Computational Notebooks to Support Developers in Prototyping IoT Systems

# Computational Notebooks to Support Developers in Prototyping IoT Systems

Fulvio Corno[a], Luigi De Russis[a], Juan Pablo Sáenz[a,*]

[a] *Department of Control and Computer Engineering, Politecnico di Torino, Torino, 10129 Italy*

**Abstract**

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 Internet of Things (IoT) systems is also complex because of IoT heterogeneous and interconnected nature. Indeed, IoT system prototyping spans multiple development and execution environments, and developers, besides focusing on the code, are required to configure various devices. To ascertain if and how computational notebooks' capabilities might be useful in the IoT scenario, this work presents an IoT-tailored notebook architecture aimed at helping developers build and share a computational narrative around the prototyping of IoT systems. To that end, we propose the concept of "IoT notebook", for which we analyze the required features and present its first implementation. We evaluate our proposal by conducting an exploratory user study among non-expert IoT developers working in a large IT company. Finally, we point out the potential of this approach.

*Keywords:* Internet of Things, Computational notebook, Software development, Prototyping

*Corresponding author

*Email addresses:* fulvio.corno@polito.it (Fulvio Corno), luigi.derussis@polito.it (Luigi De Russis), juan.saenz@polito.it (Juan Pablo Sáenz)

## 1. Introduction

The development of Internet of Things (IoT) systems is complex and poses several challenges to developers [1]. It requires an unusually broad spectrum of design and development technologies and skills [2] and spans across multiple development and execution environments. Technically speaking, IoT systems can be characterized by four architectural elements, each with a precise set of computing resources, technologies, and communication protocols. These four architectural elements are defined by [3] as devices, gateways, cloud services, and applications. *Devices* comprise hardware to collect sensor data (sensing devices) or perform actions (acting devices). *Gateways* collect, preprocess, and forward the data coming from the sensing devices to the cloud, and vice-versa with the requests sent from the cloud to the acting devices. The *cloud services* manage the devices, acquire and store the data, and provide real-time and/or offline data analytics. *Applications* range from web-based dashboards to domain-specific web and mobile applications [2].

The above implies that, 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*:

1. the programming tools for IoT development are largely the same ones used for mobile and web application development [3], 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 [4];
2. the absence of documentation written and shared by and for non-expert

developers [5, 6];

3. 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 [7].

Against these three challenges, we envision that IoT developers would greatly benefit from an interactive computing tool. Our approach draws upon Computational notebooks characteristics to document, share, and execute the configuration and programming steps that non-experienced programmers complete over diverse execution and development environments. In this sense, instead of framing our work exclusively on an End-User Development approach, rather than hiding the code and providing visual abstractions to deal with it, our goal is to enable novice programmers to become proficient in implementing these kinds of systems. We consider that if the reasoning and the followed steps are better documented into a Computational notebook and under a well-configured execution environment, the IoT systems prototyping can be more reproducible and understandable to novice programmers.

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 [8, 9]. 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.

In our previous work [10], we preliminary assessed the suitability and limitations of current computational notebooks to support the development of an IoT system. We suggested an initial set of features that an IoT notebook should enable, such as (i) multiple programming languages in the same notebook; (ii) the capability to execute code in the documents in external devices; (iii) keep some code snippets on background execution; (iv) support the specification and installation of mandatory dependencies; (v) support the visualization of data

coming from the sensing devices or external services and platforms. After exploring the computational notebooks landscape, we developed an initial iteration of the IoT notebook that partially satisfied the suggested features. This first approximation led us to envision the additional features and choose the Jupyter notebook as the most suitable tool. However, the most remarkable conclusion was that a more in-depth assessment of the benefits and limitations of the approach was necessary. Indeed, this outcome was the motivation to deepen into a more accurate definition of the features, the design of an architecture, a cleaner implementation of all the proposed features, and an exploratory first-use study addressed by the present work.

In this paper, in the light of the architectural elements present in IoT systems and the features provided by current computational notebooks, we present the design, development, and evaluation of the first implementation of an IoT-tailored notebook that can represent a feasible environment to support IoT systems prototyping.

Specifically, the contributions of this paper are:

- The idea of relying on computational narratives to support non-expert developers in the prototyping of IoT systems.
- The design and implementation of the IoT notebook, a custom-tailored computational notebook with a set of features that, based on the architectural elements present in IoT systems, are required to enable IoT prototyping.
- A usability study demonstrates that the IoT notebook can potentially support non-expert developers in two aspects: (i) following the documentation and executing the code of various IoT components from a computational notebook; and (ii) documenting and sharing their prototyping process with other developers in an IoT-tailored programming environments. Improvements and suggestions emerging from this study are then discussed.

The remainder of the paper is structured as follows. Section 2 provides an overview of the current computational notebooks landscape and specifies some definitions, while Section 3 describes the related work. Section 4 presents a use case of an IoT system prototype with the four IoT architectural elements previously mentioned (devices, gateways, cloud services, and applications). Stemming from the use case and the literature, Section 5 describes the approach to using computational notebooks for prototyping IoT systems, enumerating the set of features that, given the characteristics of these systems, an IoT notebook should offer. Additionally, we propose an architecture able to support such features. The implementation of the IoT notebook is described in Section 6. An exploratory usability study where our approach is evaluated in the context of an IT company is described in Section 7. Eventually, Section 8 concludes the paper.

## 2. Background

This section briefly introduces literate programming, a paradigm in which the logic of a computer program is explained in natural language interspersed with snippets of macros and traditional source code, and mentions some of its limitations. Furthermore, we provide definitions for some basic concepts regarding computational notebooks.

Literate programming originates in 1984 from a paper by Donald Knuth. 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"* [11]. Inspired by this proposal, literate computing tools, such as computational notebooks, have emerged as a way to support the construction and sharing of computational narratives by enabling data analysts to arrange code, visualizations, and text in a computational narrative [11, 8, 9]. These literate computing tools are based on cells containing rich text or executable code that generates results or visualizations.

*Computational notebooks* commonly refer at the same time to the interactive

literate programming documents and to the software application to execute them [12]. For the sake of clarity, throughout this article, we will refer to them 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 [13], and their execution does not require cleaning the outputs of previous executions. Thus, an executed document 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 [14]. Each kernel executes a given programming language.

Concerning their collaborative nature, Computational Notebooks have three main benefits: on the one hand, the programmers can exchange the Notebook documents (accounting for the developer's reasoning, coding, and execution results, and written using a standard format), and execute them directly in the browser. Since the computational notebook runs in the web browser, it works across platforms. Therefore, in contrast to setting up the project in a development environment, the programmer has to open and execute the notebook in the browser. On the other hand, in Computational notebooks, developers do not devote time to setup tasks because they do not have to find out how to add libraries to their projects by downloading files and adding them to their projects. Indeed, for this reason, they are suitable for the educational setting, making assignment correction easier instead of having to configure several times the execution environment, which often leads to solutions not working

correctly. By using Computational notebooks, evaluators can support students more straightforwardly. Finally, unlike traditional IDEs, other than sharing just the code, the Computational notebooks enable the developers to share all code states, including the code execution results, which can be very indicative to other programmers, especially novices, regarding the behavior of the code.

However, as stated by Borowski *et al.* [15], synchronous real-time collaboration within Notebook documents is still in its infancy. According to the authors, current computational notebooks should integrate real-time commenting features, highlight tracked changes, and provide editing history. To this end, features like remote cursors or pointers (where both developers can see through the cursor what the other is working on) are essential. Finally, keeping track in real-time of the changes that other users performed could ease splitting up work and putting results together.

## 3. Related Work

This work is intended to provide insights into the suitability of a computational narrative approach to document, execute, and share the steps involved in IoT prototyping, especially for non-expert programmers. In the following, we addressed the related work from the perspective of (i) works aimed at exploring and analyzing the characteristics of current notebooks and (ii) the use of the notebooks in diverse domains and contexts. Additionally, based on the work by Lau *et al.* [16], we present an overview of some of the most popular currently available notebooks.

### *3.1. Large-scale Computational notebooks analyses*

Rule *et al.* [9] 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. In the same way, authors [17] have also concluded that as notebooks grow, they are more challenging to navigate or understand, discouraging sharing and reuse. For this reason, and taking into account that IoT prototyping might comprise much larger code fragments than data science, we the possibility of **splitting code across various cells** as an essential feature to include in our proposed IoT literate computing approach.

Pimentel *et al.* [12] present an analysis of the notebook characteristics that impact reproducibility. The 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. More specifically, authors draw attention to the fact that although current computational notebooks present the cells in a linear-top-bottom narrative, a user might execute them in an arbitrary order. In this way, notebooks' reproducibility is negatively affected by hidden states, out-of-order cells, and hardcoded paths. This situation, and the recommendation of the authors to manage the dependencies and guarantee linear execution, led us to identify **enabling the definition of execution order constraints** as a feature for our proposed IoT notebook. Naturally, we will describe these two features in detail in Section 5.

More recently, Pimentel *et al.* [18] conducted a further analysis in which, among others, they separated a group of popular notebooks to check whether notebooks that get more attention have more quality and reproducibility capabilities, isolated library dependencies, and tested different execution orders.

Then, based on the results of these analyses and their proposed best practices, the authors developed *Julynter*, a tool in the form of a Jupyter Lab extension that performs diverse checks on the quality and reproducibility of notebooks in real-time and produces recommendations. Specifically, the recommended changes concern four categories: invalid title, hidden state, confuse notebook, and import. In this manner, the tool's graphical interface enables users to click on the recommendations to apply actions. Upon the analysis of the notebooks and the evaluation of their proposed tool, the authors determined that most notebooks do not test their code and that a large number of notebooks has characteristics that hinder the reasoning and reproducibility, such as out-of-order cells non-executed code cells.

In the same line, Källén *et al.* [19] analyzed a corpus of 2.7 million Jupyter notebooks hosted on GitHub to identify how recurrent code cloning is among Jupyter Notebooks at the level of notebook cells. Findings indicated that more than 70% of all code snippets are exact copies of other snippets, and around 50% of all notebooks do not have any unique snippet but consist solely of snippets that are also found elsewhere. Additionally, according to this analysis, notebooks are, in general small (median 13 kB) and contain relatively few code cells (median 9) and few lines of code (median 50).

Wang *et al.* [20] conducted a preliminary study to determine whether the code written in a large set of Jupyter notebooks is implemented with good qualities. The analyzed dataset was composed of a presumably high-quality set of 1982 Python-based notebooks curated by the Jupyter team. The outcomes from the study demonstrated that even the notable notebooks were inundated with low-quality code with poor respect to the Python style conventions and code qualities in terms of including unused variables and accessing deprecated features of specific libraries. Based on their findings, the authors raise attention to the need to propose approaches to improve the reliability and quality of the code, apply best practices for software quality, and ensure a good balance between text and code in Jupyter notebooks. What is more, in the authors' opinion, this need is essential considering that Jupyter notebooks are often

used as tutorials or documentation for non-experienced programmers to learn practical programming skills.

Kery *et al.* [21, 13] have studied the exploratory programming nature of Computational notebooks in the context of Data Science. In particular, the authors provided a definition for exploratory programming and analyzed code behaviors in a literate programming environment and how data scientists keep track of the variants they explore. Their definition suggests that the programmer writes code as a medium to prototype or experiment with different ideas and is pursuing an open-ended goal that might evolve through the process of programming. Authors' analyses in the context of literate computing suggest the need for automated version control mechanisms capable of tracking the different values of a variable and storing code dependency information among cells. Furthermore, they suggest providing several graphical visualization alternatives to ease versioning, comparison, and debugging.

### *3.2. Computational notebooks uses in diverse domains*

Head *et al.* [22] 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.* [23] 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 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.* [24] 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 domain-specific languages 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.* [25] 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 moni-

toring, actuator control, event detection and reaction, and access to historical data.

The CircuitPython website [26] 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 [27]. Specifically, like the kernel introduced in this paper, 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 USB link.

Among the currently available computational notebooks, Project Jupyter is one of the most widely used platforms [9]. 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 [9]. Jupyter Notebook originated from IPython [28] and, in addition to Python, it natively supports a variety of programming languages, such as Julia, R, Javascript, and C [12]. 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.

However, such notebooks have limitations [13]: (i) saving application states is complex, 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. Given these limitations, particularly the last one regarding modifying the notebook behavior, our proposed notebook has focused on changing the default notebook behavior by adding a set of features to make it viable in the IoT context. As we will describe in detail in Section 5.1, these features regard simultaneously keeping in execution two code cells, distinguishing configuration and business logic steps, splitting the code across various cells, and defining execution order constraints.

Chattopadhyay *et al.* [29] conducted a systematic study using semi-structured interviews and a survey with data scientists to identify and prioritize pain points when working with notebooks (along the entire workflow spectrum, ranging from setup to sharing). Among the findings, the authors determined that compared to traditional integrated development environments, proper code assistance within notebooks is almost non-existent. Features like autocompletion, refactoring tools, and live templates are missing or do not function properly. Additionally, notebooks provide little or no support for finding, removing, updating, or identifying deprecated packages. In particular, discovering which packages are installed isn't accessible from the notebook environment, forcing data scientists to run over the command line the commands to manage their development environment. Furthermore, it is impossible to follow the code flow for debugging purposes due to out-of-order execution. Breakpoints are not feasible in this scenario, where functions are split across separated cells. Consequently, the only way to debug most notebooks is through print statements. The authors concluded that addressing such pain points can substantially improve the usefulness, productivity, and user experience for data scientists who work with computational notebooks.

### *3.3. Currently available computational notebooks*

Lau *et al.* [16] performed a comprehensive design analysis of 60 notebooks among industry products and academic and experimental projects and formulated a design space with the variations in systems features. Specifically, the authors built the design space around ten dimensions that considered importing data, editing code and prose, running code, and publishing notebook outputs. While industry products tend to rely on the Jupyter notebook, given its popularity and widespread adoption, academic and experimental projects can afford to experiment more with different kinds of cell execution orders, live programming, and interactive outputs. Indeed, the authors adopted a broad definition of computational notebooks to determine which tools to include in their design analysis: a system that supports literate programming using a text-based pro-

gramming language while interweaving expository text and program outputs into a single document. As we have done in our research, the authors determined that a way to push notebooks beyond current designs is by focusing on user groups other than data scientists, such as educators, artists, or those in low-resource computing environments. Similarly, Verano *et al.* [30] surveyed 12 notebooks and four other systems within a formative study to inform the design of their Bacatá approach that generates notebook UIs for DSLs. Based on the insights of these two works and the set of notebooks tools that they included in their analyses, in the following, we briefly list and describe other available literate computing approaches different from Jupyter.

*RStudio.* is an open-source IDE for R that supports direct code execution from the source editor and a notebook interface that allows developers to combine text, scripts, and outputs into a single document. It supports syntax highlighting, code completion, indentation, and tools for plotting, history, debugging, and workspace management.

*Mathematica.* was one of the first literate computing environments. The Wolfram Notebook Interface enables natural language queries and real-time graphs manipulation with multiple parameters other than the usual capabilities of computational notebooks.

*Matlab.* provides a live editor to create scripts that combine code, output, and formatted text in an executable notebook. It aims at supporting teaching by dividing code into sections that can be executed independently and modifying the code on the fly to demonstrate concepts.

*Spark Notebook.* is an open-source notebook aimed at enterprise environments. It consists of an interactive web-based editor that supports reproducible analysis with Scala, Apache Spark, and the Big Data ecosystem by combining Scala code, SQL queries, Markup, and JavaScript.

*Observable.* is a notebook that supports JavaScript execution in all of its cells. Consequently, the Observable code runs directly in the browser's JavaScript engine, and the code within the cells can be defined asynchronously using promises. Additionally, the notebook automatically re-evaluates the cells whenever the code changes.

*Streamlit.* is an open-source app framework targeted at Machine Learning engineers and data scientists. It aims to enable them to create and share custom data-driven web apps through Python scripts. It provides an immediate-mode live coding environment that updates every time a Python source file changes.

*Codestrates.* is a literate computing approach targeted at non-professional programmers and aimed at extending the functionality of computational notebooks to support the development of reprogrammable applications collaboratively [31, 15]. Codestrates allows for real-time collaboration—not only for editing code of computations, but also for reprogramming and extending the notebook itself.

As can be observed from the above, although Jupyter is the most popular and widespread adopted computer notebook system, several other alternatives are publicly available and target different application domains and needs. From a broader research perspective, computational notebooks encompass various themes such as exploratory programming, live programming, and literate computing. Specifically, in the data science context, they are end-user programming environments where coding aims to create data science insights or research findings rather than implement artifacts for broader public use.

## 4. Use Case

This section describes a use case concerning a maker-level IoT system, to provide a running example for better understanding the four architectural elements present in IoT systems (as defined in [2]), as well as to identify the common characteristics of these systems. Such a use case stemmed from the an

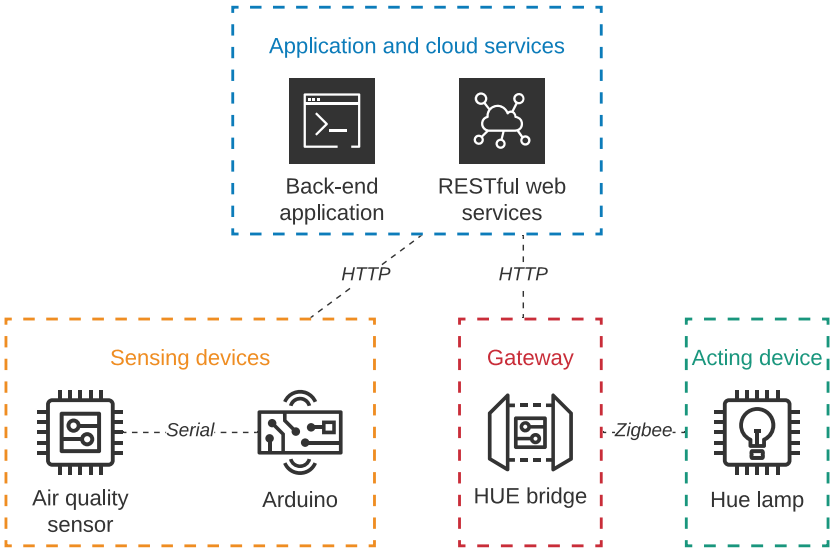



Figure 1: IoT architectural elements in the Use Case

analysis of open-source IoT [32] projects and an exploration of publicly-shared Arduino[1] projects. This allowed us to keep the use case realistic and generalizable, both for the involved technologies and for its overall goal.

### *4.1. 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 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

[1] `https://create.arduino.cc/projecthub`, last visited on September 2, 2021

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.

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, 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 RESTful web services exposed by the latter, implemented using Flask, a Python web micro-framework.

### *4.2. Characteristics of an IoT system prototype*

Starting from the use case, we can easily highlight some characteristics of the two software artifacts:

1. **They remain in background execution**. The Arduino script keeps gathering and monitoring the readings from the air quality sensor, the Back-End application keeps mediating the communication and interacting with the sensing and acting devices, while the RESTful web services keep forwarding the requests sent from the Arduino.
2. They are deployed over **heterogeneous run-time environments**. Specifically, while an Arduino script is executed on a computing-resource con-

strained device without an operating system, the Back-End application and its web services are deployed on the cloud.

3. They involve multiple devices: prototyping 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.

## 5. IoT Notebook Design

Based on the characteristics specified in Section 4.2 and in the literature (see Section 3), this section presents the concept of an IoT-tailored notebook (hereinafter, referred to as IoT notebook) as a dedicated tool to support IoT systems prototyping. We first list the set of features that an IoT notebook should offer (Section 5.1). Then, we introduce our IoT notebook proposal from two perspectives: first, conceptually, outlining which concepts from the identified features should be incorporated into the currently available computational notebooks (Section 5.2); and lastly, technically, describing the architecture that supports the implementation of the IoT notebook (Section 5.3).

### *5.1. Features of an IoT notebook*

We identified a set of 6 features that emerge either from the previously identified characteristics (**FT-1** through **FT-3**) or from the literature (**FT-4** through **FT-6**). Features extracted from the literature concern the presentation of the notebook documents, thus aiming at making notebook documents more understandable and consistent with the structure of IoT systems. These six features are summarized in Table 1.

In particular, for what concerns the technical characterstics of IoT systems prototyping:

- **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).

- **FT-2:** 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.

- **FT-3:** 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 **Identifying the configuration steps from the development ones**. Besides clearly differentiating the nature of these steps, it would provide consistency to the notebooks.

Instead, the literature informed the following three features:

- **FT-4:** Since IoT systems are composed of several devices, back-end, and end-user services and applications, they require a broad spectrum of design and development skills. They span across various development and execution environments [33, 3, 2]. Therefore, to adequately structure and support the heterogeneity of IoT systems, 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. In contrast, the group of documents regarding the setup and development of the Flask RESTful web server should be categorized as cloud service documents.

- **FT-5:** 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 [12]. This feature can be useful in the field of data science when checking if changes to a prior analytical step impact later computations [9]. However, hidden states, out-of-order

cells, hardcoded paths, and other bad practices also prevent the reproduction of notebooks [12]. 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 [34]. On the contrary, managing the dependencies of notebooks and guaranteeing the linear execution order could improve the reproducibility rate [12].

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-6:** Notebooks evolve and grow and they often become difficult to navigate or understand, discouraging sharing and reuse [17]. 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.

### *5.2. IoT notebook Conceptual Model*

Figure 2 depicts the conceptual model of the IoT notebook. Concerning the concepts involved in the Jupyter notebook[2], we introduce the concepts of *Architectural element*. It enables the classification of Notebook documents depending on whether they belong to the devices, gateways, cloud services, or applications.

[2]Given its wide adoption and open standards we take it as a reference upon which to build our proposal.

Table 1: IoT notebook features

| ID | Features |
|---|---|
| FT-1 | Keep executing code cells simultaneously |
| FT-2 | Detect and identify the devices |
| FT-3 | Identify configuration and business logic steps |
| FT-4 | Group several notebook documents |
| FT-5 | Enable the definition of execution order constraints |
| FT-6 | Split code across various cells |

As will be described in detail in Subsection 7.2, in the context of the conceptual model illustrated in Figure 2, the *IoT notebook* can be composed of various *Notebook documents*, each one of which belongs to a given *Architectural element* (among devices, gateways, cloud services, or applications). In this manner, a *Notebook document* might contain, for instance, the steps to complete the implementation and deployment of an Arduino application. In that case, the *Notebook document* belongs to the devices' *Architectural element*, and its associated *Kernel* would be the one that supports the interaction with the Arduino board. Similarly, another *Notebook document* might concern implementing and deploying a Python Back-end application. In this case, it would belong to the applications *Architectural element* and be associated with a Python *Kernel*.

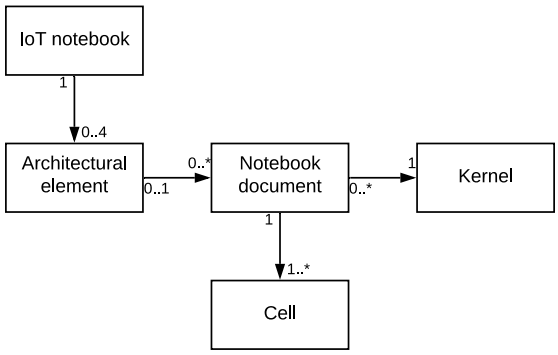



Figure 2: IoT notebook Conceptual Model

### 5.3. IoT notebook Architecture

As stated before, we studied and inspired our architecture from the architecture of Jupyter. Figure 3 depicts our 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 [3]. To that end, we structured our architecture around five blocks, listed below.

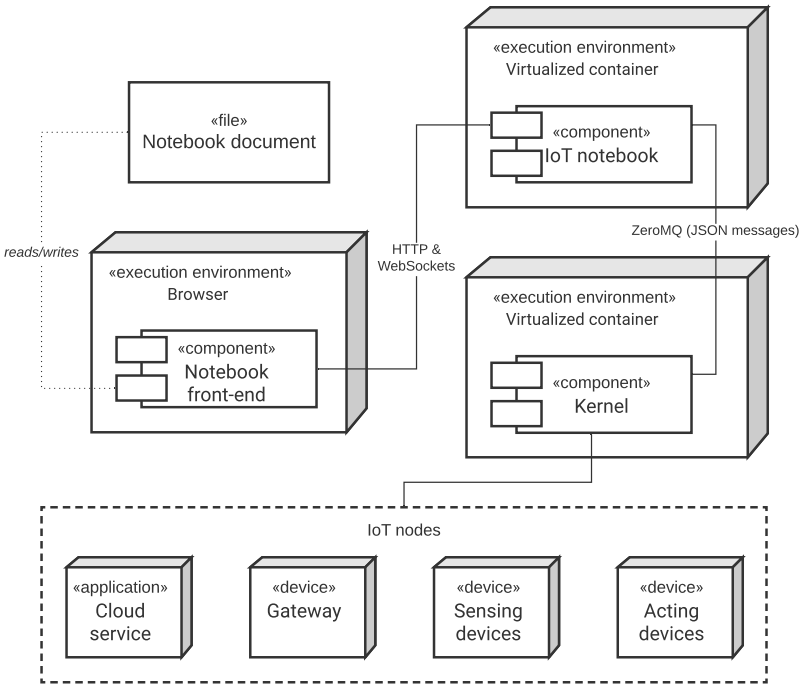



Figure 3: IoT notebook Architecture

*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 1, at the highest level, a notebook is a dictionary with a few keys: *meta-*

*data (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 and their corresponding outputs.

Aiming at satisfying our identified features, the following modifications (highlighted with red text in Listing 1) are proposed over the current *Notebook documents* structure:

- Add the `architectural_element` field (mandatory) to enable the grouping of the Notebook documents depending on the architectural element to which they belong (**FT-4**). The four possible string values for this field are: devices, gateways, cloud services or applications.

- Include the `background_execution` field (optional and `false` by default) 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 (optional and `false` by default) to indicate if the given cell must be executed whenever any other cell in the Notebook document is executed (**FT-5**). The values that this field may take are true or false.

- Include the field `is_linked_previous_cell` on each cell to enable the splitting of a large code fragment across various cells (**FT-6**). The values that this field may take are true or false. When the value is true, before executing the code in the corresponding cell, the IoT notebook executes the code in the previous cell. If that cell is also linked to its previous one, the IoT notebook acts accordingly and executes the code of various linked cells. In this manner, the code can be split across various cells linked to

another by setting this field to true.

- Add the `is_library_installation` field to represent if a given code cell has to be executed as a command-line instruction. This field is necessary to enable the execution of configuration steps, where libraries or modules have to be installed, before proceeding with the business logic steps (**FT-3**). The values that this field may take are true or false.

*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 (see Figure 7 for an example). Apart from the features that the Jupyter front-end currently provides, our IoT notebook front-end should include new user interface elements aimed at: enabling users to edit and visualize the new fields of the *Notebook document* previously described; and 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-2**).

*IoT notebook*

Our proposed architecture follows a client-server pattern [35] in which the notebook back-end is deployed remotely in the server execution environment. As shown in Figure 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, our IoT-tailored kernel is required to support the detection from the IoT notebook of the available sensing and acting devices on which the code cells may be executed (**FT-2**). The IoT-

```
{
  "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]"
    },
    "architectural_element": "devices, gateways, cloud services or
        applications"
  },
  "nbformat": 4,
  "nbformat_minor": 0,
  "cells": [
    {
      "cell_type": "code",
      "execution_count": 1,
      "metadata": {
        "background_execution": true or false,
        "is_prerequisite": true or false,
        "is_linked_previous_cell": true or false,
        "is_library_installation": true or false
      },
      "source": "[some multi-line code]",
      "outputs": [
        {}
      ]
    }
  ]
}
```

Listing 1: Notebook document JSON top structure

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 we have previously mentioned (cloud services, gateways, sensing and acting devices).

## 6. Implementation

In this section, we describe the implementation of the IoT notebook features reported in Section 5.1. To do so, we organized the presentation around each feature and upon the architecture previously outlined.

***FT-1**: Keep executing the code cells simultaneously*

The adopted strategy to implement this functionality was to rely on the `lib.backgroundjobs` IPython module that manages background (threaded) jobs for each execution. In this way, a new job can be created for each code cell so that the same kernel can run multiple code cells simultaneously, and the other cells can be run, if needed, without waiting for the other cells' execution to end.

***FT-2**: 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. In this version of the IoT notebook, we implemented a custom *IoT-tailored* kernel 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.

The development of Kernelino required to integrate the messaging protocols of Jupyter with the Arduino command-line interface [36]. However, since the messaging protocols are complex, writing a new kernel from scratch is not straightforward [37]. Thereby, we used an interface provided by Jupyter to wrap kernel languages in Python. We subclassed `ipykernel.kernelbase.Kernel` and implemented the methods and attributes that forward the code from the IoT notebook to the Arduino command-line interface and retrieve the corresponding response [38]. In short, the interface provided by Jupyter handles all the ZeroMQ (a high-performance asynchronous messaging library [39]) 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 4, the IoT notebook allows users to launch an Arduino notebook document. That document runs all the code cells (written in the Arduino programming language) directly from the notebook into the board and retrieves their corresponding output. Such integration is possible thanks to the Kernelino. It manages the execution of code cells into the board that is physically connected to the computer where the notebook is running.

Regarding the graphical user interface, we added a button in the right upper corner of the document to allow users to choose which device to run the code cells of such document. This selection is mandatory, and all the run buttons in the code cells remain disabled until the user selects the device and the custom-tailored kernel achieves the connection. When successfully connected, the label changes from "Board" to the board's model name ("Arduino MKR WiFi 1010" in this case), as shown in Figure 5.

***FT-3**: Differentiate configuration and business logic steps*

As shown in Figure 6b, a checkbox was added below each cell with the label "Is library". When it is checked, the code cell executes as a command-line instruction. For instance, in the screenshot in the Figure, the code `arduino-cli`

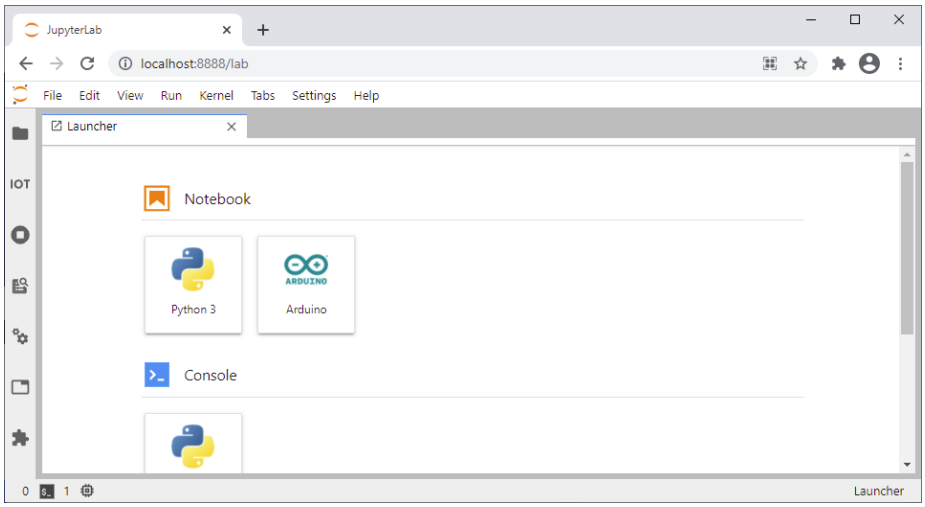



Figure 4: IoT notebook launcher

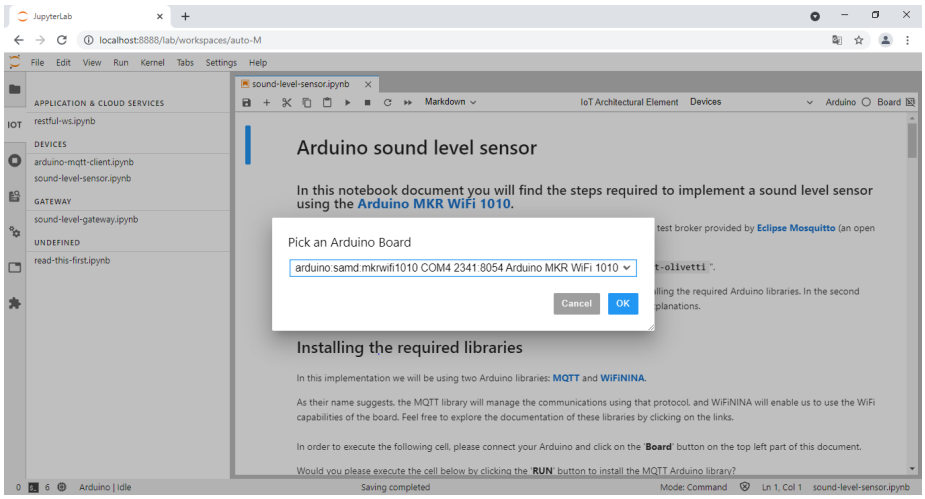
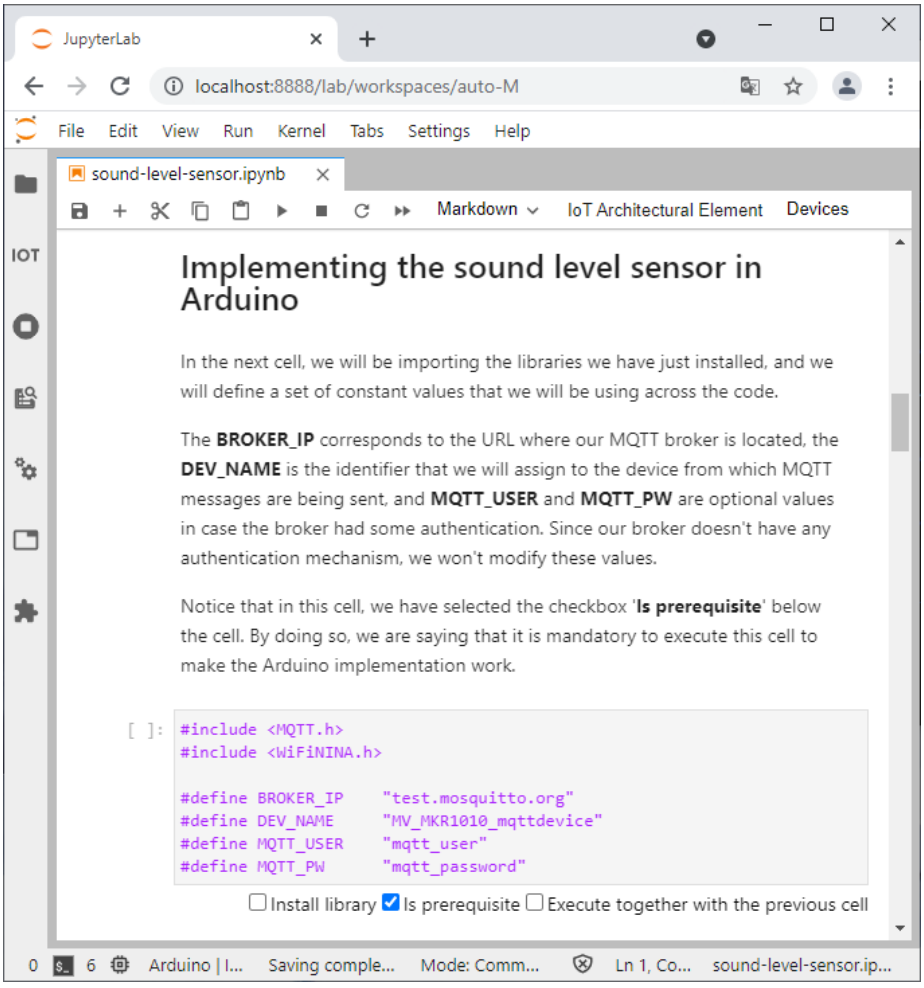


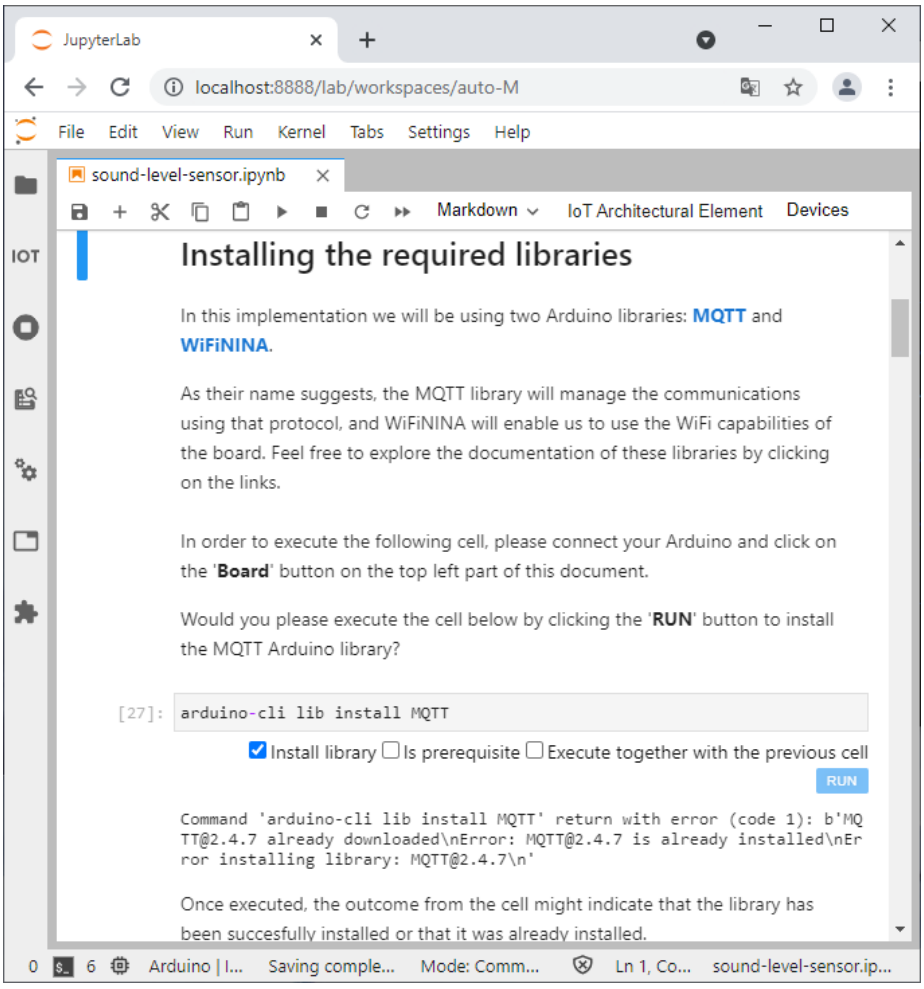

Figure 5: IoT notebook menu to choose the Arduino board

`lib install MQTT` is run as a command-line instruction. In this manner, users can include and describe, within the notebook document, the libraries or modules that must be installed before implementing the business logic steps. Furthermore, since the configuration steps should be executed individually, the "Is prerequisite," and the "Execute together with the previous cell" (**FT-5** and




(a) Implementation of the `is_prerequisite` feature in the IoT notebook front-end

Figure 6: IoT notebook screenshots regarding the prerequisites and the configuration steps




(b) Implementation of the configuration steps in the IoT notebook front-end

Figure 6: IoT notebook screenshots regarding the prerequisites and the configuration steps

**FT-6**, that are described below) options are automatically disabled when the "Is library" option is checked. Additionally, to persist the changes and represent the fact that a cell represents a configuration step, the notebook document is modified by adding a `is_library_installation` field in the cell's metadata and setting it to true. Naturally, to integrate this feature, the default behavior of the Jupyter notebook also was modified so that when a cell is marked as `is_library_installation`, the kernel executes it as a command-line instruction.

***FT-4**: Group several notebook documents*

The notebook documents are grouped according to the architectural element they represent in the prototyped IoT system. In this manner, as described in the previous section, the mandatory `architectural_element` field was added to the notebook documents, and it can take four possible values: devices, gateway, cloud service, or application. In the notebook front-end, we implemented two elements. On the one hand, as shown in Figure 7, we added a selection box in the right upper corner of the document (next to the button to choose the device). On the other hand, as also shown in Figure 7, we added a tab panel in the left sidebar.

***FT-5**: Enable the definition of execution order constraints*

The development of this feature involves the Notebook document, the Notebook front-end, and the IoT notebook component.

The Notebook document, as described in the previous section (Listing 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, as can be seen in Figure 6, we developed a custom plugin that places a checkbox below each cell so that the users can

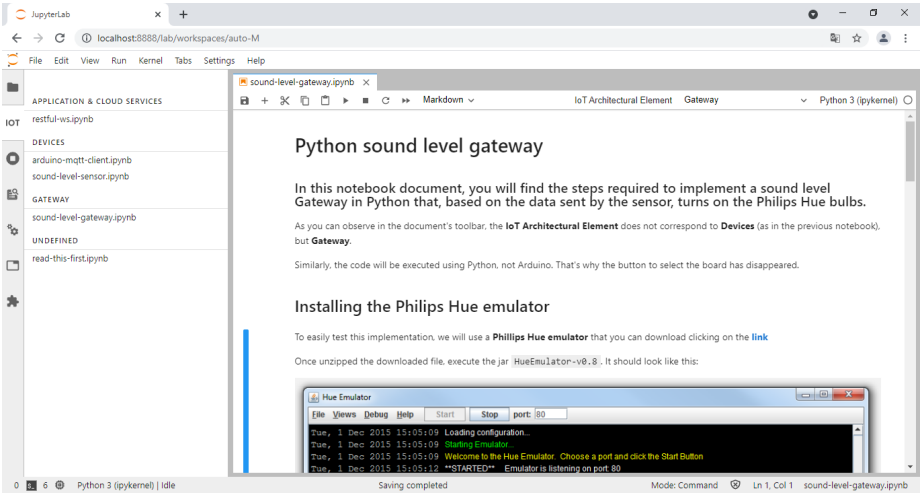



Figure 7: IoT notebook screenshot showing notebook documents grouped by architectural element

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. Additionally, when the user checks the cell as a prerequisite, the run button disappears as it is not a piece of code intended to run independently.

***FT-6**: Split the code across various cells*

Similar to the previous feature, splitting the code across various cells involved changes on the Notebook document, the Notebook front-end, and the IoT notebook component. Indeed, the implementation of this feature on the Notebook document and the Notebook front-end was almost the same. As described in the previous feature, a new field was added to the Notebook document to represent whether the last cell must be executed before the current cell `is_linked_previous_cell`. In the notebook front-end, the custom plugin that we developed also placed a checkbox below each cell to indicate if this cell has to be executed after its previous cell. In Figure 7, it can be observed how, by checking the "Execute together with the previous cell" checkbox, the

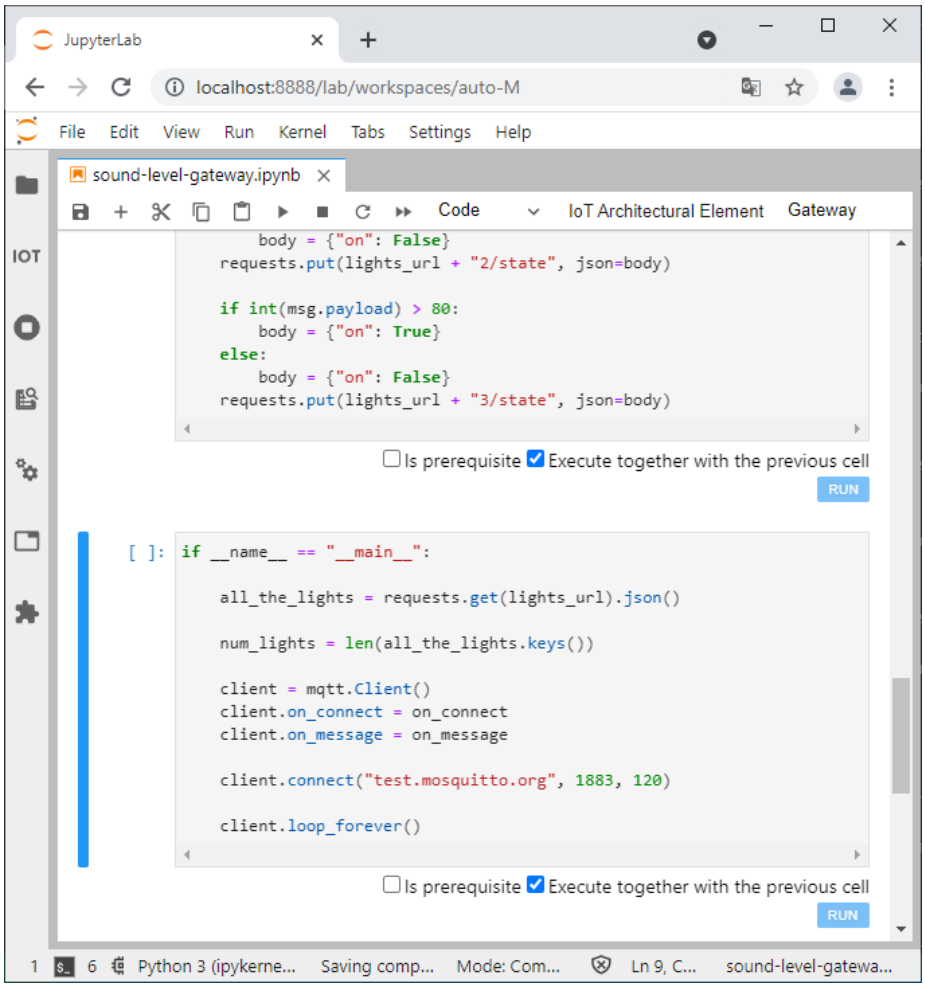



Figure 8: Split the code across various cells through the `is_linked_previous_cell` property

cell containing the function main has to be executed along with the cells in which the invoked functions are defined. In the particular example illustrated in this Figure, at least three code cells will be executed when the user clicks the run button in the last cell. In Figure 7, the markdown cells with the textual description of the code cells were removed during the screenshot to facilitate its visualization.

## 7. Exploratory User Study

To understand the usability and usefulness of the IoT notebook, we conducted a first-use controlled lab study composed of two sessions. In the first session, participants were given an implemented IoT project in the form of a set of notebook documents. They were asked to explore the documents, get familiar with the IoT notebook features, and execute the project. In the second session, participants were given an IoT project description and were asked to implement it using the IoT notebook.

In both cases, the IoT projects comprised a device, a gateway, and a back-end component. Accurately, the device corresponded to an Arduino board that interacts through MQTT messages; the gateway corresponded to an MQTT client that, once it received a message coming from the Arduino, invoked a back-end function; the back-end, for its part, corresponded to a RESTful API. Coincidentally, both projects closely recall the use case described before, even if with clear differences.

Each study session was time-boxed to one hour. At the end of the second session, participants were asked to complete a brief post-session questionnaire on the perceived usability of the IoT notebook and their first impressions of using it. This exploratory study aims to understand how usable the IoT notebook is for novices and how useful they find the added features. We also wanted to assess if a rich surrounding context in documentation interleaved with executable code increases understanding during the prototyping process.

### *7.1. Participants and treatments*

Our participants comprised thirteen employees (10 male, 3 female, ages 25-30, mean 27, SD 1.62) of a well-known large IT Italian company participating in an IoT and physical computing course. As for the education of participants, 6 held a Bachelor's Degree (B) and 7 a Master's degree (M). Their background ranged across Computer Engineering, Management Engineering, Cinema and Media Engineering, and Aerospace Engineering. Finally, as illustrated in Table 2, no participant had significant experience working with computational

notebooks, and just one of them was familiar with IoT systems at an architectural level.

Moreover, the participants were a class of newly-hired young engineers recruited by an open call with the collaboration of the industry and the regional authorities in the context of a regional program for supporting high-tech sectors that invest in new skills and personnel. Such students were selected through the public call, and for their first semester they were involved for 50% of their time in a training path, while for the rest of the time they were working their regular jobs inside different groups at the company. Both training and work were conducted from home during most of this period due to the pandemic conditions. The training path, in particular, covered a wide range of topics, such as Computer Networks, Product Lifecycle Management, Legislation, Economy, Databases and Data Analysis, and Sensors and IoT.

The user study was conducted as the last class in the 'Sensors and IoT' training module that was 28 hours long. In that module, they were introduced to the main characteristics and challenges of IoT. Besides describing the enabling technologies and protocols used in the IoT, the module outlined IoT-based systems' potentiality and provided examples in the industry, commerce, and home automation. The technical topics addressed by the course included IoT systems architecture and design, Arduino, MQTT, and REST APIs. Each lesson was four hours long, and most of the time, the first half was more theoretical. In the second half, the participants could apply what they had just learned by completing a practical exercise (individually or in groups), with commercially-available tools.

Fortunately, towards the end of the 'Sensors and IoT' training module, the pandemic situation in Italy was improving, and we could yield the last 8 hours in presence, including this user study.

### *7.2. Study procedure*

Before the first session, we obtained participants' consent, gathered their demographic and previous experience information, and provided a walk-through

Table 2: Participants self-rated previous experience (in years)

| ID | Age | Background | Programming (in general) | IoT systems architecture | Physical computing | Computational notebooks | MQTT | Python | HTML/CSS | JavaScript | Java | C |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **P-1** | 27 | Management Engineering (B) | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ |
| **P-2** | 29 | Management Engineering (M) | 1-3 | $< 1$ | $< 1$ | $< 1$ | $< 1$ | 1-3 | $< 1$ | $< 1$ | $< 1$ | $< 1$ |
| **P-3** | 30 | Aerospace Engineering (B) | 1-3 | $< 1$ | $< 1$ | $< 1$ | $< 1$ | 1-3 | $< 1$ | $< 1$ | $< 1$ | 1-3 |
| **P-4** | 25 | Management Engineering (M) | 1-3 | $< 1$ | $< 1$ | $< 1$ | $< 1$ | 1-3 | $< 1$ | $< 1$ | $< 1$ | 1-3 |
| **P-5** | 27 | Physics (B) | 1-3 | $< 1$ | $< 1$ | $< 1$ | $< 1$ | 1-3 | $< 1$ | $< 1$ | $< 1$ | 1-3 |
| **P-6** | 26 | Biomedical Engineering (M) | 4-5 | 1-3 | 4-5 | 1-3 | 1-3 | 4-5 | 4-5 | 4-5 | $< 1$ | 4-5 |
| **P-7** | 25 | Management Engineering (M) | 4-5 | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | 1-3 | 1-3 |
| **P-8** | 29 | Aerospace Engineering (M) | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | 1-3 | $< 1$ |
| **P-9** | 26 | Bioinformatics (M) | 1-3 | $< 1$ | $< 1$ | 1-3 | $< 1$ | 1-3 | 1-3 | $< 1$ | $< 1$ | 1-3 |
| **P-10** | 26 | Management Engineering (B) | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ |
| **P-11** | 28 | Mechatronic Engineering (M) | 1-3 | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | 1-3 |
| **P-12** | 25 | Computer Engineering (B) | 1-3 | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | 1-3 | $< 1$ | 1-3 | $< 1$ |
| **P-13** | 28 | Aerospace Engineering (B) | 1-3 | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | $< 1$ | 1-3 |

of the IoT notebook and the study protocol, illustrating its main goals and motivations through a PowerPoint presentation. In the first session, we gave the participants the computational notebook documents corresponding to Project 1.

**Project 1**: we implemented an IoT project that periodically (every 5 seconds) gathers and evaluates the reading from a sound sensor. Based on the measures, a given number of light bulbs are turned on. In this manner, if the sound intensity is low, just one light bulb is turned on; on the contrary, three bulbs are turned on if the intensity is high. As shown in Figure 9 (highlighted in blue), our implementation has two notebook documents. The first document outlines the implementation and deployment process of an Arduino application that:

- gathers the readings from a sensor physically connected to the board,
- connects and subscribes to an MQTT topic, and
- sends an MQTT message with the last reading value.

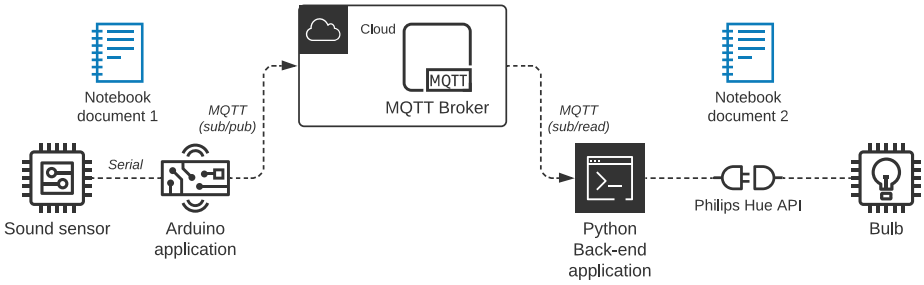



Figure 9: Project 1 architecture (with the implemented notebook documents indicated)

The second notebook document has the implementation and deployment process of a Python Back-end application that:

- connects and subscribes to the same MQTT topic as the Arduino, and
- interacts with the Phillips Hue API to turn on the light bulbs.

In the second session, we asked participants to implement Project 2 using the IoT notebook platform we provided. They were instructed to work individually, but we did not ban them from collaborating.

**Project 2**: we asked participants to implement an IoT project that depending on whether (i) there are people in a room and (ii) it is raining outside, turns on a light bulb. This implementation requires an occupation sensor communicating via MQTT, and the information about the weather must be gathered from an API. They are free to create the notebook documents they deem necessary, and naturally, they can take inspiration from Project 1.

In both sessions, we wrote down the questions posed by the participants. After the study tasks, participants were asked to complete a post-session questionnaire. The questionnaire included 14 questions with Likert-scale ratings of the tool's perceived usability and utility in solving the assigned task (Table 3). We used questionnaire responses as the basis for our post-study debriefing interview, conducted as a round table. In particular, by a video beam, we projected the anonymous answers for each questionnaire item as frequency graph bars encouraging the participants to provide qualitative justifications and specific anecdotes to supplement their scores. In this way, we aimed to promote the participants' discussion by sharing and confronting their perceptions and experiences.

Table 3: Summary of post-study questionnaire responses averaged over 13 subjects and sorted by mean agreement level on a 5-point Likert scale from Strongly Disagree (1) to Strongly Agree (5).

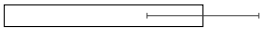
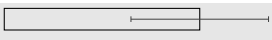
| ID | Questionnaire Item | $\mu$ | $\sigma$ | |
|---|---|---|---|---|
| | | | | 0 1 2 3 4 5 6 |
| IT-1 | The documentation interleaved with lines of code helped me to better understand the implementation strategy and the reasoning behind the code | 4.38 | 0.74 |  |
| IT-2 | I would use the IoT notebook as a means to prototype an IoT system | 4.31 | 0.91 |  |

Table 3: Summary of post-study questionnaire responses averaged over 13 subjects and sorted by mean agreement level on a 5-point Likert scale from Strongly Disagree (1) to Strongly Agree (5). (continued)

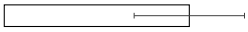
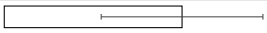
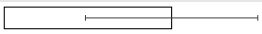
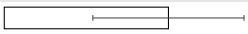
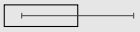
| ID | Questionnaire Item | $\mu$ | $\sigma$ | |
|---|---|---|---|---|
| | | | | 0 1 2 3 4 5 6 |
| IT-3 | I would use the IoT notebook as a means to document my prototyping process | 4.08 | 0.73 |  |
| IT-4 | I prefer the documentation in the IoT notebook than the documentation in the code | 3.92 | 1.07 |  |
| IT-5 | It was easy to understand how did the "Is prerequisite" and the "Execute together with the previous cell" options work | 3.92 | 1.07 |  |
| IT-6 | Working through the computational notebook increased my confidence that I was doing each step correctly | 3.85 | 1.10 |  |
| IT-7 | The use of the IoT notebook encouraged me to document my own prototyping process | 3.85 | 0.77 |  |
| IT-8 | The grouping of the notebook documents in the left sidebar helped me to better understand the architecture of the IoT prototype | 3.77 | 0.80 |  |
| IT-9 | The definition of prerrequisite cells helped me to avoid missing configuration requirements | 3.69 | 1.14 |  |
| IT-10 | The use of the IoT notebook, other than following instructions, helped me to document my own prototyping process | 3.69 | 0.91 |  |
| IT-11 | The use of the IoT notebook helped me to distinguish the configuration and implementation steps | 3.62 | 1.00 |  |
| IT-12 | It was easy to execute the code | 3.46 | 1.22 |  |
| IT-13 | The use of the IoT notebook helped me gain a broader perspective of the interaction among diverse components | 3.38 | 0.84 |  |

Table 3: Summary of post-study questionnaire responses averaged over 13 subjects and sorted by mean agreement level on a 5-point Likert scale from Strongly Disagree (1) to Strongly Agree (5). (continued)

| ID | Questionnaire Item | $\mu$ | $\sigma$ | 0 1 2 3 4 5 6 |
|---|---|---|---|---|
| IT-14 | Working throught the computational notebook was frustrating | 1.62 | 0.74 |  |

*7.3. Results*

The most notorious observation was that the participants highly appreciated the approach of the computational notebooks, specifically having documentation interleaved with lines of code (first question, **IT-1**, Table 3). They perceived that it effectively helped them better understand the implementation strategy and the reasoning behind the code. Especially if considering that, besides textual explanations written in the notebook, the documentation might include links to other websites, YouTube videos, and images. Furthermore, in some cases, the documentation concerned the expected outputs so that after running the cell, the user had the chance to verify if the execution was successful. This outcome is consistent with the fact that participants prefer the documentation in the IoT notebook to the documentation in the code (**IT-4**). As one of them commented during the post-study debriefing interview:

> *"You understand better the cells of code because the initial comment explains well what the lines of code should do, and seeing later the execution makes it understand even better."*

Nevertheless, since participants were conscious that the consistency between the code and the documentation and the completeness of the textual descriptions are fundamental to support non-expert IoT developers effectively, they suggested that:

> *"It would be nice to do like GitHub, and every time you edit the code, a pop-up comes up related to the text below and gives you the option*

*to edit the text (like doing a commit). In this manner, the tool would induce the user to update the textual descriptions constantly."*

Additionally, participants agreed in the questionnaire that they would use the IoT notebook to prototype and document an IoT system (**IT-2** and **IT-3**):

*"With a closed environment like this, it is much faster to prototype something. You don't have to worry about copying and pasting things from the web or from one development environment to another."*

However, concerning the documentation, there was also who expressed that:

*"I'd rather document at the end. If it were some development from scratch, I would do it with an IDE, and in case someone not very technical wants to visualize steps, I would do a notebook."*

From this discussion emerged that the notebook should not compete with popular IDEs. Instead, a few participants envisioned the possibility to implement a plugin that would enable the integration of an IDE with the IoT notebook. In particular, they envisioned that within an IDE (such as Visual Studio Code), the developer should have the opportunity to select a piece of code, right-click over it and select from a context menu an option to create a code cell in the notebook, automatically. Additionally, after adding the code cell, the plugin would open by default a text editor to enable the developer to write the corresponding textual description. Naturally, that description would be included in the notebook as a markup cell.

Similarly, regarding **IT-4**, where most of the participants agreed that they preferred the documentation in the IoT notebook rather than the documentation in the code, the same improvement suggestion emerged. The participants who choose the documentation in the code suggested enabling, through a plug-in, to automatically export/transfer the comments on the code files.

Most of the participants understood how did the "Is prerequisite" and the "Execute together with the previous cell" options work (**IT-5**) and how do they

differentiate. There were no additional comments or questions regarding such questionnaire item. Regarding **IT-6** and **IT-7**, all participants agreed in general terms that the IoT notebook increased their confidence that they were doing each step correctly and that the use of notebooks encouraged them to document their prototyping process. Indeed, while working on the second project, we observed that many participants were structuring their notebook documents by adding titles, subtitles, and descriptions other than the code cells.

Due to a technical inconvenience with the browser where the participants opened the IoT notebook, the button to run the code cells when working with the Kernelino got disabled. For this reason, participants disagreed on the questionnaire with the **IT-12** (Executing the code was easy). Nevertheless, during the debriefing interview, they were asked if, apart from that particular inconvenience, the running button and the mechanics of executing the cell were clear, and all of them answered positively. Indeed, they did not have any inconvenience running the cells in the other notebook documents. However, a participant suggested that the run button should be enlarged because it was not evident to him how to run the code cell, at the beginning.

When asked if the use of the IoT notebook as frustrating (**IT-14**), participants disagreed: they found it helpful and they understood the implemented features. Concerning the features provided by the IoT notebook, the participants did not agree much that using the IoT notebook helped them gain a broader perspective of the interaction among diverse components (**IT-13**). This disagreement is also connected with question **IT-8** since the left sidebar was the graphical interface component that we introduced to support the visualization of the diverse architectural elements present in the IoT system prototype. In the debriefing interview, two observations emerged. On the one hand, there is the fact that the projects used during the study were not big enough to appreciate the utility of this feature:

> *"In this particular project, it was not so useful since there were only two notebooks. Anyway, I understood the functionality."*

On the other hand, instead of representing the architectural elements into a sidebar, the participants agreed that they would prefer a more graphical representation. They suggested that:

> *"An architectural graphical view of the system was somehow missing, to avoid reading the text and have an impact view right away. It would be nice to integrate a special kind of notebook where the components of the IoT system prototype are graphically represented and clicked to open the corresponding notebook document where each one is implemented."*

In summary, the answers to the questionnaire items and the discussion in the post-study debriefing interview enabled us to ascertain that the participants perceived the IoT notebook as a helpful and usable tool. It guided them well through the development process of a prototype IoT system. They acknowledged that they would use this approach, particularly for learning and teaching purposes, when getting started with IoT development. Additionally, two main improvement suggestions emerged from the round table. Firstly, to change the visualization of the architectural elements by introducing a special kind of notebook in which the architecture of the prototype IoT system is displayed graphically (as a diagram) to better provide a broader perspective of the interaction among diverse components and enhance the understanding of the architecture. Secondly, to implement a plug-in aimed at achieving a seamless integration with currently available IDEs. Instead of using the IoT notebook as an IDE, the participants suggested developing integration with a traditional IDE and giving the option to the users to export and document their code files to a literate computing scheme. In this regard, users seem to lean towards developing prototypes from scratch in their preferred IDE and then sharing their process through a computational notebook. This way of working is consistent with notebooks weaknesses that have been identified in the literature [29, 16], mainly associated with poor code management affordances. Specifically, such limitations concern lack of code assistance; the impossibility to explore the API

and functions of external libraries; the little-to-no support for finding, removing, updating, or identifying deprecated packages; the difficulty in debugging due to out-of-order execution and the consequent impossibility to add breakpoints to follow the code flow. Consequently, if the integration suggested by the participants is achieved, developers would benefit from all the code management features that traditional IDEs provide. Furthermore, the linking between cells and the order constrained could be managed by the plug-in that, at this point, would know how the complete code file looks like and would automatically generate such constraints.

Further, at the end of the first session, we collected all the notebook documents that the participants implemented. In total, we collected 17 notebook documents. Ten of them corresponded to the Arduino application. It gathers readings from the occupation sensor, connects to an MQTT topic, and sends a message marking if there are people in the room. Seven of them corresponded to the Python Back-end application. It subscribes to the same MQTT topic as the Arduino application, determines if it is raining by invoking a weather API, and, based on the data sent by the Arduino application, interacts with the Philips Hue API to turn on the application. Four participants were able to work on both documents. Naturally, all the participants tended to structure their implementation according to the notebook documents we gave them in the first session. In this sense, for time constraint reasons, they were allowed to start their development not from scratch but upon these notebooks. Similarly, the participants were indicated which weather API they could use. This weather API was chosen by us because, instead of using complex authentication protocols such as OAuth, it just required a key that the users could get after registering.

Table 4 summarizes the characteristics of the collected notebook documents. As the most outstanding observations, the participants were receptive to documenting the code cells. However, just adding a header over a set of cells was enough for some of them (5 notebook documents). On the contrary, some other participants tried to add a descriptive text per code cell (7 notebook docu-

Table 4: Participant's notebook documents characteristics

| Characteristics | Notebook documents (out of 17) |
|---|---|
| Textual explanations | 12 |
| Just headers | 5 |
| Headers and descriptive texts | 7 |
| Architectural elements | 14 |
| Compilable | 13 |
| Functional | 8 |
| `is_prerequisite` | 12 |
| `is_linked_previous_cell` | 12 |
| `background_execution` | 8 |
| `is_library_installation` | 10 |

ments). Almost all the notebooks had specified the IoT architectural element to which they belong (14 notebook documents). When trying to execute them, 13 notebook documents were compilable. Three notebook documents had syntax errors that did not enable their execution. Since notebooks do not provide the same support as a traditional IDE, participants were probably not conscious of these errors. The other notebook document was not complete. Eight out of the seventeen notebook documents were correctly implemented. The background_execution was less used among the new fields we introduced in the IoT notebook. From our point of view, we consider it is due to its low visibility and the fact that it might be confused among the other options that we also represented graphically with a checkbox under the code cell. Indeed, this hypothesis is consistent with the comments that emerged during the round table described above.

### *7.4. Study Limitations and Future Work*

The main limitation of our study is that it was exploratory in nature, so we cannot make any rigorous claims about the specific effects of the IoT notebook

on the IoT systems prototyping process. However, this study represented a first assessment of the utility and usability of a set of features that we retained necessary for the IoT notebook. Consequently, more than conducting an exhaustive large-scale study, the goal was to get feedback from non-expert developers.

In the same vein, time constraints forced us to design exercises with a difficulty level to be realistically implemented in three hours. This limitation led us to propose projects involving just a few computational notebooks. To some extent, it prevented the participants from finding the utility of features like grouping the documents. Indeed one of the participants stated that:

> *"Short time to get familiar with the tool, but I would like to use it again in the future."*

Consequently, further validation with a more open-ended use of the tool and fewer time restrictions would be necessary to dive into aspects that could provide new insights. In particular, a workshop in which participants are asked to prototype an IoT system of their ideation would be ideal.

Finally, we consider that the round table format that we used in the debriefing interview was very enriching since the participants had the chance to discuss among themselves their impressions, and most importantly, to find agreements on how the improvement suggestions for the IoT notebook. However, that format might also – even if their attitude did not give us that impression – inhibit the participation.

## 8. Conclusion

In this work, we propose a set of features that an IoT-tailored literate computing approach should satisfy to support the prototyping of IoT systems. We implemented a first version of what we called *IoT notebook* integrating such features, and we validated its usefulness and usability by conducting an exploratory usability study among non-expert developers. The study enabled us to ascertain that the participants perceived the IoT notebook as helpful and

usable, by guiding them through the development process of a prototype IoT system. They acknowledged that they would use this approach, particularly for learning and teaching purposes, when getting started with IoT development.

However, since the IoT landscape is vast and involves an enormous number of artifacts, platforms, devices, applications, protocols, and domains, we recall that our proposal frames in a specific context: prototyping relatively simple IoT systems by non-expert developers. In this sense, by "IoT systems", we consider systems composed of the four interconnected architectural elements listed by Taivalsaari *et al.* [3] (i.e., cloud services, applications, gateways, and devices). Similarly, by "simple", we refer to focusing on the data exchanged among the different components. Therefore, the prototype IoT systems users can develop using the IoT notebook use contextual data to trigger some functionality. Naturally, our proposal does not support prototyping all kinds of IoT applications (supposing that such delimitation could be done), and the IoT notebook does not represent a viable deployment platform. Indeed, we expect non-expert programmers to use the IoT notebook at an early development stage to: understand the steps they must follow to configure and implement a prototype; verify how the various components might interact (in what concerns data exchange); document and share their prototyping. Finally, a more open-ended use of the tool is required to evaluate our approach's suitability further.

## References


[1] F. Corno, L. De Russis, J. P. 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, ACM, New York, NY, USA, 2018, pp. 13–16. doi:10.1145/3183377.3183385.
URL http://doi.acm.org/10.1145/3183377.3183385

[2] A. Taivalsaari, T. Mikkonen, On the development of iot systems, in:

2018 Third International Conference on Fog and Mobile Edge Computing (FMEC), 2018, pp. 13–19. doi:10.1109/FMEC.2018.8364039.

[3] A. Taivalsaari, T. Mikkonen, A roadmap to the programmable world: Software challenges in the iot era, IEEE Software 34 (1) (2017) 72–80. doi:10.1109/MS.2017.26.

[4] P. Selonen, A. Taivalsaari, Kiuas – iot cloud environment for enabling the programmable world, in: 2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2016, pp. 250–257. doi:10.1109/SEAA.2016.10.

[5] F. Corno, L. De Russis, J. P. Sáenz, On the challenges novice programmers experience in developing IoT systems: A survey, Journal of Systems and Software 157 (2019) 110389. doi:10.1016/j.jss.2019.07.101.

[6] F. Corno, L. De Russis, J. P. 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. 1, 2017, pp. 250–255. doi:10.1109/COMPSAC.2017.186.

[7] S. Oney, J. 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, ACM, New York, NY, USA, 2012, pp. 2697–2706. doi:10.1145/2207676.2208664.
URL http://doi.acm.org/10.1145/2207676.2208664

[8] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, C. Willing, J. development team, 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/

[9] A. Rule, A. Tabard, J. D. Hollan, Exploration and explanation in computational notebooks, in: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, CHI '18, ACM, New York, NY, USA, 2018, pp. 32:1–32:12. doi:10.1145/3173574.3173606.
URL http://doi.acm.org/10.1145/3173574.3173606

[10] F. Corno, L. De Russis, J. P. Sáenz, Towards computational notebooks for iot development, in: Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems, CHI EA '19, ACM, New York, NY, USA, 2019, pp. LBW0154:1–LBW0154:6. doi:10.1145/3290607.3312963.
URL http://doi.acm.org/10.1145/3290607.3312963

[11] D. E. Knuth, Literate programming, Comput. J. 27 (2) (1984) 97–111. doi:10.1093/comjnl/27.2.97.
URL http://dx.doi.org/10.1093/comjnl/27.2.97

[12] J. F. Pimentel, L. Murta, V. Braganholo, J. Freire, A large-scale study about quality and reproducibility of jupyter notebooks, in: Proceedings of the 16th International Conference on Mining Software Repositories, MSR '19, IEEE Press, Piscataway, NJ, USA, 2019, pp. 507–517. doi:10.1109/MSR.2019.00077.

[13] M. B. Kery, M. Radensky, M. Arya, B. E. John, B. A. Myers, 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, ACM, New York, NY, USA, 2018, pp. 174:1–174:11. doi:10.1145/3173574.3173748.
URL http://doi.acm.org/10.1145/3173574.3173748

[14] A. 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).

[15] M. Borowski, J. Zagermann, C. N. Klokmose, H. Reiterer, R. Rädle, Exploring the Benefits and Barriers of Using Computational Notebooks for Collaborative Programming Assignments, Association for Computing Machinery, New York, NY, USA, 2020, p. 468–474.
URL https://doi.org/10.1145/3328778.3366887

[16] S. Lau, I. Drosos, J. M. Markel, P. J. Guo, The design space of computational notebooks: An analysis of 60 systems in academia and industry, in: 2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2020, pp. 1–11. doi:10.1109/VL/HCC50065.2020.9127201.

[17] A. Rule, I. Drosos, A. Tabard, J. D. Hollan, Aiding collaborative reuse of computational notebooks with annotated cell folding, Proc. ACM Hum.-Comput. Interact. 2 (CSCW) (2018) 150:1–150:12. doi:10.1145/3274419.
URL http://doi.acm.org/10.1145/3274419

[18] J. F. Pimentel, L. Murta, V. Braganholo, J. Freire, Understanding and improving the quality and reproducibility of jupyter notebooks, Empirical Software Engineering 26 (4) (2021) 65.

[19] M. Källén, T. Wrigstad, Jupyter notebooks on github: Characteristics and code clones, The Art, Science, and Engineering of Programming 5 (3) (Feb 2021). doi:10.22152/programming-journal.org/2021/5/15.
URL http://dx.doi.org/10.22152/programming-journal.org/2021/5/15

[20] J. Wang, L. Li, A. Zeller, Better code, better sharing: On the need of analyzing jupyter notebooks, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER '20, Association for Computing Machinery, New York,

NY, USA, 2020, p. 53–56. doi:10.1145/3377816.3381724.
URL https://doi.org/10.1145/3377816.3381724

[21] M. Beth Kery, B. A. Myers, Exploring exploratory programming, in: 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2017, pp. 25–29. doi:10.1109/VLHCC.2017.8103446.

[22] A. Head, F. Hohman, T. Barik, S. M. Drucker, R. DeLine, Managing messes in computational notebooks, in: Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI '19, ACM, New York, NY, USA, 2019, pp. 270:1–270:12. doi:10.1145/3290605.3300500.
URL http://doi.acm.org/10.1145/3290605.3300500

[23] D. Yin, Y. Liu, A. Padmanabhan, J. Terstriep, J. Rush, S. Wang, 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, ACM, New York, NY, USA, 2017, pp. 18:1–18:8. doi:10.1145/3093338.3093378.
URL http://doi.acm.org/10.1145/3093338.3093378

[24] M. V. Merino, J. Vinju, T. 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, ACM, New York, NY, USA, 2018, pp. 210–214. doi:10.1145/3276604.3276981.
URL http://doi.acm.org/10.1145/3276604.3276981

[25] A. Azzarà, D. Alessandrelli, S. Bocchino, M. Petracca, P. Pagano, Pyot, a macroprogramming framework for the internet of things, in: Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014), 2014, pp. 96–103. doi:10.1109/SIES.2014.6871193.

[26] B. Rubell, Overview. CircuitPython with Jupyter Notebooks. Adafruit Learning System, https://learn.adafruit.com/

circuitpython-with-jupyter-notebooks, online; last accessed September 23, 2019 (2018).

[27] CircuitPython, CircuitPython, https://circuitpython.org/, online; last accessed September 23, 2019 (2019).

[28] F. Perez, B. E. Granger, Ipython: A system for interactive scientific computing, Computing in Science Engineering 9 (3) (2007) 21–29. doi:10.1109/MCSE.2007.53.

[29] S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, T. Barik, What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities, Association for Computing Machinery, New York, NY, USA, 2020, p. 1–12.
URL https://doi.org/10.1145/3313831.3376729

[30] M. Verano Merino, J. Vinju, T. van der Storm, Bacatá: Notebooks for dsls, almost for free, The Art, Science, and Engineering of Programming 4 (3) (Feb 2020). doi:10.22152/programming-journal.org/2020/4/11.

[31] R. Rädle, M. Nouwens, K. Antonsen, J. R. Eagan, C. N. Klokmose, Codestrates: Literate computing with webstrates, in: Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology, UIST '17, Association for Computing Machinery, New York, NY, USA, 2017, p. 715–725. doi:10.1145/3126594.3126642.
URL https://doi.org/10.1145/3126594.3126642

[32] F. Corno, L. De Russis, J. P. Sáenz, How is open source software development different in popular iot projects?, IEEE Access 8 (2020) 28337–28348. doi:10.1109/ACCESS.2020.2972364.

[33] X. Larrucea, A. Combelles, J. Favaro, K. Taneja, Software engineering for the internet of things, IEEE Software 34 (1) (2017) 24–28. doi:10.1109/MS.2017.28.

[34] D. Koop, J. Patel, Dataflow notebooks: Encoding and tracking dependencies of cells, in: 9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2017), USENIX Association, Seattle, WA, 2017.

[35] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, 3rd Edition, Addison-Wesley Professional, 2012.

[36] Arduino, GitHub - arduino/arduino-cli: Arduino command line interface, `https://github.com/arduino/arduino-cli`, online; last accessed September 23, 2019 (2019).

[37] C. Rossant, IPython Interactive Computing and Visualization Cookbook, Packt Publishing, 2014.

[38] Jupyter Development Team, Making simple Python wrapper kernels, `https://jupyter-client.readthedocs.io/en/stable/wrapperkernels.html`, online; last accessed September 23, 2019 (2015).

[39] The ZeroMQ authors, ZeroMQ, `https://https://zeromq.org/`, online; last accessed September 23, 2019 (2019).