp. 24–28. ISSN: 0740-7459. DOI: 10.1109/MS.2017.28.

[61] *LimeSurvey: the online survey tool - open source surveys*. https://www.limesurvey.org. Accessed: 2018-12-11.

[62] H. Mäenpää et al. “Assessing IOT Projects in University Education - A Framework for Problem-Based Learning”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)*. May 2017, pp. 37–46. DOI: 10.1109/ICSE-SEET.2017.6.

[63] Mauricio Verano Merino, Jurgen Vinju, and Tijs van der Storm. “Bacatá: A Language Parametric Notebook Generator (Tool Demo)”. In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2018. Boston, MA, USA: ACM, 2018, pp. 210–214. ISBN: 978-1-4503-6029-6. DOI: 10.1145/3276604.3276981. URL: http://doi.acm.org/10.1145/3276604.3276981.

[64] Microsoft. *Microsoft Azure IoT Reference Architecture*. white paper. 2018. URL: http://aka.ms/iotrefarchitecture.

[65] Daniele Miorandi et al. “Internet of things: Vision, applications and research challenges”. In: *Ad Hoc Networks* 10.7 (2012), pp. 1497–1516. ISSN: 1570-8705. DOI: 10.1016/j.adhoc.2012.02.016.

[66] B. Morin, N. Harrand, and F. Fleurey. “Model-Based Software Engineering to Tame the IoT Jungle”. In: *IEEE Software* 34.1 (Jan. 2017), pp. 30–36. ISSN: 0740-7459. DOI: 10.1109/MS.2017.11.

[67] Stephen Oney and Joel Brandt. “Codelets: Linking Interactive Documentation and Example Code in the Editor”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’12. Austin, Texas, USA: ACM, 2012, pp. 2697–2706. ISBN: 978-1-4503-1015-4. DOI: 10.1145/2207676.2208664.

[68] E. Osipov and L. Riliskis. “Educating innovators of future Internet of Things”. In: *2013 IEEE Frontiers in Education Conference (FIE)*. Oct. 2013, pp. 1352–1358. DOI: 10.1109/FIE.2013.6685053.

[69] Pallets. *Welcome to Flask — Flask Documentation (1.1.x)*. https://flask.palletsprojects.com/en/1.1.x/. Online; last accessed February 26, 2019. 2019.

[70] Luca Pascarella et al. "How is Video Game Development Different from Software Development in Open Source?" In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR '18. Gothenburg, Sweden: ACM, 2018, pp. 392–402. ISBN: 978-1-4503-5716-6. DOI: 10.1145/3196398.3196418.

[71] Pankesh Patel and Damien Cassou. "Enabling high-level application development for the Internet of Things". In: *Journal of Systems and Software* 103 (2015), pp. 62–84. ISSN: 0164-1212. DOI: 10.1016/j.jss.2015.01.027.

[72] Arnold Pears et al. "A Survey of Literature on the Teaching of Introductory Programming". In: *SIGCSE Bull.* 39.4 (Dec. 2007), pp. 204–223. ISSN: 0097-8418. DOI: 10.1145/1345375.1345441.

[73] F. Perez and B. E. Granger. "IPython: A System for Interactive Scientific Computing". In: *Computing in Science Engineering* 9.3 (May 2007), pp. 21–29. DOI: 10.1109/MCSE.2007.53.

[74] M. Piccioni, C. A. Furia, and B. Meyer. "An Empirical Study of API Usability". In: *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. Oct. 2013, pp. 5–14. DOI: 10.1109/ESEM.2013.14.

[75] João Felipe Pimentel et al. "A Large-scale Study About Quality and Reproducibility of Jupyter Notebooks". In: *Proceedings of the 16th International Conference on Mining Software Repositories*. MSR '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 507–517. DOI: 10.1109/MSR.2019.00077.

[76] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Berlin, Heidelberg: Springer-Verlag, 2005. ISBN: 3540243720.

[77] M. M. Raikar, P. Desai, and J. G. Naragund. "Active Learning Explored in Open Elective Course: Internet of Things (IoT)". In: *2016 IEEE Eighth International Conference on Technology for Education (T4E)*. Dec. 2016, pp. 15–18. DOI: 10.1109/T4E.2016.012.

[78] Baishakhi Ray et al. "A Large Scale Study of Programming Languages and Code Quality in Github". In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 155–165. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635922. URL: http://doi.acm.org/10.1145/2635868.2635922.

[79] Baishakhi Ray et al. “A Large-scale Study of Programming Languages and Code Quality in GitHub”. In: *Commun. ACM* 60.10 (Sept. 2017), pp. 91–100. ISSN: 0001-0782. DOI: 10.1145/3126905.

[80] Martin P. Robillard and Robert DeLine. “A field study of API learning obstacles”. In: *Empirical Software Engineering* 16.6 (Dec. 2011), pp. 703–732. ISSN: 1573-7616. DOI: 10.1007/s10664-010-9150-8.

[81] Christoffer Rosen and Emad Shihab. “What Are Mobile Developers Asking About? A Large Scale Study Using Stack Overflow”. In: *Empirical Softw. Engg.* 21.3 (June 2016), pp. 1192–1223. ISSN: 1382-3256. DOI: 10.1007/s10664-015-9379-3.

[82] Cyrille Rossant. *IPython Interactive Computing and Visualization Cookbook*. Packt Publishing, 2014.

[83] Brent Rubell. *Overview. CircuitPython with Jupyter Notebooks. Adafruit Learning System*. https://learn.adafruit.com/circuitpython-with-jupyter-notebooks. Online; last accessed September 23, 2019. 2018.

[84] Adam Rule, Aurélien Tabard, and James D. Hollan. “Exploration and Explanation in Computational Notebooks”. In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI ’18. Montreal QC, Canada: ACM, 2018, 32:1–32:12. ISBN: 978-1-4503-5620-6. DOI: 10.1145/3173574.3173606. URL: http://doi.acm.org/10.1145/3173574.3173606.

[85] Adam Rule et al. “Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding”. In: *Proc. ACM Hum.-Comput. Interact.* 2.CSCW (Nov. 2018), 150:1–150:12. ISSN: 2573-0142. DOI: 10.1145/3274419. URL: http://doi.acm.org/10.1145/3274419.

[86] I. Salman, A. T. Misirli, and N. Juristo. “Are Students Representatives of Professionals in Software Engineering Experiments?” In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. May 2015, pp. 666–676. DOI: 10.1109/ICSE.2015.82.

[87] John R. Savery. “Overview of Problem-based Learning: Definitions and Distinctions”. In: *The Interdisciplinary Journal of Problem-based Learning* (2006), pp. 9–20. DOI: 10.7771/1541-5015.1002.

[88] P. Selonen and A. Taivalsaari. “Kiuas – IoT Cloud Environment for Enabling the Programmable World”. In: *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. Aug. 2016, pp. 250–257. DOI: 10.1109/SEAA.2016.10.

[89] Janet Siegmund et al. “Measuring and modeling programming experience”. In: *Empirical Software Engineering* 19.5 (Oct. 2014), pp. 1299–1334. ISSN: 1573-7616. DOI: 10.1007/s10664-013-9286-4.

[90] S. M. Sohan et al. “A study of the effectiveness of usage examples in REST API documentation”. In: *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Oct. 2017, pp. 53–61. DOI: 10.1109/VLHCC.2017.8103450.

[91] J. A. Stankovic. “Research Directions for the Internet of Things”. In: *IEEE Internet of Things Journal* 1.1 (Feb. 2014), pp. 3–9. ISSN: 2327-4662. DOI: 10.1109/JIOT.2014.2312291.

[92] Andreas Stefik and Susanna Siebert. “An Empirical Investigation into Programming Language Syntax”. In: *Trans. Comput. Educ.* 13.4 (Nov. 2013), 19:1–19:40. ISSN: 1946-6226. DOI: 10.1145/2534973.

[93] Gary Stubbings and Simon Polovina. “Levering object-oriented knowledge for service-oriented proficiency”. In: *Computing* 95.9 (2013), pp. 817–835. ISSN: 1436-5057. DOI: 10.1007/s00607-013-0304-6.

[94] A. Taivalsaari and T. Mikkonen. “A Roadmap to the Programmable World: Software Challenges in the IoT Era”. In: *IEEE Software* 34.1 (Jan. 2017), pp. 72–80. ISSN: 0740-7459. DOI: 10.1109/MS.2017.26.

[95] A. Taivalsaari and T. Mikkonen. “A Taxonomy of IoT Client Architectures”. In: *IEEE Software* 35.3 (May 2018), pp. 83–88. ISSN: 0740-7459. DOI: 10.1109/MS.2018.2141019.

[96] A. Taivalsaari and T. Mikkonen. “Beyond the next 700 lot platforms”. In: *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. Oct. 2017, pp. 3529–3534. DOI: 10.1109/SMC.2017.8123178.

[97] A. Taivalsaari and T. Mikkonen. “On the development of IoT systems”. In: *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*. Apr. 2018, pp. 13–19. DOI: 10.1109/FMEC.2018.8364039.

[98] Jupyter Development Team. *Making simple Python wrapper kernels*. https://jupyter-client.readthedocs.io/en/stable/wrapperkernels.html. Online; last accessed September 23, 2019. 2015.

[99] Chun-Wei Tsai, Chin-Feng Lai, and Athanasios V. Vasilakos. “Future Internet of Things: Open Issues and Challenges”. In: *Wirel. Netw.* 20.8 (Nov. 2014), pp. 2201–2217. ISSN: 1022-0038. DOI: 10.1007/s11276-014-0731-0.

[100] G. Uddin and M. P. Robillard. “How API Documentation Fails”. In: *IEEE Software* 32.4 (July 2015), pp. 68–75. ISSN: 0740-7459. DOI: 10.1109/MS.2014.80.

[101] Jean-Philippe Vasseur and Adam Dunkels. *Interconnecting Smart Objects with IP: The Next Internet*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN: 0123751659, 9780123751652.

[102] Camilo Vieira et al. "Writing In-Code Comments to Self-Explain in Computational Science and Engineering Education". In: *ACM Trans. Comput. Educ.* 17.4 (Aug. 2017), 17:1–17:21. ISSN: 1946-6226. DOI: 10.1145/3058751. URL: http://doi.acm.org/10.1145/3058751.

[103] M. Weyrich and C. Ebert. "Reference Architectures for the Internet of Things". In: *IEEE Software* 33.1 (Jan. 2016), pp. 112–116. ISSN: 0740-7459. DOI: 10.1109/MS.2016.20.

[104] Michael Weyrich and Christof Ebert. "Reference Architectures for the Internet of Things". In: *IEEE Software* 33.1 (Jan. 2016), pp. 112–116. ISSN: 0740-7459. DOI: 10.1109/MS.2016.20.

[105] Claes Wohlin et al. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012. ISBN: 3642290434, 9783642290435.

[106] Dandong Yin et al. "A CyberGIS-Jupyter Framework for Geospatial Analytics at Scale". In: *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*. PEARC17. New Orleans, LA, USA: ACM, 2017, 18:1–18:8. ISBN: 978-1-4503-5272-7. DOI: 10.1145/3093338.3093378. URL: http://doi.acm.org/10.1145/3093338.3093378.

[107] F. Zambonelli. "Key Abstractions for IoT-Oriented Software Engineering". In: *IEEE Software* 34.1 (Jan. 2017), pp. 38–45. ISSN: 0740-7459. DOI: 10.1109/MS.2017.3.

[108] A. Zanella et al. "Internet of Things for Smart Cities". In: *IEEE Internet of Things Journal* 1.1 (Feb. 2014), pp. 22–32. ISSN: 2327-4662. DOI: 10.1109/JIOT.2014.2306328.

This Ph.D. thesis has been typeset by means of the TEX-system facilities. The typesetting engine was pdfLATEX. The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete TEX-system installation.