Increasing the Feedback on IoT Development in Node-RED

Diogo Luís Rey Torres

July 13, 2020
Increasing the Feedback on IoT Development in Node-RED

Diogo Luís Rey Torres

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Gil Manuel Magalhães de Andrade Gonçalves
External Examiner: Prof. Ângelo Manuel Rego e Silva Martins
Supervisor: Prof. André Monteiro de Oliveira Restivo

July 13, 2020
Abstract

We live in a world where technology takes a big part in our lives. In order to promote a better quality of life, we automated manual tasks, making us depend even more on machines. As a consequence, Internet-connected devices are everywhere, capable of sensing and actuating in the real world. This is known as the Internet of Things (IoT), the current peak of ubiquitous computing and connectivity.

Creating and managing these systems can be simplified through the abstraction of low-level details (e.g., heterogeneity) into a more high-level logic. Some tools achieve this through visual metaphors, e.g., Visual Programming Languages (VPLs).

Node-RED is one of the most widespread VPLs targeting IoT systems that mashups hardware devices, APIs, and third-party services, in a hybrid text-visual programming approach. With Node-RED, it is possible to create flows with the aim of composing a set of rules that the system has to comply with. These flows can cover a wide variety of needs, such as turning on the coffee-machine whenever the user wakes up and turning on the heating system when the user is returning home after work.

The inherent complexity of these systems makes them difficult to understand and Node-RED, after the system is deployed, becomes relatively opaque. It does not provide any feedback on what is happening inside the nodes and their side-effects, neither if a connection between nodes is “correct” until deployment and execution. In fact, Node-RED provides almost zero debug capabilities besides the usual “log to console” strategy leading to the addition of debug nodes, thus, requiring the system re-deployment. However, we found that these problems have been at least partially solved in other areas, such as video/image rendering and game development.

We proposed modifying Node-RED to present a more complete and immediate feedback about the system by observing its messages flowing through the nodes, using different visual metaphors, and providing enhanced debugging mechanisms through breakpoints and runtime modifications (i.e., injecting and changing messages). With these modifications, we want to explore if an increase of the observability and testability of the system towards a more “live” setting improved the ability of users to successfully develop their systems.

To validate this approach, our solution was tested against a physical setup and a device simulator, which we developed to easily reproduce sensors and actuators and also validate the correctness of experimental scenarios. We then executed a controlled experiment with 20 participants, by providing them a set of tasks under two different treatments (with the original Node-RED, and with our solution), for which they needed to maintain, evolve or build an IoT system. We verified that, overall, such enhancements reduced the development time by 10%, the number of failed attempts to deploy the system by 15% and the number of deployments by 51%, concluding that our tool reduces the human errors that emerge during development and also helps with the maintenance of the system.

Keywords: Internet of Things, Node-RED, Visual Programming, Observability, Monitoring, Live Programming, Debugging
Resumo

Vivemos num mundo em que a tecnologia ocupa uma grande parte das nossas vidas e, para promover uma melhor qualidade de vida, automatizamos uma série de tarefas, tornando-nos ainda mais dependentes das máquinas. Como consequência, estamos rodeados de dispositivos ligados à Internet, que são capazes de recolher dados e de atuar em tempo real. Este fenómeno é conhecido como Internet das Coisas (IoT), o auge da computação e conectividade. A criação e manutenção destes sistemas é complexa, mas pode ser simplificada através da conversão de detalhes de baixo nível, numa lógica de mais alto nível. Algumas ferramentas conseguem fazê-lo através de metáforas visuais, p.e., Linguagens de Programação Visual (VPLs).

O Node-RED é uma das VPLs mais utilizadas para sistemas IoT que integra dispositivos, APIs e serviços de terceiros, usando uma programação texto-visual híbrida. Com o Node-RED é possível criar fluxos para definir um conjunto de regras a que o sistema tem de obedecer. Estes fluxos podem cobrir uma vasta gama de necessidades tais como, ligar a máquina de café sempre que o utilizador acorda, ligar o A/C quando a temperatura está acima de um certo valor, entre outras.

A complexidade inerente a estes sistemas dificulta a interpretação do que está a acontecer e, assim que o sistema entra em funcionamento, o Node-RED torna-se relativamente opaco. Isto é, não fornece nenhuma informação prévia acerca do que se está a passar no interior dos nodos, nem se uma conexão entre nodos está “correta”. De facto, o Node-RED fornece muito poucos recursos de depuração, para além da estratégia “imprimir para a consola”. Esta estratégia leva à adição de nodos de depuração e, consequentemente, a novos deployments do sistema. No entanto, encontramos que alguns destes problemas foram parcialmente resolvidos em outros domínios, como renderização de imagem/vídeo e desenvolvimento de jogos.

Para colmatar estes constrangimentos propusemo-nos modificar o Node-RED de modo a torná-lo capaz de apresentar mais informação sobre o sistema através da observação das mensagens que fluem pelos nodos, usando diferentes metáforas visuais e providenciando melhores mecanismos de depuração através de pontos de interrupção e modificações durante a execução (injeção e modificação de mensagens). Com as propostas introduzidas foi pretensão verificar se um aumento da observabilidade e testabilidade do sistema, rentabiliza a capacidade dos utilizadores para construir, fazer evoluir e manter os seus sistemas operacionais.

Numa primeira fase, as funcionalidades da nossa solução foram testadas num sistema físico e num simulador de dispositivos, especificamente por nós desenvolvido para facilitar a reprodução de sensores e atuadores e também para verificar a correção dos cenários experimentais. Posteriormente, foi feita uma experiência controlada com um grupo de 20 participantes. A experiência consistiu em completar um conjunto de tarefas, nas duas versões do Node-RED (a original e a nossa). Com este procedimento foi possível verificar que as alterações introduzidas permitem reduzir o tempo de desenvolvimento em 10%, o número de tentativas até tornar o sistema operacional em 15% e o número de deployments em 51%. O que permitiu concluir que a nossa ferramenta reduz o número de erros que emergem durante o desenvolvimento e que constitui uma ajuda na manutenção destes sistemas.

Palavras-chave: Internet das Coisas, Node-RED, Programação Visual, Observabilidade, Monitorização, Programação Em Tempo Real, Depuração
Acknowledgements

First, I would like to thank my supervisors André Restivo and Hugo Sereno Ferreira for their friendship, guiding me whenever I needed a light, putting me in constant learning path and for their useful critiques of this research work. I would also like to express my gratitude towards João Pedro Dias for his elaborate suggestions to the project and document, and for teaching me some basics of electronics, even though he was not required to.

I also had the great pleasure of working with my dissertation colleagues from “IoTices” and I want to thank them for the help and motivation throughout this entire process.

To all my friends from “Um sargento nunca abandona os seus soldados” for the amazing friendship, “futeboladas” and late nights during these last years. I’m grateful for meeting them and following me along this entire journey, never leaving anyone behind. I want to especially acknowledge João Damas for the extra support and very valuable comments in this dissertation.

To my gym buddies and influencers in this industry for motivating me by sharing their struggles and teachings. I would also thank my current gym for being like a second home for me, even with the closing by the COVID-19 pandemic.

I want to thank all my other friends, teachers and family. This accomplishment would not have been possible without them.

Finally, I want to give my deep gratitude for all the obstacles that appear in life, that even though they are difficult and hard, they have made and sculpted me to what I am today.

Diogo Torres
“No matter the obstacle that you go through in your life, there is one thing you can be sure of... It is designed to make you better.”

Bradley Martyn
# Contents

1 Introduction 1
   1.1 Context ................................................................................. 1
   1.2 Motivation ............................................................................. 2
   1.3 Problem Definition ................................................................. 2
   1.4 Objectives ............................................................................. 3
   1.5 Document Structure ............................................................... 3

2 State of the Art 5
   2.1 Internet of Things ................................................................. 5
      2.1.1 Architecture .................................................................. 6
      2.1.2 Enabling Platforms ........................................................ 7
      2.1.3 Applications ................................................................... 8
      2.1.4 Key Challenges ............................................................... 9
   2.2 Programming Frameworks for IoT .......................................... 11
      2.2.1 General Purpose Frameworks ......................................... 11
      2.2.2 Visual Programming Frameworks .................................... 11
   2.3 Monitoring of IoT Systems .................................................... 15
      2.3.1 Visual Programming Languages ...................................... 15
      2.3.2 Node-RED Environmental Monitoring ............................ 16
      2.3.3 Node-RED Observability ............................................... 18
   2.4 Visual Programming in Other Domains .................................... 20
      2.4.1 Blender ........................................................................... 20
      2.4.2 Unreal Engine ................................................................. 21
      2.4.3 Nodes ............................................................................ 22
   2.5 Summary ............................................................................... 24

3 Problem Statement 25
   3.1 Current Issues ........................................................................ 25
   3.2 Motivational Scenario ........................................................... 26
   3.3 Proposed Solution ................................................................. 27
   3.4 Hypothesis ............................................................................ 28
   3.5 Research Questions ............................................................... 28
   3.6 Research Methodology .......................................................... 28
   3.7 Summary ............................................................................... 29

4 Proposed Solution 31
   4.1 Overview ............................................................................... 31
   4.2 Implementation Overview ..................................................... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3 Modifying Node-RED’s UI</td>
<td>34</td>
</tr>
<tr>
<td>4.4 Showing Messages</td>
<td>36</td>
</tr>
<tr>
<td>4.4.1 Messages Plot</td>
<td>37</td>
</tr>
<tr>
<td>4.4.2 Messages History</td>
<td>37</td>
</tr>
<tr>
<td>4.5 Injecting Messages</td>
<td>38</td>
</tr>
<tr>
<td>4.6 Breakpoint System</td>
<td>39</td>
</tr>
<tr>
<td>4.7 Enhancing the Debug Node</td>
<td>40</td>
</tr>
<tr>
<td>4.8 Summary</td>
<td>41</td>
</tr>
<tr>
<td>5 Validation</td>
<td>43</td>
</tr>
<tr>
<td>5.1 Overview</td>
<td>43</td>
</tr>
<tr>
<td>5.2 Experimental Goals</td>
<td>45</td>
</tr>
<tr>
<td>5.3 Experimental Parameters</td>
<td>45</td>
</tr>
<tr>
<td>5.4 Tasks</td>
<td>46</td>
</tr>
<tr>
<td>5.5 Analysis</td>
<td>48</td>
</tr>
<tr>
<td>5.5.1 Control Task</td>
<td>48</td>
</tr>
<tr>
<td>5.5.2 Experimental Tasks</td>
<td>49</td>
</tr>
<tr>
<td>5.5.3 Post-Test Questionnaire</td>
<td>53</td>
</tr>
<tr>
<td>5.6 Discussion</td>
<td>53</td>
</tr>
<tr>
<td>5.7 Threats to Validity</td>
<td>54</td>
</tr>
<tr>
<td>5.8 Summary</td>
<td>55</td>
</tr>
<tr>
<td>6 Conclusions</td>
<td>57</td>
</tr>
<tr>
<td>6.1 Summary</td>
<td>57</td>
</tr>
<tr>
<td>6.2 Hypothesis Revisited</td>
<td>58</td>
</tr>
<tr>
<td>6.3 Main Contributions</td>
<td>59</td>
</tr>
<tr>
<td>6.4 Future Work</td>
<td>59</td>
</tr>
<tr>
<td>A Example of Data Stored in editor-client</td>
<td>61</td>
</tr>
<tr>
<td>B Sensor/Actuator Simulator</td>
<td>63</td>
</tr>
<tr>
<td>C Experimental Guide</td>
<td>67</td>
</tr>
<tr>
<td>D Validation Environment</td>
<td>77</td>
</tr>
<tr>
<td>E Validation Questionnaire</td>
<td>79</td>
</tr>
<tr>
<td>References</td>
<td>83</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Smart home devices interaction ........................................ 6
2.2 Typical three-tier architecture of Internet of Things systems ................................. 7
2.3 A Raspberry Pi and an Arduino ........................................ 8
2.4 IoT applications domains ............................................... 9
2.5 Popularity of visual programming solutions for IoT ........................................... 12
2.6 Node-RED flow to handle Twitter posts .................................. 13
2.7 Rule engine and Dashboard of ThingsBoard ........................................ 16
2.8 Node-RED flow and dashboard applied to air monitoring ................................. 17
2.9 Node-RED flow and dashboard applied to seawater purification ............................ 18
2.10 Node-RED flow and dashboard applied to air quality monitoring ....................... 18
2.11 Node-RED Debug .......................................................... 19
2.12 Monitoring architecture with Node.js and Prolog ...................................... 19
2.13 Blender Nodes Editor ..................................................... 20
2.14 Blender Settings of a Node ............................................... 21
2.15 Blender Node Groups ..................................................... 21
2.16 Unreal Engine Blueprints ............................................... 22
2.17 Blueprints Call Stack and Execution Trace ......................................... 22
2.18 Nodes Editor ............................................................... 23
2.19 Nodes Dynamic Node Comments ........................................ 23
3.1 Node-RED flow to handle a heating system ........................................ 26
3.2 Mock-up of the proposed solution ........................................ 27
4.1 CAULDRON User Interface ................................................ 32
4.2 CAULDRON Detailed Flow ................................................ 32
4.3 CAULDRON Detailed Node ................................................ 33
4.4 Node-RED WebSockets Communication ......................................... 34
4.5 CAULDRON Messages Plot ............................................... 37
4.6 CAULDRON Last received message ....................................... 37
4.7 CAULDRON Messages History ........................................... 38
4.8 CAULDRON Inject Message ............................................... 38
4.9 CAULDRON Breakpoint System .......................................... 39
4.10 Node-RED Edit Debug Node .............................................. 40
5.1 SmartLab Components Diagram ........................................... 44
5.2 Control Tasks Solution Node-RED flow ..................................... 46
5.3 Experimental Tasks Solution Node-RED flow .................................... 47
5.4 Box plot of time and number of deployments in Control Task ....................... 48
5.5 Box Plot of time spent per Experimental Task ...................................... 50
5.6 Box plot of number of deployments per Experimental Task ............................................. 51
5.7 CAULDRON feature usage analysis through number of clicks ................................. 52
5.8 Questionnaire Post-Test Results .................................................................................. 53
List of Tables

4.1 Node-RED sub-modules description ........................................... 33
4.2 Node-RED events description ................................................. 35
5.1 Time spent and number of deployments in Control Task ................. 48
5.2 Time spent per Experimental Task ........................................... 49
5.3 Number of deployments per Experimental Task ............................. 50
5.4 Number of verification requests per Experimental Task .................... 51
5.5 CAULDRON usage analysis by task .......................................... 52
B.1 Sensor/Actuator Simulator - Routes available ............................... 65
List of Listings

A.1 Example of data stored on editor-client module .......................... 61
B.1 Example of a sensor’s data file in the simulator .......................... 63
B.2 Example of an actuators file in simulator ................................. 64
B.3 Example of a validation file in the simulator ............................. 64
D.1 Docker Compose configuration file to the user study setup .............. 77
Abbreviations

API    Application Programming Interface
I/O    Input/Output
IEC    International Electrotechnical Commission
ISO    International Organization for Standardization
IoT    Internet of Things
JSON   JavaScript Object Notation
MQTT   Message Queuing Telemetry Transport
REST   Representational State Transfer
RFID   Radio-Frequency Identification
SBC    Single-Board Computers
VPL    Visual Programming Language
WSN    Wireless Sensor Networks
Chapter 1

Introduction

1.1 Context 

The Internet of Things has been emerging in the last years, permeating our daily lives. Xia et al. [74] define IoT as the networked interconnection of everyday objects equipped with ubiquitous intelligence. The pervasiveness of these smart objects around us creates a foundation for a more interactive environment between things and humans, with the potential and promise of improving the quality of life.

The IoT paradigm, with its unique characteristics — communication, identification, and interactivity — is present in various domains, such as Home Automation, Transportation, Manufacturing, Healthcare, Farming, and Retail [53, 45]. With the proliferation of the Internet of Things in broader domains, Mckinsey [42] estimates that one trillion IoT devices will be deployed by 2025 and this area will have an economic impact of $11 trillion per year by the same year. The growing number of IoT systems and their increasing complexity, which can be observed in aspects such as
the multitude and continuous growth in diversity of communication protocols, architectures, and
development approaches, together with the pervasiveness in application domains, has led to several
shortcomings, including the lack of human resources having the technical knowledge needed to
develop IoT systems [43].

Visual Programming Languages (VPLs) are commonly used for creation and management for
these systems, allowing users to define the system’s behavior by manipulating visual elements
rather than text. Among these solutions, we can highlight Node-RED, an open-source tool that
allows the mashup of hardware devices, APIs, and third-party services, in a hybrid text-visual
programming approach [50], as one of the most used [22].

1.2 Motivation

In an attempt to tackle both the growing complexity of developing IoT systems and the lack of
specialized resources, several approaches have been proposed, by both industry and academia,
empowering the so-called end-user development, allowing users with little to no proficiency in
the area to develop and configure their IoT systems. Among those, the most common are visual
programming solutions.

Among the available solutions, Node-RED is one of the most popular examples of a visual
approach that allows the creation of flows, connecting nodes that represent the various elements of
a system (e.g., sensors, actuators) in order to compose a rule set the system must follow. However,
as the system evolves, potentially increasing in complexity, understanding what is happening can
be difficult, as Node-RED lacks in presenting feedback to the user during development. This makes
it difficult for a user to create and modify existing rules, while ensuring that changes do not break
the expected behavior [27].

1.3 Problem Definition

Node-RED does not provide any feedback on what is happening inside each node, neither does it
allow the injection or modification of messages at runtime, or even to check if a connection between
nodes is “correct” until deployment and execution. Moreover, every change that the user makes to
the system, such as adding debug nodes, requires a new deployment.

Therefore, users, have a difficult task in their hands because they do not want to break the
current system with new changes. Additionally, if their system is not working properly, they want to
quickly understand the reason behind that. Node-RED provides almost zero debugging capabilities
besides the usual “log to console” strategy. This strategy could lead to an increase of accidental
complexity where the flows become larger and more complex due to an increase of non-essential
nodes and a substantial amount of logs. Thus, it became common for Node-RED users to resort to
external solutions that provide visualization and monitoring mechanisms. These mechanisms make
the system observable to a certain degree, often through processes of analysis [7, 18].
1.4 Objectives

Even when considering other less popular solutions for developing IoT systems, we encounter similar problems, including lack of observability (i.e., feedback between the development environment and the system under development) and weak, or nonexistent, mechanisms to properly debug the system.

1.4 Objectives

The main goal of our work is to explore if a visual programming tool with more feedback and improved debugging capabilities is able to enhance the ability of users to successfully build, evolve and maintain their IoT systems. To achieve that, we will use Node-RED and extend it to be able to present: (a) the runtime state of the system (i.e., observing input/output of each node), (b) debugging mechanisms (i.e., breakpoints) and (c) runtime modifications (i.e., injecting and changing messages).

We expect our solution to decrease the effort needed to develop an IoT system by improving its observability, testability and sanity/safety-checking. The advantages provided by this enhanced system are also expected to reduce human errors that emerge during the development process and help monitor the system while it is running.

1.5 Document Structure

The rest of this dissertation is structured as follows:

- Chapter 2 (p. 5), State of the Art, mainly divided into three parts: an in-depth background of the Internet of Things, work related to the visual programming platforms of IoT systems, and how monitoring has been applied in visual platforms.

- Chapter 3 (p. 25), Problem Statement, focuses on the issues that this project aimed to resolve, describing the proposed solution to solve, or at least diminish, the problem in question. Along with that, the techniques used to measure this project’s success are also presented.

- Chapter 4 (p. 31), Proposed Solution, gives an overview of the solution and describes the implementation of each feature in detail.

- Chapter 5 (p. 43), Validation, presents the empirical evaluation process applied and the answer to each research question.

- Chapter 6 (p. 57), Conclusions, summarizes the work developed, the main contributions and future improvements.
Chapter 2

State of the Art

2.1 Internet of Things

The Internet of Things (IoT), according to ISO/IEC 20924, is an “infrastructure of interconnected entities, people, systems and information resources together with services which processes and reacts to information from the physical world and virtual world” [1].

However, the term ‘Internet of Things’ was first coined by Kevin Ashton in a presentation in 1999 on supply-chain management. He has mentioned that “The Internet of Things has the potential to change the world, just as the Internet did. Maybe even more so” [29]. Also, during his period as executive director at MIT’s Auto-ID Center, he contributed to the extension of Radio-Frequency Identification (RFID), which served as the basis for the early development of IoT [12].

From its inception, IoT is characterized as a utopia of ubiquitous computing and connectivity [19]. Any object that is capable of connecting to the Internet and able to communicate with
other entities falls under the umbrella of IoT. Initially, RFID was the dominant technology behind IoT development, but, with further advances in technology, Wireless Sensor Networks (WSN) and Bluetooth-enabled devices boost the mainstream adoption of the IoT trend [12]. Nowadays, ZigBee and Wi-Fi are used in WSN because of their energy-efficient design and their low cost [38, 68, 67].

2.1.1 Architecture

IoT systems are generally composed of devices acting as providers or consumers of data associated with the physical world. These devices can be categorized into sensors and actuators, where sensors sense data and actuators respond to events [59]. The main focus of these systems is on data collected and transmitted rather than point-to-point communications. This resulted in the possibility of the adoption of architectures and principles focused on content [45].

The core workflow of IoT is described by Khan et al. [33] and shown in Figure 2.1:

1. **Object sensing, identification and communication**: The information is the sensed data, according to the type of sensors. This information is then transmitted to another object.

2. **Trigger an action**: The received information is handled by a device that then invokes the appropriate action according to its configured rules.

3. **Provide feedback**: The system provides monitoring capabilities to the user administrator, being able to observe the current system status and the result of the executed actions.

![Image](image.png)

Figure 2.1: Smart Home devices communicating through a wireless network that can be controlled via PC, phone or voice activation [75].

The architecture of IoT systems normally follows a three-tier structure, as explained in Figure 2.2 (p. 7). This three-tier architecture has been described in the architectural pattern Fogxy and is used in several IoT systems [59].

The need for an intermediate tier is due to the fact that traditional IT cloud computing models do not satisfy the needs of IoT systems. For most situations, these needs are not met due to limitations in bandwidth, latency, heterogeneity between devices and the increasing volume of the
2.1 Internet of Things

![Diagram of IoT architecture]

Figure 2.2: Typical three-tier architecture of Internet of Things systems [20]. The higher tier consists of servers, normally available in the cloud, that have capabilities to store, process, and offer services to the lower tier. The lower tier is constituted by devices that can sense and actuate over the environment. Finally, the intermediate tier is composed of gateways that ensure the communication between the previous tiers. From top to bottom the latency decreases.

Data being generated, transmitted and analyzed. Therefore, Fog Computing is used as a solution for the challenges mentioned earlier, focusing on distributing data across the IoT system, as closely as possible to the edge tier [20].

Other architectures have been defined as they are needed. Khan et al. [33] describe an architecture composed of five layers: (1) perception layer, (2) network layer, (3) middleware layer, (4) application layer, and (5) business layer. The two architectures mentioned above exposes two different perspectives on the same problem. However, they are not incompatible and can be applied together.

2.1.2 Enabling Platforms

Single-Board Computers (SBCs) are typically part of the fog tier. Moreover, they are built on a single circuit board that includes microprocessor(s), memory, I/O and others, thus encompassing everything needed in a full computer. These devices are able to run complex operating systems such as different Linux distributions. One example is the Raspberry Pi, one of the most popular SBCs [20].

On a similar concept, Single-Board Microcontrollers are also built on a board but with the memory and processor embedded. With less computational power than SBC, they only provide the minimum requirements for computing and control tasks. However, these devices are also capable of running simple operating systems like FreeRTOS\(^1\) and RIOT\(^2\). In IoT systems, these boards

---

\(^1\)FreeRTOS, [https://www.freertos.org/](https://www.freertos.org/)

\(^2\)RIOT, [https://www.riot-os.org/](https://www.riot-os.org/)
equipped with communications microchips are typically used in the lower tier [20]. Arduino boards are an example of these devices [62].

Figure 2.3 shows one example of each device.

Both the Pi community and the Arduino community are quite active [69, 16], and software support is always assured. Furthermore, they have developed interesting projects like a smart home application on Arduino [31] or an Internet Weather Station on Raspberry [58]. The first one uses various sensors to collect information about light, temperature, humidity, sound, rain, soil moisture, and motion. Additionally, based on this information an event-based system was developed. This system informs the users via mobile phone whenever an event related to a subscribed topic by them occurs. The sensors nodes use a Wi-Fi card to communicate to the Arduino board. Then, on the central server, a rule engine (Drools\(^3\) is used in this study) reacts according to the previous configurations. Finally, the Weather Station was built to provide an inexpensive and functional solution for real-time monitoring climate. Savic et al. [58] use a DHT11 sensor to extract the humidity and a BMP180 sensor to atmospheric pressure and air temperature. They also calculate a relative altitude based on the pressure value. Lastly, a Raspberry Pi is used as a server to collect all the data.

However, most of these devices are either quite expensive or large in terms of weight and size. Thus, new microcontrollers appeared with low price and good performance properties. Currently, the ESP32 and its predecessor ESP8266, which are powerful microcontrollers with Bluetooth and Wi-Fi, are widely used in a large variety of IoT applications [41, 67].

### 2.1.3 Applications

IoT aims to stimulate new types of interactions among things and humans. These new types of interaction induce in a better quality of life and resource management. Namely, health services are easily accessible, the city management and its infrastructure are less onerous, and it is easier to recover from system disasters [20].

\(^3\)Drools Rule Engine, [https://www.drools.org/](https://www.drools.org/)
IoT can offer competitive solutions over the current traditional ones in several fields. Asghari et al. [6], in a systematic literature review, considered six IoT application domains that encompass health-care, environmental, commercial, smart city, industrial and general aspects. The smart city approach has the highest percentage of the application domains with 30% usage, followed by health-care applications having 20%, commercial applications 14%, environmental applications 12%, general applications 12% and finally industrial applications 10% usage in IoT. These values are shown in Figure 2.4.

Figure 2.4: This chart, based on the results provided by Asghari et al. [6], shows the percentage of IoT applications based on his usage.

### 2.1.4 Key Challenges

Several authors recognize some challenges in the IoT area [63, 6, 45]. Khan et al. [33] group and state the most important ones that are briefly described below:

**Naming and Identity Management**: Due to a large number of objects in an IoT system and the need for each object to have a unique identity over the Internet, it is necessary to have an efficient naming and identity management system to dynamically allocate and manage these identifiers.

**Interoperability and Standardization**: Most of the devices are from different manufacturers that use their technologies. These technologies may not be available to others and standardization is necessary to provide better interoperability for all devices.

**Information Privacy**: An IoT system is composed of many objects that use different types of object identification technologies like RFID, 2D-barcodes, among others. Furthermore, it is required to take appropriate privacy measures and prevent unauthorized access.
**Objects safety and security**: IoT systems are composed of several devices, spread over a specific area, that are aware of its environment. Consequently, it is essential to prevent the intruder’s access to these devices so they cannot change their operation or physically damage them.

**Data confidentiality and encryption**: The large amount of data generated and distributed by sensors should be encrypted to guarantee data integrity. Additionally, the IoT service should have the ability to determine who has the authorization to access the data.

**Network security**: Due to the large amount of data sent by devices over the network, the transmission system should be able to handle such amount of data without losing it as a result of network congestion. Then, proper security measures should be taken to prevent external interference or monitoring.

**Spectrum**: The devices need an exclusive channel to transmit data over the network. However, the spectrum available to allocate these channels is restricted. Therefore, it is necessary to develop an efficient allocation mechanism to allow a huge number of devices to communicate over a wireless network.

**Greening of IoT**: The energy consumption by an IoT system is rising due to the increasing number of devices, thus increasing the amount of data sent over the network. Moreover, this data transmission rate is getting higher and higher. Therefore, it is necessary to adopt measures to make the network as energetically efficient as possible.

To achieve the full potential of the IoT paradigm, it is necessary to address these challenges and develop technological solutions for tackling them.
2.2 Programming Frameworks for IoT

IoT devices are commonly characterized by having limited storage and processing capacity, which may not be capable of processing some heavy tasks by themselves. Thus, to bypass the limitations of these devices, some of these tasks are moved to the cloud. The cloud can provide the computational power to complete these heavy tasks through their back-ends and also presenting a web-based front-end to interact with the user in an intuitive and pleasant way [12].

Since IoT has a very complex domain [39, 32, 40] and it is still in its early stages, most of the frameworks designed to construct these systems are also in a similar state [12].

2.2.1 General Purpose Frameworks

Buyya et al. [12] present a set of minimal features that must be satisfied by these frameworks:

**Coordination**: It is necessary to orchestrate activities from the different computing elements that play different roles.

**Heterogeneity**: The large diversification of IoT devices should be handled efficiently by the framework.

**Scalability**: The systems developed should handle a massive number of users who would benefit from their deployment. Consequently, these frameworks need to support a variety of programming patterns and implement dynamic load balancing.

**Fault-tolerant**: IoT devices sometimes go offline due to energy issues or another kind of failure. Moreover, the framework should allow developers to build applications that have self-healing capabilities.

**Lightweight footprint**: The framework should have a low impact on computational resources during runtime and the developers should have a small learning curve to learn how to develop in these systems.

**Support for latency-sensitive applications**: IoT, due to its nature, will have many applications geographically distributed and thus may be latency-sensitive. However, pushing all the computations to the cloud will not help make this application feasible, and it requires that the programming framework supports these sorts of requirements dynamically.

2.2.2 Visual Programming Frameworks

Visual Programming Languages allow users to create programs by manipulation of visual images representing objects rather than by specifying them textually. For instance, some VPLs could also be considered flow-based programming. These types of programs are composed of flows that allow the exchange of data in the form of messages through predefined connections. These connections act as a black-box process, abstracting the data transformation that occurred [8].
The Wiley Encyclopedia of Computer Science and Engineering [20] defines VPL as:

“A language in which significant parts of the structure of a program are represented in a pictorial notation, which may include icons, connecting lines indicating relationships, motion, color, texture, shading, or any other non-textual device.”

Such programming language makes the task of computer program development easier for programmers but also accessible to non-expert programmers [46]. Moreover, visual programming has enabled faster prototyping and application development. In fact, users can focus more on what functionalities they want in their programs, since they do not have to invest so much time in learning a new programming language [77].

Considering IoT systems are built upon highly heterogeneous architectures and the use of visual metaphors allows an abstraction of low-level details into a more high-level logic, an increase in visual-based programming approaches has been witnessed over the years [22].

Dias et al. [22] conducted a study that measures the popularity, using the number of stars in GitHub\(^5\), of some open-source visual programming tools, previously identified by Ray et al. [54]. They conclude that the most popular visual programming tool for IoT is Node-RED\(^6\). Additionally, we joined another tool designated Thingsboard and presented the results in Figure 2.5.

![Figure 2.5: Number of stars on GitHub for visual programming solutions for IoT.](https://github.com)  
\(^5\)GitHub, https://github.com  
\(^6\)Node-RED, http://nodered.org
Node-RED

Node-RED is an open-source web-based tool, initially developed by IBM’s Emerging Technology Services team and now a part of the JS Foundation. It allows developers to wire together hardware devices, APIs and online services using a visual flow-based programming model, and a drag-and-drop interface. Node-RED has as basis Node.js, taking full advantage of its event-driven, non-blocking model. Its “programs” are treated as flows and consist of nodes connected by wires.

A node is the basic building block of a flow. Flows are the parts of a system that logically group sequences of functionalities in order to complete tasks. After the user develops these flows, they are deployed. Node-RED provides an export mechanism (JSON serialization) to help this deployment. The Node-RED community is quite active and they have a library with various node implementations for completing a variety of tasks, like reading values from a database or receiving the feeds from a Twitter account (as depicted in Figure 2.6). Also, this tool is simple to extend. One example implemented by Blackstock et al. provides multiple run-time environments.

However, Node-RED, despite its features and popularity, has limitations. For example, there is no proper mechanism to debug and test the developed flows. Given that IoT systems are typically large-scale and complex by nature, it is easy to end up with highly complex flows that are prone to faults complicating the task of identifying and fixing bugs.

Figure 2.6: Node-RED flow to handle Twitter posts

---

8 OpenJS Foundation, https://openjsf.org/
9 NodeJS, https://nodejs.org/
10 JSON, https://www.json.org/
Other Tools

Despite Node-RED being our focus, we also briefly present other tools.

**XOD**: A visual programming language for microcontrollers that outputs native code for the target platform. In this language, each node has typed inputs and outputs. This allows the tool to check the link between two nodes to pass data: the link is only created if the data types are compatible, thus allowing the program to always compile. Moreover, it is possible to create and share nodes with the community [76].

**Ardublock**: A VPL based on Blockly\textsuperscript{11}. It allows users to connect and visually program the Arduino platform to create an IoT-based application [5]. It does not appear to be actively maintained, though, as the last commit on GitHub dates to November 2017.

**S4A**: A modified version of the Scratch\textsuperscript{12} visual programming language that allows for simple programming of the Arduino platform. It provides new blocks for managing sensors and actuators connected to Arduino [56].

**Wyliodrin**: A browser-based visual programming environment that allows the development of IoT systems on several devices. It also provides shell access and textual programming languages. It works by installing the Wyliodrin server in the device, which provides remote control capabilities and a dashboard for visualizing the data collected [73].

**Minibloq**: A graphical development environment for Arduino and other platforms. This tool contains all the toolchain necessary to compile and deploy the code to the selected hardware target [44].

**NetlabToolkit**: A simple web interface to connect sensors, actuators, and others for the development of quick prototypes. Devices are represented as nodes, which are connected to other nodes to communicate. The tool also allows users to add custom JavaScript snippets and create their reusable widgets. Arduino and newer Linux embedded systems like Raspberry Pi are supported [52]. Like Ardublock, this tool’s development is not very active, considering its repository has not been updated since December 2017.

**ThingsBoard**: ThingsBoard is an open-source IoT platform that can do data visualization but also has a rules engine, similar to the Node-RED engine. This data visualization tool is made through a dashboard that has many different visual metaphors. Additionally, it provides the ability to review incoming and outgoing messages for each node. Like in Node-RED, it is also possible to export the rule chain to JSON format [66].

This is not an extensive list, but rather a sample that allows an overview of the current open-source tools and their main features.

\textsuperscript{11}Blockly, https://developers.google.com/blockly  
\textsuperscript{12}Scratch, https://scratch.mit.edu/
2.3 Monitoring of IoT Systems

Monitoring is the task of collecting and displaying data from a given system. Nowadays, monitoring has evolved to deliver detailed metrics, tracing, and alerting on performance and user experience across the entire technology stack, including the cloud [55]. Furthermore, with the collected data, it is also possible to perform meaningful analysis that improves the capacity of understanding the system [70]. However, monitoring does not exist if the system is not observable. In fact, monitoring is only achieved when data is made available by the system.

Over the years, systems have become increasingly complex resulting in increased efforts by programmers to understand them, thus causing an increase in the number of human-induced errors and failures [22]. Therefore, the system’s observability becomes extremely important for the success of software teams. Giving the ability to see all the data in one place and in real-time leads to a faster resolution of the system’s issues since they have a better understanding of what caused the issue [55].

Internet of Things has been acknowledged as an example of such complexity [39, 32, 40]. As a result, some authors have proposed the usage of monitoring to be able to identify faulty components in the system, caused by bugs in the code or hardware problems or to observe some controlled environments.

Dias et al. [19] refer that taking advantage of Live Programming [64] in these systems may reduce the waiting between a programming action and seeing its effect on program execution by simplifying the detection of behavior changes at runtime. This simplification could be done through the reduction of the number of phases of the traditional program development cycle (edit, compile, link, run) to one phase that involves the program constantly running, even as various editing events occur.

2.3.1 Visual Programming Languages

As referred previously, the use of visual programming techniques has become widely adopted in the IoT area with the intention of improving the development of these systems. However, over the past years, many of the visual programming tools focused on the rapid prototyping of systems, having fundamental gaps in the mechanisms that allow the evolution and maintenance of these systems, including debugging and runtime observation. The only way the user has to debug is through the source level debug information provided, which required a greater effort to understand the system and what is going on, than the original visual programming task [77].

Most of the tools previously described in Section 2.2.2 (p. 11) do not provide a way to observe the running system. Those that usually do, incorporate it in a separate component from the development one like ThingsBoard [65]. This tool has a rules engine and a dashboard decoupled from each other, as depicted in Figure 2.7 (p. 16).

Other tools, such as Kibana [13] or Grafana [37], allow for not only the monitoring of the system device’s values (e.g., values of sensors and position of actuators) but also its usage (e.g.,
(a) A rule engine, similar to Node-RED rules, consists of colorful nodes connected by wires.

(b) A dashboard where it is observable the sensors readings through different visual metaphors.

Figure 2.7: ThingsBoard

...middleware device processor usage and battery levels). With such tools, all this monitoring can be done remotely with a web interface that displays all that information in a friendly way.

However, since our focus is on Node-RED we will describe some of the research made in this area. In research, only two topics are frequently mentioned. The first one refers to the monitoring of environments (Section 2.3.2) and the second to observe the system in terms of visualizing the information flowing through it and also react according to predefined rules (Section 2.3.3, p. 18). However, none of the topics addresses the increase of observability in this tool during development.

2.3.2 Node-RED Environmental Monitoring

Environmental monitoring describes the processes needed to characterize and monitor the quality of the environment, more specifically the air, water, and soil [28].

Shinde et al. [61] develop an IoT system for indoor and outdoor environmental monitoring.
The system consists of two sensor nodes and a gateway. Node-RED, installed in a Raspberry Pi, is used to collect data from the sensor node at regular intervals and then publishes on a cloud (IBM Cloud\textsuperscript{13}). The sensors, present in this system, measure humidity, temperature and CO$_2$ concentration presented in the air. In addition, this system can alert the user if the sensors’ data do not respect limits previously imposed by them. The data is displayed through the Node-RED dashboard \cite{14}.

Node-RED dashboard is a Node-RED package that provides a set of nodes to quickly create a live data dashboard. With the use of additional nodes in the current program, it is possible to create a dashboard independent of Node-RED that allows the user to observe his system in the form of values, graphics, among other forms of visualization, as illustrated in Figure 2.8.

![Figure 2.8](image-url)

Figure 2.8: This figure shows on the left the rules to collect and publish data about air characterization and on the right the respective dashboard where it is possible to observe this data in the form of values and graphics, as presented by Shinde et al. \cite{61}.

Kalpana et al. \cite{30} adopt a similar approach to setup the process of purification of seawater, which is used for the boiler and cooling tower. This process is made automatically by using a Debris filter with the help of a Raspberry Pi 3 and Node-RED. The control shifts from auto to manual mode when the value of DP (difference in pressure) surpasses 120. This is assured by comparing the detected values with the reference value in Node-RED. In addition, this system contains a layer of security where only authorized people can have access. All the data collected is sent to the cloud where users can watch the information on DP level, time, among others. This information is displayed using the Node-RED Dashboard package, as shown in Figure 2.9 (p. 18).

Finally, the work of Kodali et al. \cite{36} focuses on monitoring the air quality of a specific region. This collected data is then sent to people who can use it to improve the quality of living of citizens. This work uses a Wio Link board\textsuperscript{14}, and various grove sensors, which are connected to the Wio Link, \textit{e.g.}, humidity and temperature sensor (DHT-11), air-quality and light sensors. Then, Node-RED, installed in a Raspberry Pi, is used to publish the data over an MQTT broker. Moreover, every time that data is published, an SMS, an e-mail and a Twitter post are sent depicting the values of various sensors. The dashboard and the correspondent flow are illustrated in Figure 2.10 (p. 18).

\textsuperscript{13}IBM Cloud, \url{https://www.ibm.com/cloud}

\textsuperscript{14}Wio Link board, \url{https://www.seeedstudio.com/Wio-Link.html}
2.3.3 Node-RED Observability

Node-RED natively comes with debug nodes that display messages in the debug sidebar within the editor, as shown in Figure 2.11 (p. 19). Alongside each message, the debug sidebar includes information about the received time of the message and which Debug node sent it [49]. This type of implementation is also referred by Zodik et al. [77] that said that this set of debug nodes are able to generate progress reports during execution that include content and structure messages that are displayed to the user.

However, in complex systems, with several nodes, adding several debug nodes that write to the same sidebar makes the debugging process very difficult.

Nonetheless, the literature points to similar approaches, where some authors implement their own debug nodes, though with some external components to process the information coming from them. Ancona et al. [4] describe a way to implement runtime monitoring on Node-RED with trace expressions. In this work, they instrument the source code of the program using the Jalangi

---

15 Trace expressions can be language and system agnostic through the notion of event domain and type.
Figure 2.11: The *Debug* node, represented on the left, contains a button to enable/disable its output. On the right, a Debug sidebar where all the information (time, node id and message) provided by these nodes are displayed [49].

A framework [60] that is able to capture all relevant events for the domain in use. Additionally, they have a Prolog\(^\text{16}\) server implementing the operational semantics of trace expressions and offering a simple REST interface, shown in Figure 2.12. In the Node-RED implementation, they add a new node called *Monitor* that works in a similar way as the *Debug* node. In conclusion, with the help of this tool, it is possible to recover from errors through the dynamic checking of the correct behavior of the system performed by the Prolog monitor, which verifies if the events captured at runtime are compliant with the expected behavior expressed with trace expressions.

![Diagram of the monitoring process](image)

Figure 2.12: The Node.js code is instrumented by the Jalangi framework, which allows whenever an event is triggered, it is sent through an HTTP request to the server. Then, the server parses the event using trace expressions and reacts if necessary [4].

\(^{16}\text{SWI-Prolog, https://www.swi-prolog.org/}\)
2.4 Visual Programming in Other Domains

The literature, as of the time of writing, does not explore monitoring in the area of bringing liveness during development but as an independent tool/component to observe the system. However, in other areas, like image and video edition or game development, there already are visual debugging tools that allow a more interactive debugging process (e.g., using breakpoints and highlighting the messages’ path), runtime modifications, and monitoring at the application level. Debugging these systems is cumbersome because the programmer needs to imagine the underlying logic. Moreover, in flow-based environments, where data is constantly flowing through the system, it is fairly difficult, if not impossible, to have a clear and complete view of the application state and its messages’ flow [17]. We found examples in Blender [26], Unreal Engine [25] and Nodes [51] that provide some of these characteristics.

2.4.1 Blender

Blender is an open-source 3D creation suite that can be used to create 3D visualizations such as still images, video, and real-time interactive video games. It is a cross-platform application and has relatively lower hardware requirements compared to similar tools. Its interface uses OpenGL\(^\text{17}\) to provide a consistent experience across all supported hardware and platforms [10].

We will focus on the Nodes Editor (Figure 2.13) that provides feedback in real-time during development. This editor supports three types of nodes, each one with its own specific purpose: Shader Nodes, Composite Nodes and Texture Nodes [11].

![Figure 2.13](image)

Figure 2.13: In Nodes Editor, it is possible to observe inside each node their information such as title, input/outputs, settings, and the values in different visual metaphors like values or plots.

All nodes in Blender have a similar structure. They are constituted by a title, sockets, inputs, outputs and settings. The title displays the name or type of the node by default and it is possible to customize it. The socket’s input and output values from the node are color-coded according to the

\(^{17}\text{OpenGL, https://www.opengl.org/}\)
2.4 Visual Programming in Other Domains

The data type (e.g., color, numeric, vector, shader) it handles. Inputs are positioned on the bottom left side of the node, while outputs are on the top right side. The nodes also have settings positioned in between inputs and outputs, as shown in Figure 2.14. These settings alter the way how incoming information is processed, possibly resulting in different outputs. Besides the previous features, this editor allows to group nodes to simplify a node tree by allowing instancing and hiding parts of the tree, as depicted in Figure 2.15.

Figure 2.14: The settings of a given node that are possible to modify and see its current value.

Figure 2.15: On the left, the interface that allows editing Node Groups. On the right, the corresponding node called Mix Shader.

2.4.2 Unreal Engine

Unreal Engine 4 is a game engine written in C++ and developed by Epic Games. This engine provides a tool called Blueprints, which enables the creation of gameplay elements through classes that are created in-editor through wiring function blocks and property references together. Blueprints also provides a debugger that is capable of pausing the execution of the game and step through the graph nodes by using breakpoints, as shown in Figure 2.16 (p. 22). This debugger allows to see the current flow of the messages and also see their value and other variables in each node. Moreover, it
provides a Call Stack and an Execution Trace that shows a list of the nodes executed and allows further inspection about variables and function outputs, as depicted in Figure 2.17.

Figure 2.16: In Blueprints debugging mode, it is possible not only to watch the values from each node and the current path of the messages but also use breakpoints.

Figure 2.17: On the left, it is possible to observe the call stack of a current program, and some items can expand. On the right, it is displayed the current execution trace.

2.4.3 Nodes

Nodes is a web-based 2D canvas for computational thinking based on JavaScript. This tool is still in the development phase, so there is no public release available. However, through its website and available videos and screenshots, we were able to verify that this tool provides some feedback during development.

This tool uses a hybrid approach mixing a node-based with a text-based environment where it is possible to connect nodes, change their values and verify all these changes automatically at runtime, as shown in Figure 2.18 (p. 23). It is also possible to run multiple branches of the graph at
the same time. Each node is represented in JSON format that allows the user to save it or just copy and paste through different browser windows. To help the debugging process, all nodes that contain errors are automatically highlighted in the graph, allowing a quick find and fix. During this process, this tool preserves the last working state until all the bugs are solved. Furthermore, to help users to better understand the internal flow of the application, nodes with trigger ports that have not been updated in the last frame are faded out. It is also possible to visualize the CPU usage of each node. Lastly, dynamic node comments allow visualizing the global state at runtime as serialized JSON, which includes loading status, simple plots, images, among others, as depicted in Figure 2.19.

Figure 2.18: On the left, it is presented the current flow to draw a rectangle. In the middle, the visualization of the rectangle. Lastly, on the right, the rectangle node settings where it is possible to change them. Both the visualization and changing settings could be done at runtime.

Figure 2.19: The Dynamic Comments feature where it is possible on each node to observe some information like their current message in a text format (e.g., JSON notation) or even images.
2.5 Summary

IoT brings a new vision where devices not only can communicate with each other and humans, but are also aware of the environment they’re in. This stimulates new types of interaction that allow a better quality of life and resource management. IoT is growing, and developing technological solutions for enabling such a vision is the main challenge ahead of us.

Visual Programming Languages are used in this area to allow the abstraction of low-level concepts and highly heterogeneous architectures that these systems have. Node-RED is the most popular open-source visual programming tool that allows developers to construct programs through a drag-and-drop interface. Its “programs” are treated as flows and consist of nodes connected to perform complex tasks. Nevertheless, this tool has some limitations in terms of debugging and development. In fact, Node-RED does not provide feedback during development or at runtime. During development, it is unknown if the connections between nodes are “correct”. Moreover, at runtime, the only way to visualize a node’s information is through debug nodes that “log to console”. This overloads the system with non-essential nodes, resulting in an increase in accidental complexity, where debugging and understanding becomes harder. Furthermore, there isn’t a way to change or inject messages to check how the system responds to certain events. Finally, every time the flow changes, like adding a debug node, it is necessary to (re)deploy the system.

The need to turn these systems more transparent, in order to ease the process of development, understanding, and maintaining, led some authors to create independent dashboards to visualize all of the system’s information. To achieve that, they use a Node-RED package, that provides a set of nodes to create an independent visualization component. Others created their own dashboards using a similar strategy but with more nodes. In addition, there are other tools solely focused on monitoring these systems. However, all of these solutions rely on an independent component from the development one.

In other areas, like image and video edition or game development, there already are visual debugging tools that allow an observable system during development and at runtime, allowing, e.g., to see the internal state of the node through different visual metaphors, or the current path of incoming messages, and even insert breakpoints to further inspect certain nodes. We believe that these features, adapted to our context, will bring value to Node-RED and its users.
Chapter 3

Problem Statement

This chapter describes the problem and proposed solution. Section 3.1 summarizes the current problems identified by state-of-the-art analysis. Section 3.2 presents a motivational scenario to a better understanding of the exposed issues. Section 3.3 details the solution proposed to address these issues. Section 3.4 describes the hypothesis that will be validated in this dissertation. Section 3.5 contains the questions that will be answered. Section 3.6 overviews the methods that will be used to evaluate the proposed solution. Finally, Section 3.7 summarizes all the points mentioned above.

3.1 Current Issues

In Chapter 2 (p. 5), we observed how there is a considerable amount of visual programming solutions for IoT available in the market. However, they are typically limited in ways similar to Node-RED. For instance, none provides immediate feedback during development or at runtime. Other limitations include:

Observability: No existing tools support visualizing the information that flows through the system in the development component. Consequently, after deployment, users are unable to observe the current state. Therefore, they find it cumbersome to understand it and check if it is reacting as expected, without adding non-essential nodes or resorting to external tools.

Structural Correctness: There is no verification of the correctness of connections between nodes, during development. This implies that potential errors are only reported after deployment, and,
in some cases, errors are only triggered in specific conditions, making it next to impossible to check the system’s correctness (i.e., one must, for example, let the system run for some time in a testbed setup to check for potential errors [21]).

**Runtime Modification:** It is not possible to change messages at runtime to check the system’s reaction to a specific behavior (e.g., when a temperature sensor emits a reading, it should be possible to change it). Thus, it is impossible to inject faulty messages to verify if the system reacts as expected.

**Exploration:** Every change in the system requires its deployment to production, including added debugging nodes. This includes minor changes, e.g., such as changing a branching condition.

### 3.2 Motivational Scenario

To better understand these limitations, we start by presenting a motivational scenario, part of a deployed IoT system:

> Whenever the temperature falls below 22°C, the heating system must turn on until the temperature reaches that value.

Figure 3.1 shows a possible Node-RED flow based on this scenario. After its deployment, its real behavior becomes relatively opaque. Node-RED does not provide a way to observe, debug and easily maintain the system.

![Node-RED flow](image)

Figure 3.1: Node-RED flow that handles a heating system. It contains a temperature sensor node, which communicates its value to a Change node that extracts the temperature’s value. Then, a Switch node checks if its value is below 22°C. If this condition holds, a turn ON message is sent to the heating system otherwise, it is sent a turn OFF message.

More specifically, users could wish to know what messages are flowing through the system, without adding more nodes, in order to verify if these messages are correct (e.g., they could want to check that, if the temperature is above 22°C, the heating system turns off). During development, a tool enforcing structural correctness would allow users to understand, before deployment, that the Switch node only receives numbers instead of string messages. Furthermore, they could want to verify if the heating system is properly reacting by changing the current value of the temperature. Finally, users could want to explore other scenarios without the need to deploy the entire system, like changing the current condition of the Switch node to activate or deactivate the heating system.
3.3 Proposed Solution

Inspired by the existing features on VPLs from other domains (Section 2.4, p. 20), we consider that these issues can be addressed in visual programming solutions for IoT. Hence, we want to modify Node-RED to tighten the feedback loop between the development environment and the system under development.

A more “live” development environment will give users the ability to better understand how the system works or why something was done [2]. For example, with Node-RED, this is only possible by looking at all the configured rules to figure out which one could have caused something to happen. However, with this solution, they will be able to see the values between the nodes in different visual metaphors, reducing the necessary effort in assessing how the system is functioning. Figure 3.2 illustrates a mock-up of how a potential solution would implement visualization, inspired by Blender Editor (cf. Figure 2.14, p. 21) and Nodes (cf. Figure 2.19, p. 23).

Furthermore, with a more observable system, it will be easier to perform debugging, and we will provide a breakpoint system, inspired by Blueprints on Unreal Engine (cf. Figure 2.17, p. 22), as well as the ability to change and inject messages at runtime to verify specific system’s behaviors.

We consider that these enhancements would improve the development of IoT systems, by reducing development time, number of bugs and improving overall system maintenance.

We presented the following desiderata that the solution must meet in order to solve some of the problems identified in Section 3.1 (p. 25):

D1. Observable nodes: The input/output of each node must be observable to ease debugging and understanding of Node-RED systems.

D2. Message change at runtime: The user should be able to modify messages to check if specific parts of the system are working as supposed to.

D3. Multiple visual metaphor support: The visualization of the data should adapt to its datatype.

D4. Breakpoint system: The development environment should provide breakpoints on each node that allow to debug the running system without the need for re-deployment.
3.4 Hypothesis

In tackling the problem of understanding the impact of a development environment with real-time feedback and increased debugging capabilities, we aim to find if, by using such an enhanced tool, programmers are faster and less error-prone, thus simplifying their task in creating, understanding and modifying their IoT systems. As such, we propose the following hypothesis:

“Enhancing the feedback loop between system and development environment improves the ability of users to successfully build, evolve and maintain their IoT system.”

3.5 Research Questions

Taking into account the hypothesis statement, the main research questions of this dissertation are:

**RQ1:** Would users with increased exposure to real-time information about the running system build and manage it faster?

**RQ2:** Does providing users with real-time feedback increase their ability to understand and change existing systems?

**RQ3:** Is an IoT visual programming environment, able to reduce human-induced errors during development by providing real-time feedback?

Since these research questions and hypothesis uses terms that can be ambiguous, we will try to clarify them: **successfully** refers to the system working as expected; and **easier** means the users spend less time developing their systems, while also having a lower trial-error rate (i.e., fewer deployments made).

3.6 Research Methodology

The main purpose of this project is not only to develop an enhanced tool, but also explore if providing users with not yet available Node-RED features facilitates the development cycle. Therefore, the successful implementation of this project will be measured in two ways.

First, the project should contain the features mentioned in Section 3.3 (p. 27). Moreover, a set of use cases will be defined and tested upon multiple scenarios to ensure that the features’ implementation was successful. These features are the focal point of the project and are also the driving forces behind the novelty introduced to the system.

Then, an empirical test will be made to evaluate the performance of the modified Node-RED. The main goal of the study is to understand if it is easier to develop an IoT system using this solution when compared to the original tool. To achieve that, we will have two groups of users who will solve three different problems. In the first one, users will try to understand and debug an existing system. Then, they will extend it by adding new rules. In the last one, users will be tasked
with building a new IoT system. One group will use the solution from this dissertation, and the other will use the original Node-RED, thus constituting the baseline. With this procedure, we will be able to evaluate their speed and trial-error rate (e.g., by measuring the number of deployments made) in order to verify if the modified tool improves the development cycle.

3.7 Summary

This project aims to pave the way for a development environment with real-time feedback in IoT. By enhancing Node-RED with features that (a) are able to present the runtime state of the system (i.e., observing input/output of each node), (b) provide debugging mechanisms (i.e., breakpoints) and (c) allow runtime modifications (i.e., injecting and changing messages), we expect that users will develop their IoT systems with more ease. Some of these features already exist in other areas with successful applications, and they will be used as an inspiration to bring similar functionalities to IoT.

With this solution, users will receive more complete and immediate feedback, helping them, while programming, with information about the system. Furthermore, these systems will provide a set of functionalities that will help users to easily debug them, while also being able to better understand a complex system and change it, if necessary.

To validate this approach, a set of use cases will be defined and tested upon multiple scenarios to ensure that the features’ implementation was successful. Then, a controlled experiment will be made to evaluate the performance of the modified Node-RED comparatively with the original tool.
Chapter 4

Proposed Solution

4.1 Overview

As described in Chapter 3 (p. 25), we aim to solve some issues in Node-RED concerning the lack of observability, the inexistence of an approach to modify messages and the absence of other debugging mechanisms like breakpoints, all of them at runtime. Therefore, considering the current state-of-the-art, more specifically what already exists in visual programming tools applied to other areas (Section 2.4, p. 20), Node-RED was modified in order to augment the system’s observability and tight the feedback loop between the development environment and its runtime, in an effort to improve users’ ability to build, evolve, and maintain IoT systems. This modified tool provides as main features:

- The ability to show the information which flows through the nodes using different visual metaphors.
- The injection of messages at runtime.

This chapter describes the solution and its features in detail. Section 4.1 presents the solution, briefly explaining the main features and its interface. Section 4.2 to 4.7 describe the necessary modifications for Node-RED to support each of the features implemented, explaining its architecture and challenges faced. Section 4.8 summarizes the whole process.
• Enhanced debugging capabilities through breakpoints on each node.

We nicknamed this solution as **CAULDRON**: Capacitating Agile Users with Live Debugging Resources On Node-RED, shown in Figure 4.1. With our approach, each node presents new inputs as they are received. However, in nodes without any input, the output is shown instead. This special case ensures that we can see all the information flowing through all the nodes, regardless of their position in the flow (cf. Figure 4.2). In nodes without inputs, the inject message and the breakpoint functionalities are not available, since they are related to input messages. Using a **Switch** node as example, we can observe the added features in more detail in Figure 4.3 (p. 33).

---

**Figure 4.1: CAULDRON - User Interface.** The same flow from Figure 3.1 (p. 26) as seen when using **CAULDRON** showcasing the new features that provide more information about the system and also more debugging capabilities. To observe the same information with the original Node-RED, it is necessary to add three debug nodes and inspect the debug sidebar (also shown).

**Figure 4.2: CAULDRON - Detailed Flow.** Using the flow from Figure 4.1 as an example, it is possible to observe in the first node the absence of the inject and breakpoint functionalities. Additionally, in each node, messages are shown in text, as well as their values plotted.
4.2 Implementation Overview

Node-RED is an extensive framework and, despite a thorough documentation [48], it is not trivial to modify it. Therefore, there was a need to analyze the source code down to each individual module to develop our solution. This tool is composed of one module called node-red, which is responsible for assembling the six sub-modules shown in Table 4.1.

<table>
<thead>
<tr>
<th>Sub-Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>editor-API</td>
<td>An Express(^1) application that serves editor-client and provides an Admin API used to interact with runtime</td>
</tr>
<tr>
<td>editor-client</td>
<td>The front-end component of the editor, where all the functionalities and events needed to interact with the user are implemented</td>
</tr>
<tr>
<td>nodes</td>
<td>A repository to host all the code related to the default nodes</td>
</tr>
<tr>
<td>registry</td>
<td>An internal registry that contains the nodes and flows</td>
</tr>
<tr>
<td>runtime</td>
<td>The back-end component where all the logic is implemented, including communication via WebSockets with the front-end</td>
</tr>
<tr>
<td>util</td>
<td>Contains some utility functions that are common between various modules</td>
</tr>
</tbody>
</table>
Besides the existence of all these sub-modules, the focus on developing CAULDRON lied on the editor-client and runtime sub-modules. These two sub-modules communicate between them through HTTP Requests and WebSockets. The modifications made to the latter are presented in Figure 4.4 and described in the next subsections.

![Node-RED WebSockets communication](image)

Figure 4.4: Node-RED WebSockets communication between editor-client and runtime. The first four topics already exist. We added the remaining ones with the peculiarity of the debug and breakpoint topic being bidirectional.

Initially, this solution was meant to be as less intrusive as possible by being developed as a Node-RED plugin. However, this was not achievable because the sub-modules presented are very tightly coupled.

In the following sections, each functionality implemented will be described in more detail, including some challenges and the core points of its architecture.

### 4.3 Modifying Node-RED’s UI

In the editor-client sub-module, responsible for the user interface, a different component was created to separate new code from existing one. This component is divided into two files, one related to the logic needed to support the new functionalities, and another with the views used. This code is executed after the editor is initialized and fully running.

CAULDRON changes each node’s interface by enhancing them with new functionalities. The nodes in the editor are SVG\(^2\) elements where it is not possible to directly inject HTML elements. However, a workaround was found by injecting a ForeignObject SVG element, which supports (X)HTML, into the SVG’s node [23]. By taking advantage of the method `getActiveNodes` already available in this sub-module that retrieves the current nodes in the flow, as well as taking advantage

4.3 Modifying Node-RED’s UI

that each node has an ID equal to the SVG element’s ID, this procedure is then repeated for every node. Aside from this initial injection, there are other cases where we need to inject our solution in order to maintain a pleasant user experience.

The modifications to the user interface were aimed to be as less intrusive as possible by applying the current style of the nodes (e.g., adopting its color, size, and borders). However, there are some exceptions related to:

**Node’s width**: In some nodes, where the node’s width is too small due to the absence of labels or the lack of space between the end of the label and the node’s border, the default size’s width was increased to host the debug button. To achieve that, the current draw method of the node was modified.

**Debug button’s position**: Since the icons’ position in the nodes can be on the left or right side, the position of the debug button needs to be set according to that. Moreover, an observer is used to update the debug button’s position in case the node’s label is updated. To achieve that, a ResizeObserver, which reports all the changes on the dimension of an SVG Element, was used by virtue of its performance, as it uses events instead of constantly polling [24].

**ForeignObject size/position**: The object that shows the new functionalities developed needs an initial size and position. The extra information provided by our tool can change its size as a consequence of toggling its display or the message value’s size. Therefore, the use of constant values for sizing brings bad user experience, so an observer in each node was used to correct the size (the same used in the previous exception).

Moreover, the editor-client sub-module has an internal event system and we subscribed some of the available events to mitigate the remaining limitations, as described in Table 4.2.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes:add</td>
<td>Triggered when a new node is added, although this event is triggered on node’s creation and not when it is added to DOM (Document Object Model)</td>
<td>A MutationObserver^3 was added to wait until the node is drawn and, after that, inject the ForeignObject element and disconnect the observer</td>
</tr>
<tr>
<td>workspace:change</td>
<td>Triggered when the current workspace is changed by switching the active tab</td>
<td>Inject the ForeignObject</td>
</tr>
<tr>
<td>editor:save</td>
<td>Related to the modification of the settings of a node, e.g., updating the label, resulting in a change in the node’s width</td>
<td>Update the position and size of the ForeignObject, if necessary</td>
</tr>
</tbody>
</table>

Finally, some of these modifications needed features that are only available with JavaScript ES6. However, the editor-client doesn’t support this version of JavaScript, therefore the libraries
Proposed Solution

responsible for compiling this sub-module were substituted, *i.e.*, Uglify\(^4\) was replaced by Terser\(^5\). While Uglify was discontinued, Terser is a fork that continued its work by supporting ES6 and maintaining compatibility with the original implementation.

### 4.4 Showing Messages

The tool has to have information about the messages that are currently passing through the nodes, in order to make them observable. However, the *editor-client* sub-module does not have this information, so it needs to be passed from the back-end to the front-end.

The debug nodes present in Node-RED use WebSockets\(^6\) to communicate their values to the sidebar, offering a long-lived, bidirectional communication channel between client and server with low overhead and latency. Based on this mechanism, a similar approach was adopted to transmit the input and output messages of each node. Leveraging the already existing communication mechanism between *editor-client* and *runtime*, through the *RED.comms* and the *runtime.events* modules, respectively, a new topic, *monitoring*, was added that allows showing the runtime data (*i.e.*, messages between nodes) in the UI.

On the *runtime* sub-module, the class *Node* is the object that all nodes extend. This class itself inherits from the EventEmitter\(^7\) Class with some methods overridden to handle some events properly. This class was adapted to transmit the messages’ values to the UI in two specific cases: on input and output. Message input in Node-RED can be of two different types: a single message or an array of messages. Their output works similarly, however, nodes can have multiple outputs, so there is a possibility of array (message) nesting. These two cases are handled properly in the back-end and, later, on the front-end to correctly save the values. Furthermore, due to asynchronous problems, each message is cloned before being sent through the sockets. More specifically, since the message’s reference through the flow is always the same and its value is constantly changing, not cloning would result in sending a different value than when sent.

On the *editor-client* sub-module, the information received is parsed and stored in a JavaScript Object. An example is shown in Listing A.1. This object acts as a Map where the key is the node’s ID and the value contains an Object with all the information needed. This object contains a field with an array of input messages, a field with various settings to improve the user experience specific to each node, and a field with an array of outputs. For the latter, each element corresponds to a specific output with an array of messages. For each message, we were interested in the timestamp when it arrived, the payload, the ID, and the datatype of the payload (*e.g.*, object, number, string).

---

\(^4\)UglifyJS, [https://www.npmjs.com/package/uglify-js](https://www.npmjs.com/package/uglify-js)

\(^5\)Terser, [https://www.npmjs.com/package/terser](https://www.npmjs.com/package/terser)


\(^7\)EventEmitter, [https://nodejs.org/api/events.html](https://nodejs.org/api/events.html)
4.4 Showing Messages

4.4.1 Messages Plot

The messages plot allows the visualization of incoming values in two different ways. If the message’s payload type is a number, it displays a line plot where the user can easily observe how its value evolves, as visible in Figure 4.5a. Otherwise, it displays a scatter plot with the message’s frequency, as shown in Figure 4.5b. This frequency allows the user to verify if the node is receiving messages and at what pace these messages are coming. Both of these plots are configured to display only the last 10 messages received.

![Messages line plot](image1)

![Messages scatter plot](image2)

Figure 4.5: Messages Plot Functionality

The main implementation challenge was importing the Chart.js library, given that the editor-client sub-module does not support import or require statements. Hence, it was necessary to download the minified library and compile it alongside the Node-RED code.

4.4.2 Messages History

Aside from the ability of showing messages through a plot in a specific node, this solution also presents the last message received and message history since the editor was opened. Furthermore, it allows clearing this history.

The last input message is available by toggling the expand button, as shown in Figure 4.6. Only string and numbers are displayed. However, when the value is larger than the node’s width, it is trimmed. Exceptionally, objects are represented by ‘[Object object]’ notation.

![Message example](image3)

Figure 4.6: An example of a received message in the Heat System node. The time of the last received message and its corresponding value are shown.

The message history is available through the Expand Button, where a popup is opened, containing all previous messages of the current node, as depicted in Figure 4.7 (p. 38).

---

8Chart.js, [https://www.chartjs.org/](https://www.chartjs.org/)
Proposed Solution

Figure 4.7: The last messages of the Switch node presented in Figure 4.3 (p. 33), according to their input/output status and ordered by timestamp. Each message is displayed with the format: Time-Value.

4.5 Injecting Messages

The injection of a message is done through the input of a specific node. This feature could be used to test some specific system behavior. To ease the process of this injection, the interface shows the last message received, which helps when the payload of a given message is an object or a complex value, as shown in Figure 4.8.

Figure 4.8: The interface used to inject a message in a specific node.

To send the message from the editor-client sub-module to the runtime sub-module through WebSockets, the former was adapted by turning the communication between these two sub-modules bidirectional. Furthermore, the latter needed to subscribe messages that are coming by its clients. This was done when connecting to runtime. These injected messages are sent using the debug topic.
4.6 Breakpoint System

The breakpoint system lets the user “pause” the incoming messages of a given node by queuing them. Additionally, the user can step forward a message at a time and change its payload. It is also possible to clear all the queued messages. This process is shown in Figure 4.9. When the user closes the Node-RED editor, all nodes automatically resume.

(a) The Switch node is in “pause” status. Notice that subsequent nodes have earlier timestamps on their last received messages.

(b) The message’s payload was modified to 50°C.

(c) The message modified is presented in the Turn OFF node, where we can state that the Switch node is working properly.

Figure 4.9: Breakpoint System presented in CAULDRON where the different steps in debugging a system can be visualized.

This functionality uses the same mechanism used in the previous one, where it is possible to exchange commands between editor-client and runtime, however, the breakpoint topic is used instead of the debug one. Then, in each node, a queue was implemented, where all incoming messages are stored when a node is in pause status. Moreover, for each message, a field was added with an arrival timestamp at a certain node.

Every time the step command is triggered, a message is extracted from the queue and, if the message provided by the Node-RED editor is different from this one, we know it was edited and so
it is sent. Furthermore, when the queue is empty, the editor is notified.

When a breakpoint is deleted, all messages queued on the respective node are released in the “same” frequency that they were received. More specifically, the time between the last two messages was calculated, and this value is used to set the pace. If the node receives more messages while the queue is being released, these new messages are still pushed to the queue. This can lead to messages always being queued. However, if the user did not want to preserve the old messages, it is possible to just clear the queue.

There were two main obstacles during implementation. First, there was the need to support different data types, specifically objects. All messages are converted to a string before sent, consequently, it was necessary to parse them correctly. The other difficulty was the fact that Node-RED overrides multiple HTML DOM events, i.e. the `keyDown` and `mouseDown` events that block the interaction with the HTML input’s element necessary to change the message on the step functionality. Therefore, these methods were modified by whitelisting our HTML class.

### 4.7 Enhancing the Debug Node

The `Debug` node was enhanced with the same message visualization capabilities of other nodes. However, the source of these messages is the `debug` topic, already used for communication between the `editor-client` and `runtime` sub-modules, unlike other nodes, where this source was the `monitoring` topic, which was used to show the messages in the other nodes. This was adopted because this node is capable of: (a) accessing a specific field in the message’s payload, (b) accessing the entire message, or (c) applying a JSONata expression, as shown in Figure 4.10. Additionally, it can also disable incoming messages by temporarily pausing the associated event.

Therefore, by using the messages received through this topic, `CAULDRON` can automatically change the plot and messages’ values according to the configured condition.

![Figure 4.10: The different ways to parse the input’s messages on Debug node.](image-url)

---

9JSONata, [https://jsonata.org/](https://jsonata.org/)
4.8 Summary

We modified Node-RED to augment the system’s observability and tighten the feedback loop between the development environment and system under development, in an effort to improve the users’ ability to build, evolve, and maintain IoT systems. Some of these features already exist in other areas, and they were used as an inspiration to introduce novelty to the system.

CAULDRON focuses on addressing some of the identified missing features in IoT: (1) Observability, by providing the ability to show the information flowing through the nodes using different visual metaphors, (2) Runtime Modification, by allowing the injection of messages during runtime, and (3) Exploration, by enhancing the debugging capabilities through breakpoints on each node without the need for re-deployments.

Node-RED is composed of six sub-modules and our development focuses on two: editor-client and runtime. They communicate with each other through HTTP Requests and WebSockets. These WebSockets communications were adapted by turning them bidirectional and by creating new topics used in the new features. The sub-modules already presented are very coupled between them and CAULDRON tried to be as less intrusive as possible by applying a similar design and internal event handling to Node-RED.

Leveraging the already existing communication mechanism (between the runtime and UI), a new topic was added that allows showing the runtime data (i.e., messages between nodes) in the UI. Using this additional communication channel, we can visualize incoming messages through two different plots: if the message’s payload type is a number, it displays a line plot; otherwise, a scatter plot is shown. This plot lets the user check how values change or if the node is receiving messages, and at what pace they are coming. By allowing this communication to be bidirectional, and applying the same strategy, we can inject messages at runtime to test specific system behavior.

We also added extra debugging capabilities in the form of breakpoints. The breakpoint system allows the user to “pause” the incoming messages of a given node by queuing them. Additionally, they can step forward a message at any time and change its payload. It is also possible to clear all the queued messages. When the node returns to its normal function, the messages queued are released in the “same” frequency that they were received (the time between the last two messages is used as a fixed pace). We further enhanced the Debug node with the same message visualization capabilities of other nodes.

With our approach, each node presents each input message, with the exception of nodes without any input, where the output is shown instead. This way, we can see all the information flowing through all nodes without needing to add new ones. We can also pause the flow through the use of breakpoints and resume it as we like. Finally, we can inject messages to check faulty behaviors.
Chapter 5
Validation

5.1 Overview

In the previous chapter, we described the development and features present in CAULDRON. All of these features were tested with the help of our SmartLab1, described in more detail in Figure 5.1 (p. 44), where we can validate the tool’s expected behavior against a physical setup. Furthermore, a sensor/actuator simulator was developed in order to automatically check the correctness of the tasks made by test subjects in the quasi-experiment. In addition, this simulator was used to test different scenarios with various types of messages and frequencies.

---
1SmartLab is an experimental testbed deployed in a software engineering laboratory at FEUP
Figure 5.1: SmartLab’s component diagram, showing the main parts of the system, along with the different devices and communication protocols. This system is composed of 4 actuators and 3 sensing devices (each with more than one sensor) [22].

Simulator

The device simulator was developed in Express and based on a template of our own\(^2\) that already provides a solid architecture, splitting code in controllers (business logic), services and routes, following good programming practices. It uses Mosquito\(^3\) as an MQTT broker and the solution is self-contained by using Docker\(^4\) and Docker Compose\(^5\). Docker allows the deployment and use of the simulator on any target machine, providing an isolated application with decoupled components.

The devices’ behavior is simulated by applying the publish-subscribe pattern. In this pattern, publishers are responsible for injecting messages in specific queues, while subscribers consume those messages. Any message published to a topic is immediately received by all of the subscribers to that topic [3]. In the simulator, sensors operate as publishers by injecting readings. On the other hand, actuators serve as subscribers by receiving commands and, thus, updating their state. This tool does not simulate the interaction between sensors and actuators, i.e., the state of the actuators is not influenced by the sensor’s readings, but instead by the commands received.

Aside from the capabilities of simulating devices, it is also possible to validate a given scenario. This mechanism is based on specific data that is sent through the sensors, causing a change of state in the actuators, which is checked to validate the scenario.

The main goal of the development of this simulator was to easily reproduce sensors and actuators in order to provide constant information in testing scenarios. This information is introduced through JSON files, leading to a generic simulator that creates all the logic needed at runtime. More information about these files and communications available is available in Appendix B (p. 63).

All the information used to develop this simulator was based on the data collected by our SmartLab providing a good approximation to a real scenario.

---

\(^2\)Express-template, https://github.com/diogotorres97/express-template

\(^3\)Eclipse Mosquito, https://mosquitto.org/

\(^4\)Docker, https://www.docker.com/

\(^5\)Docker Compose, https://docs.docker.com/compose/
5.2 Experimental Goals

Our goal is to verify if such an enhanced tool impacts the development process. We carried a controlled experiment to compare the performance and behavior of two developer groups, a strategy that has been suggested by some authors [34, 35, 57]. We hypothesized that a tool with the characteristics detailed in Section 3.3 (p. 27) would improve the ability of users to successfully build, evolve, and maintain IoT systems in a faster, easier, and less error-prone way.

This experiment is split into five parts: the first one, related to the design of the experiment in terms of participants, collected data, and other experimental settings; the second, explaining the tasks carried out; the third, concerning the results obtained; the fourth, where we revisit our research questions; and the last one, related to possible validation threats.

5.3 Experimental Parameters

We started by doing a preliminary assessment of our procedure with two participants having distinct backgrounds: (1) a casual Node-RED user and (2) a user with no previous experience in Node-RED. Then, we set out to adopt the following parameters for the full study, whose guide is available in Appendix C (p. 67):

**Experiments:** They consisted of (a) debugging, (b) improving, and (c) creating an IoT system using Node-RED; hence, development experience and basic familiarity with IoT were required.

**Participants:** The population size was twenty participants, all final-year computer science students with at least basic IoT knowledge, but with no Node-RED experience.

**Duration:** To avoid participants’ overload and, at the same time, provide a reasonable time to finish all of the tasks, the duration of the experiment was set to 90 minutes, with a 25 minute timeout per task.

**Procedure:** We made use of a mix of a quasi-experimental with ethnographic research. The population was split into two groups, GA and GB, with different treatments: GA used unmodified Node-RED, while GB used our tool. As there were no guarantees of equal technical knowledge among groups, two control tasks (CT) were performed to provide basic familiarity with the tool. Following those, three experimental tasks (ET) were given to each group: *viz.* (a) debug, (b) improve, and (c) create a system from scratch. In these three tasks, GB was provided with additional documentation regarding the available new features. All tasks were solved in the same order, with a small time break between them.

**Environment:** All experiments were conducted in a remote environment. The needed tools were hosted in a private virtual server, with more details given in Appendix D (p. 77). Video call

---

6Due to the COVID-19 pandemic.
software was used to communicate and provide access to the participants’ screen. With this procedure, it was possible to observe and take notes on their behavior, clarify task-related questions, and verify if a certain solution was correct.

**Data:** For both groups, we recorded: (a) the *time* taken to reach the solution; (b) the *number of deployments* made; and (c) the *number of verification requests* (i.e., every time the user thought the task was finished). For GB subjects, the *number of clicks* in each new functionality was also recorded.

**Post-test:** A questionnaire was carried to assess the participants’ overall experience, as well as to collect improvement suggestions. For this, we resorted to five statements evaluated using a Likert-scale: three related to existing functionalities in CAULDRON, and two regarding future improvements. Some questions had small differences depending on the group the subject belonged to. One version of this questionnaire is available in Appendix E (p. 79).

### 5.4 Tasks

To make it possible to run the experiments with equal operating conditions, the sensor/actuator simulator developed was used (having a deterministic behavior) to provide real-time data (continuous flow of messages). It was also used to validate the correctness of experimental outcomes. The Control Tasks are described below and depicted in Figure 5.2:

**CT1.** A preliminary task where Node-RED is introduced alongside the concept of creating a simple flow. It shows how to manually inject messages in a flow, parse them with custom JavaScript, and then display them in the sidebar.

**CT2.** A task where data from seismometers must be used to activate an alarm, depending on the earthquake’s magnitude. This task introduced new nodes and logic (i.e., read data from sensors, add intermediate logic, send commands to the actuators...) that served as a basis in further tasks.

![Figure 5.2: Node-RED flow representing a possible solution to the Control Tasks.](image-url)
The first two Experimental Tasks were both based on a *smart farming* scenario, where a system would automatically control a strawberry plantation inside a greenhouse. A third task focused on the development of a simple *smart home* system. A possible solution to the smart-farming tasks is shown in Figure 5.3a and to the smart-home system in Figure 5.3b. More specifically, these tasks consisted of:

**ET1.** A debugging task with a set of rules. The system was capable of keeping the soil at a certain moist and temperature level. For this, the user was able to control (a) a heating system, (b) an irrigation mechanism, and (c) automatic windows. These were controlled by a humidity/temperature sensor. These rules had some bugs related to (a) erroneous conditions, (b) wrong commands sent to the actuators, and (c) mismatched field accessors.

**ET2.** An improvement task, where the user is responsible for adding a new feature to the previous system, by using new devices (both sensors and actuators). More precisely, (a) the status of the UV lamps should be adjusted according to weather forecasts, and (b) if the UV lamps’ are on, the window should be closed.

**ET3.** A system design task, where the user must create a simple *smart home*. Two different types of rules were given: (a) the lights should turn on when there is movement in the kitchen, and (b) every day, at a given hour, the water heater and the coffee machine should be turned on (recurrent rule).

![Diagram](image-url)  
**Figure 5.3:** Node-RED flow representing a possible solution to the Experimental Tasks.
5.5 Analysis

We now provide an analysis of the results for both the Control and Experimental Tasks. We discarded CT1, as it was mostly used as a sanity check. For all statistical hypothesis tests, the significance level to compare the $\rho$-value was set as 0.05.

5.5.1 Control Task

We used CT2 to verify if there was a statistical difference between the two experimental groups, with 10 elements each, by measuring the time spent and number of deployments required, as presented in Table 5.1.

Table 5.1: Time spent and number of deployments in CT2.

<table>
<thead>
<tr>
<th>Grp</th>
<th>Mean</th>
<th>$\sigma$</th>
<th>Median</th>
<th>Shapiro-W. ($\rho$)</th>
<th>Levene ($\rho$)</th>
<th>t-test ($\rho$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>8:30</td>
<td>2:00</td>
<td>9:09</td>
<td>0.69</td>
<td>0.54</td>
<td>&gt; 0.99</td>
</tr>
<tr>
<td>B</td>
<td>8:30</td>
<td>2:15</td>
<td>8:54</td>
<td>0.61</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deploys</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>4.00</td>
<td>1.25</td>
<td>4.00</td>
<td>0.55</td>
<td>0.75</td>
<td>0.87</td>
</tr>
<tr>
<td>B</td>
<td>3.90</td>
<td>1.37</td>
<td>3.50</td>
<td>0.16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.4: Time (a) and number of deployments (b) in CT2.

We start with Levene’s test, verifying if both groups are from populations with equal variances. As the obtained $\rho$-value is 0.54 for time, and 0.75 for the number of deployments, we cannot reject the null hypothesis (i.e., both groups present equal variances). We then applied a Shapiro-Wilk test to verify if each of the groups were drawn from populations with a normal distribution. Since the resulting $\rho$-value is above the significance level (time: $\rho(GA) = 0.69$ and $\rho(GB) = 0.61$;
deployments: \( \rho(GA) = 0.55 \) and \( \rho(GB) = 0.16 \), we also fail to reject the null hypothesis (i.e., both groups present a normal distribution in the results). Ergo, we assume that both samples come from normally distributed populations with equal variances.

We then use Student’s \( t \)-test for assessing the following hypothesis related to time, viz. \( H_0 \): both groups needed a similar amount of time to complete the task, and \( H_1 \): there exists a significant difference in the average time for each group to complete the task. Concerning deployments, we assume \( H_0 \): both groups made a similar amount of deployments to complete the task, and \( H_1 \): there exists a significant difference in the average number of deployments made in each group to complete the task.

We observe that the time spent has a \( \rho \)-value = 0.997 and the number of deployments has a \( \rho \)-value = 0.866, not allowing us to reject \( H_0 \), and thus be forced to consider that there is no statistical difference between the two groups (cf. Figure 5.4, p. 48) as intended.

### 5.5.2 Experimental Tasks

Using the same hypotheses described in the Control Tasks, we present the results of the Experimental Tasks, together with a qualitative analysis.

#### 5.5.2.1 Time

Analyzing the time spent for the three tasks and the results from the \( t \)-test presented in Table 5.2, we were initially unable to reject the null hypothesis for all tasks.

<table>
<thead>
<tr>
<th>Task</th>
<th>Grp</th>
<th>Mean</th>
<th>( \sigma )</th>
<th>Median</th>
<th>( t )-test (( \rho ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>ET1</td>
<td>A</td>
<td>12:53</td>
<td>5:34</td>
<td>12:17</td>
<td>0.75</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>12:08</td>
<td>4:33</td>
<td>11:36</td>
<td></td>
</tr>
<tr>
<td>ET2</td>
<td>A</td>
<td>8:13</td>
<td>2:10</td>
<td>8:34</td>
<td>0.30 (0.03*)</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>6:57</td>
<td>3:05</td>
<td>5:47</td>
<td></td>
</tr>
<tr>
<td>ET3</td>
<td>A</td>
<td>8:34</td>
<td>2:32</td>
<td>8:12</td>
<td>0.47</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>7:49</td>
<td>1:59</td>
<td>8:05</td>
<td></td>
</tr>
</tbody>
</table>

We started by concluding that there are no relevant differences between the two groups, as depicted in Figure 5.5 (p. 50). However, a Grubb’s test for outliers singled out one in ET2, forcing us to discard it (cf. Figure 5.5b, p. 50), resulting in a \( \rho^* \)-value of 0.03. This allows us to conclude that the experimental group does present a statistical difference when adding new features to an existing system concerning time. Regarding the other tasks, we believe that they might have not captured a sufficient degree of difficulty/complexity to evidence substantial differences, or the sample size was insufficient. We do, however, consistently observe a lower mean and median for all tasks in the experimental group. More specifically, in terms of mean time per task, the experimental
Validation

Figure 5.5: Time spent in ET1–3.

group showed a reduction in ET1 time by 45 seconds (5.8% faster), in ET2 by 1 minute and 16 seconds (15.4% faster) and in ET3 by 45 seconds (8.7% faster).

5.5.2.2 Deployments

All experimental tasks present $\rho$-values lower than the significance level (0.05), as observed in Table 5.3. This allows us to reject the null hypothesis and accept the alternative, i.e., there is a significant difference in the average number of deployments made between the groups, with the experimental group performing fewer, as evidenced in Figure 5.6 (p. 51).

Table 5.3: Number of deployments in ET1–3.

<table>
<thead>
<tr>
<th>Task</th>
<th>Grp</th>
<th>Mean</th>
<th>$\sigma$</th>
<th>Median</th>
<th>$t$-test ($\rho$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ET1</td>
<td>A</td>
<td>7.90</td>
<td>3.60</td>
<td>7.50</td>
<td>$&lt; 0.01$</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>3.00</td>
<td>1.05</td>
<td>3.00</td>
<td></td>
</tr>
<tr>
<td>ET2</td>
<td>A</td>
<td>4.30</td>
<td>2.11</td>
<td>4.50</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2.10</td>
<td>1.29</td>
<td>2.00</td>
<td></td>
</tr>
<tr>
<td>ET3</td>
<td>A</td>
<td>4.50</td>
<td>2.07</td>
<td>4.00</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>2.70</td>
<td>1.49</td>
<td>2.50</td>
<td></td>
</tr>
</tbody>
</table>

Comparing the mean and median of the number of deployments to reach the solution presented in Table 5.3, there is a clear tendency for the experimental group to need fewer deployments — nearly half compared to the control group. Focusing on the mean value per task, the experimental group showed a reduction in ET1 by 4.90 deployments (62% less), in ET2 by 2.20 deployments (51% less) and in ET3 by 1.80 deployments (40% less). This aligns with our initial hypothesis since every time the user needs to add new debug nodes in the control group, they are forced to
5.5 Analysis

Figure 5.6: Number of deployments in ET1–3.

deploy. On the other hand, the experimental group was presented with real-time feedback, thus decreasing such need.

5.5.2.3 Verification Requests

A verification request occurred every time a participant regarded their task as completed. The statistical analysis allows us to reject the null hypothesis on both ET2 and ET3, as presented in Table 5.4.

Table 5.4: Number of verification requests in ET1–3.

<table>
<thead>
<tr>
<th>Task</th>
<th>Grp</th>
<th>Mean</th>
<th>σ</th>
<th>Median</th>
<th>t-test (ρ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ET1</td>
<td>A</td>
<td>1.50</td>
<td>0.53</td>
<td>1.50</td>
<td>0.33</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>1.80</td>
<td>0.79</td>
<td>2.00</td>
<td></td>
</tr>
<tr>
<td>ET2</td>
<td>A</td>
<td>1.50</td>
<td>0.53</td>
<td>1.50</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>1.10</td>
<td>0.32</td>
<td>1.00</td>
<td></td>
</tr>
<tr>
<td>ET3</td>
<td>A</td>
<td>1.80</td>
<td>0.92</td>
<td>1.50</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>B</td>
<td>1.10</td>
<td>0.32</td>
<td>1.00</td>
<td></td>
</tr>
</tbody>
</table>

We thus conclude that there is a significant difference between the two groups regarding the construction and evolution tasks concerning the subjective perception of when a task was done, with the experimental group being more often correct (lower attempts) than the control group. Looking at the mean of verification requests per task, the experimental group showed an increase in ET1 of 0.30 (20% more) and a decrease in both ET2 and ET3 of 0.40 (26.7% less) and 0.70 (38.8% less), respectively.
5.5.2.4 Behavior

We observed that the experimental group, especially during ET1, changed their debugging strategy by focusing on visualizing and understanding the messages in the system instead of attempting to understand the underlying logic of each node. This was one of the most interesting observed phenomena because it represents a change in the participants’ behavior when approaching their tasks. This finding merits further study before any major conclusions can be drawn.

5.5.2.5 Experimental Group Feature Usage Analysis

After aggregating the results for each task (cf. Figure 5.7), we conclude that the most used features in CAULDRON were those related to the visualization of the messages, i.e., (1) plot, (2) detailed message, and (3) history.

![Figure 5.7: Clicks on CAULDRON functionalities.](image)

In terms of usage by task, presented in Table 5.5, higher values are seen on the mean and median for ET1, following by ET3 and then ET2. These results are according to our expectations since on ET1 participants spent more time in understanding the system, and consequently the messages that flow through it. In ET2, the extra features were not used as much because the participants already understood the system and did not feel the need for a deeper exploration. ET3 was focused on constructing a new system, which results in the observed higher values as they attempted to understand the messages’ flow.

Table 5.5: Total clicks aggregated by ET1–3.

<table>
<thead>
<tr>
<th>Task</th>
<th>Mean</th>
<th>σ</th>
<th>Median</th>
<th>Min.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ET1</td>
<td>54.60</td>
<td>34.36</td>
<td>43.00</td>
<td>21</td>
<td>130</td>
</tr>
<tr>
<td>ET2</td>
<td>17.50</td>
<td>12.64</td>
<td>12.50</td>
<td>1</td>
<td>35</td>
</tr>
<tr>
<td>ET3</td>
<td>23.10</td>
<td>15.38</td>
<td>21.00</td>
<td>4</td>
<td>61</td>
</tr>
</tbody>
</table>
5.6 Discussion

5.5.3 Post-Test Questionnaire

To evaluate the participants’ experience, we performed a post-test questionnaire composed of six questions, one about the general satisfaction (S1), and five concerning each one of the functionalities (Q1–Q5). More specifically, (Q1) is related to showing the input messages on each node, (Q2) to the plot that shows the messages, (Q3) to the breakpoint system, (Q4) the possibility of having typed connections between nodes, and (Q5) the prospect of highlighting the node path of a message. Although only the experimental group (GB) used some of the new features, we also asked the control group (GA) if they would like to have had such features. Interestingly, we have found a very close match between the two groups (cf. Figure 5.8).

The highest divergence was found in Q4 and Q5, which referred unavailable features on both groups (i.e., not implemented in CAULDRON). This can be explained considering that the experimental group was exposed to the experience of having real-time feedback during development, and not feeling the need of these extra features. In Q3, results were similar, since in our tool most participants ended up not using breakpoints. Overall, we conclude that most participants seem to want the functionalities described in each question. Finally, the results of S1 suggest that the experimental group had a more enjoyable experiment.

5.6 Discussion

Taking into account the experimental results presented in the previous section (5.5), we now revisit our research questions:

**RQ1. Would users with increased exposure to real-time information about the running system build and manage it faster?** Both groups spend a similar amount of time in solving the tasks, with a statistical significant difference observed on improving systems. We also noted that the experimental group presented consistently smaller mean and median values, that corresponds to a decrease of 55 seconds (i.e., about 10% faster) comparing to the control group.

**RQ2. Does providing users with real-time feedback increase their ability to understand and change existing systems?** According to the number of deployments performed per task, together with qualitative analysis, we can conclude that in a system with higher feedback,
users tend to perform less attempts of deployment (a difference of approximately 2.96, a decrease of 51%), thus pointing that these features make the system easier to change.

**RQ3.** Is an IoT visual programming environment, able to reduce human-induced errors during development by providing real-time feedback? By analyzing the number of deployments and attempts, we see a substantial difference with users in the experimental group having 51% less need to deploy and more confidence in their solution (i.e., they required 15% less attempts to achieve a successful task completion). This can be specially useful in more critical systems, where deployments should be kept to a minimum.

Overall, we are confident that, within some validation threats, a development environment with real-time feedback and improved debugging capabilities improves the ability to build, maintain and improve IoT systems.

### 5.7 Threats to Validity

A controlled experiment seems to be one of the most appropriate techniques for testing our hypothesis. However, it is not bulletproof, and validation threats can appear [71]. These threats are related with the question of how the conclusions might be wrong and we identified and categorized them in internal and external validity [71, 72, 15].

Internal validity concerns issues that may indicate a causal relationship between the treatment introduced (the cause) and the outcome (the effect). This means that these issues can make the experiment show a behavior that is not caused by the treatment, but a disturbing factor. External validity concerns the ability to generalize results for a broader scope than the one of the experimental setting. This means that the results could not be applied in other contexts due to the effect of the experiment design/parameters chosen.

**Internal Validity**

**Overhead of necessary tools to perform the tasks:** Because the experiment was timed and has a complex setup, it is necessary to ensure that participants focus on the tasks at hand. To discard this threat, the necessary setup in a private server was conducted before the experiments.

**Environment:** All experiments were conducted in a remote environment. Aside from different conditions between the participants, the results provided by Control Tasks allow us to conclude that this threat does not influence the results.

**Unsuitable skills:** The tasks required participants to have the development experience and basic familiarity with IoT. This threat is discarded by selecting final-year computer science students with at least basic IoT knowledge, but with no Node-RED experience. To mitigate this last point two control tasks were given to both groups.
5.8 Summary

Misunderstanding the guide: Because the tasks relied on specifications given in the textual form, it is necessary to ensure that the participants correctly interpret them. This threat is mitigated by allowing the participants to clarify doubts about the tasks.

External Validity

Problem Set: The problems might have not captured a sufficient degree of difficulty/complexity to evidence substantial differences. However, considering the time collected for each task, an increase in the complexity of these tasks could lead to an excessive overburden of the participants and thus decreased motivation.

Sample Size: The sample size was insufficient. We observe an outlier and believe that the population may have different skills. However, the two control tasks deduce that there is no significant statistical difference between groups.

Sample Representation: The sample does not contain users with different levels of experience. A richer population (i.e., with experient Node-RED users) will validate our hypothesis with other end-users.

We are aware of these threats and we mitigate them as possible. Thus, we can conclude that our hypothesis remains valid.

5.8 Summary

CAULDRON features were tested against a physical setup and a sensor/actuator simulator that allowed us to check different scenarios and settings. This simulator was developed due to the need of automating the verification of the tasks’ correctness in the controlled experiment.

A quasi-experiment was carried out to compare the performance and behavior between the original Node-RED and one with our developed extensions, since our hypothesis stands for the impact of an enhanced tool in the ability of the users in creating, maintaining and improving their IoT systems.

We tested our hypothesis against 20 final-year computer science students with basic IoT knowledge, but with no Node-RED experience. The population was split into two groups with different treatments and documentation was provided according to each group. Two control tasks were performed to verify if both groups are similar. Following this, three experimental tasks focused on (a) debugging, (b) improving, and (c) creating an IoT system were supervised. The first two tasks were related to a smart farming scenario and the other a simple smart home system. All experiments were conducted in a remote environment and, for both groups, we recorded: (a) the time taken to reach the solution; (b) the number of deployments made and (c) the number of verification requests. We also recorded information related to feature usage analysis in CAULDRON and a post-experiment questionnaire took place to assess the overall participant experience.
We ran statistical tests to ensure that there is no difference between the two groups through control tasks. Then, we evaluated the metrics described above using a t-test and other descriptive statistics (e.g., mean, median). Through the time variable, we concluded that there are no statistical differences between the groups. However, in the improvement task, we found one outlier that contradicts this statement. Furthermore, in all the tasks, the group with our tool presented a smaller mean and median when compared to the other group, being faster by about 10%. In terms of deployments, the group that used our tool seemed to be benefited, nearly halving the number of deployments. We also observed a behavioral difference during the debugging task where the group with CAULDRON focused on understanding the messages instead of attempting to understand the underlying logic of each node. Finally, we saw that the most used features in our tool were those related to the visualization of messages.

We identified some validation threats that were handled accordingly and concluded that a development environment with real-time feedback and improved debug capabilities increases the users’ ability to understand and change an IoT system and also reduce human-induced errors during development.
Chapter 6

Conclusions

6.1 Summary

We studied the Internet of Things and the current state-of-the-art for monitoring applied to visual programming tools.

The appearance of the Internet of Things as the peak of ubiquitous computing and connectivity is the result of having Internet-connected devices everywhere capable of sensing and actuating in the real world. This new vision, where devices and humans interact with each other, enables a better quality of life and resource management.

Developing IoT systems is challenging due to a set of particular characteristics of IoT systems, including heterogeneity, dynamic topologies, and highly-distributed nature. To abstract some of these characteristics, Visual Programming Languages have been used. One of the most widespread visual programming solutions tailored for IoT is Node-RED. This tool allows the development of these systems through a drag and drop interface, connecting nodes by wires. However, Node-RED is no “silver-bullet” and has several issues and gaps when compared with the development ecosystem of other tools. We observed how Node-RED does not provide feedback during development or at runtime and lacks on debugging capabilities. During development, there is no information about the “correctness” of connections between nodes. At runtime, it is impossible to observe the messages...
that flow through the system. Furthermore, there is no way to change or inject messages in the system to evaluate some behavior. Finally, every time that the user changes the flow, it must be redeployed.

However, we found that these problems have been at least partially solved in other areas, such as video/image rendering and game development. The tools specific for those areas already present the current state of the system and allow to change some settings in the nodes they present at runtime. Moreover, it is also possible, with breakpoints, to pause the system in a given node to understand with finer granularity.

6.2 Hypothesis Revisited

Considering the Chapter 2 (p. 5), we were able to identify the issues that needed to be addressed and stated our main hypothesis as:

"Enhancing the feedback loop between system and development environment improves the ability of users to successfully build, evolve and maintain their IoT system."

Our hypothesis is not focused on enhancing Node-RED but to explore if this kind of proposed environment towards a more “live” setting does improve the ability of users to successfully build, evolve and maintain their systems, thus improving the observability, testability, sanity and safety-checking of the overall system. To achieve that, and based on existing features in other domains, we modified Node-RED (cf. Chapter 4, p. 31) to present: (a) the runtime state of the system (i.e., observing input/output of each node through different visual metaphors), (b) debugging mechanisms (i.e., breakpoints), (c) runtime modifications (i.e., injecting and changing messages) and (d) an overall increase of immediate feedback. These modifications were as less intrusive as possible by adopting a similar design to Node-RED, handling some of their internal events and extending their WebSockets communication mechanism.

To validate this approach (cf. Chapter 5, p. 43), our solution was tested against a physical setup. We also developed a sensor/actuator simulator to ease the process of validation on a conducted user study, as well as extensively testing the new features by simulating different scenarios. In the controlled experiment, we chose users interested in IoT systems and split them into 2 groups. One group used the original Node-RED, while the other had access to CAULDRON’s functionalities. Each completed the same set of tasks, hence, with this procedure, we were able to evaluate their speed, trial-error rate and behavior to verify if the modified system impacts development.

We revisited our research questions and concluded that users in a system with such functionalities had no significant differences in time spent in tasks related to (a) debugging, (b) improving and (c) creating an IoT system, despite always presenting a decreased time in 10%, in the experimental group. Furthermore, the number of deployments needed in such environment is nearly half compared to the control group, reflecting the usefulness of real-time feedback provided instead of having to rely on adding debug nodes and, consequently, redeploying the system. Users with more feedback about the system have more confidence in their solutions, resulting in 15% less attempts
to successfully solve the tasks comparing to the control group. Finally, the participants’ subjects with CAULDRON react differently compared to the control group in the task of debugging, where they analyze the messages flowing through the system as opposed to understanding the underlying logic in each node.

6.3 Main Contributions

As a result of this dissertation, our main contributions are:

**State-of-the-art:** We studied IoT and how monitoring has been applied in this area, as well as in video/image rendering and game development. We finished it by presenting some of the issues and gaps present in development environment solutions to IoT, specifically Node-RED.

**CAULDRON:** Our solution is a modified version of Node-RED with increased feedback of the system, without the need of extra nodes and re-deployments, and enhanced debugging capabilities through the use of breakpoints.

**Sensor/actuator simulator:** A tool to provide real-time data (continuous flow of messages) that implements mechanisms to validate the correctness of experimental scenarios.

**Quasi-experiment with an ethnographic research:** A group of users completed a set of tasks in the two different Node-RED versions (original and CAULDRON), for which they need to debug, evolve or build an IoT system, concluding that, with such enhancements, the number of failed attempts to deploy the system is reduced, as well as a slight reduction in the development time.

6.4 Future Work

Taking into account the recorded suggestions from our experiment’s participants, as well as the remaining issues presented in Chapter 3.1 (p. 25), we present the following possible paths:

**User Interface/Experience:** The user interface is one of the main shortcomings of the modified Node-RED. Despite the author’s attempt to respect the tool’s interface guidelines, a more thorough approach to UX / UI would increase interaction with the new features. Messages history would be one natural area for improvement.

**Configuration of the extra functionalities:** The new functionalities could be enabled/disabled through the presence of a configuration node. Aside from that, some settings could be parameterized: (a) personalized messages plot (e.g., by changing its color, type or x-axis size), (b) the queue size and its behavior in the breakpoint system (e.g., when it reaches the maximum size, the system could discard the more recent or older messages) and (c) the behavior when a breakpoint is removed (e.g., sending all the queued messages at once, with the same frequency as received or discard all the received messages).
**Typed connections**: Through verification of the correctness of connections between nodes, during development, it would be possible to detect some potential errors that are only reported after deployment (e.g., a certain node expects to receive messages of string type and receives a number instead).

Despite our focus being on the enhancement of feedback in IoT development, we also present enhancements to our sensor/actuator simulator:

**State machine**: By implementing a more complex state machine, this tool will be able to test more complex scenarios with multiple states and transitions (e.g., testing multiple branches of an “if” condition in a single test).

**Data generation**: Data generated, in a predefined domain, at runtime would be useful to test random scenarios or more thoroughly test a system.

**Actuators influence sensors**: The sensors’ data should change the actuators’ behavior. This will be beneficial in cases where a given actuator turning on influences a specific sensor, e.g., a temperature sensor changing its readings due to the presence of a heating system.

Furthermore, we also intend to improve the validation protocol of the proposed solution where we would like to:

**Increase the tool testing**: By testing it against different physical setups.

**Improve the user study**: By repeating the experiment with a larger population. These participants should have different qualifications e.g., have experienced Node-RED users as well.

Finally, we consider that there are still several points of improvement that were not addressed during this dissertation and can be taken as future research directions.
Appendix A

Example of Data Stored in *editor-client*

This appendix includes a sample of the data stored in the Node-RED front-end component.

```json
1 {
  "764400a.19503": {
"input": [

  {
    "x": "2020-05-10T10:31:10.405Z",
    "y": 25,
    "type": "number",
    "id": "7b980248.12881c"
  },

  {
    "x": "2020-05-10T10:35:13.452Z",
    "y": 19,
    "type": "number",
    "id": "e5028b7a.3e3488"
  },

  ...}

"output": [

  {
    "x": "2020-05-10T10:35:13.454Z",
    "y": 19,
    "type": "number",
    "id": "e5028b7a.3e3488"
  },

  ...}

}]
```
Listing A.1: An example of how data is stored in the `editor-client` module. The first object with ID "764400a.19503" represents the Switch Node presented in the flow in Figure 4.2. The input field corresponds to the incoming messages from the temperature sensor. The output field contains two arrays, the first one related to the first node’s output where messages are lower than 22°C and the other related to the second output where messages are higher than the same temperature. Moreover, it presents the settings with the default messages shown (`defaultMessages` and `portNumber` fields) and if the breakpoint is activated (`pauseDebug` field).
Appendix B

Sensor/Actuator Simulator

This appendix includes a sample of the input files in the simulator and the routes available.

Listing B.1: An example of a sensor’s data file in the simulator. This file is composed of an array of messages. For each message, the payload is published in a queue with the topic associated. This topic name depends on the scenario name that follows the format :scenarioName:topic.
Listing B.2: An example of an actuators file in the simulator. This file is composed of an array of actuators. For each actuator, the `defaultValue` is the initial state of the actuator every time that the scenario is (re)started. The topic represents the name of the queue where the actuator will receive commands. This topic name depends on the scenario name that follows the format `:scenarioName/:topic/command`.

Listing B.3: An example of a validation file in the simulator. This file is composed of an array of actuators. For each actuator, the `topic` is equivalent to its name. Then, an `initialState` and a `finalState` are provided to check if, given a certain sensor’s data, the actuators achieve that state.

Each of these files is inside a folder related to its type (i.e., sensors-data, actuators, validation) that allows multiple files of each type to a given scenario. Since the simulator supports multiple scenarios, these folders are inside a folder with the given scenario name.
Table B.1: Sensor/Actuator Simulator - Routes available

<table>
<thead>
<tr>
<th>URL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/:scenario/load</td>
<td>Loads a given scenario</td>
</tr>
<tr>
<td>/:scenario/start/:messageFrequency?</td>
<td>Starts a given scenario with a given message frequency in milliseconds (optional)</td>
</tr>
<tr>
<td>/:scenario/reset</td>
<td>Resets a given scenario where the queues are emptied and the actuators’ state restored</td>
</tr>
<tr>
<td>/:scenario/validate/:messageFrequency?</td>
<td>Starts a given scenario where the actuators are initialized with an initial state and after all the messages are sent (with a 5 seconds wait) the final state is checked</td>
</tr>
<tr>
<td>/:scenario/:actuator/status</td>
<td>Retrieves the actual state of an actuator to a given scenario</td>
</tr>
<tr>
<td>/:scenario/sensorsData</td>
<td>Retrieves all the messages loaded to a given scenario</td>
</tr>
<tr>
<td>/:scenario/actuators</td>
<td>Retrieves all the actuators available to a given scenario</td>
</tr>
</tbody>
</table>

Note: There are other endpoints in the API, however, only the more relevant ones are described.
Appendix C

Experimental Guide

This appendix includes the experimental guide corresponding to the group with CAULDRON. The difference to the other group is the absence of the documentation relative to functionalities developed.
Increase the feedback in IoT development in Node-RED, a Quasi-Experiment

Node-RED is one of the most widespread VPLs targeting IoT systems that mashups hardware devices, APIs, and third-party services, in a hybrid text-visual programming approach. With Node-RED, it is possible to create flows with the aim of composing some rules that the system has to comply with. These flows can be something like turning on the coffee-machine whenever the user wakes up, turning on the heating system when the user is returning home after work, turning on the A/C if the temperature rises above a certain level, and many other examples.

So, let’s begin with a tutorial in order to become familiar with the tool that will be used in the next set of tasks.

Tutorial

Task 1

This task is similar to a “Hello World” where Node-RED is introduced alongside some basic concepts by creating a flow that demonstrates the Inject, Debug and Function nodes.

1. Go to http://darkmagic.ddns.net:3000/nodered where the tool is running.
2. Go to https://nodered.org/docs/tutorials/first-flow and proceed with the tutorial skipping the first step.

Task 2

This task will introduce a flow based on data from an earthquake sensor to do something useful. Also, it will introduce four new nodes (MQTT in, MQTT out, Switch and Change) that will be useful in further experiments.

1. Add an MQTT IN node, click on it to see further information and edit as follows:

More information about MQTT.
2. Add a Debug node to the output. Deploy the system and observe the data in the Debug Sidebar. You should see a list of entries with some contents that look like:

```json
msg.payload : Object
```

3. Now click on the little arrow to the left of each property to expand them and examine the contents.

4. Add a Change node, wired to the output of the MQTT IN node. Configure it to set `msg.payload` to `msg.payload.mag`.

5. Change the Debug node created in step 2 by setting `msg.payload` to `msg.payload.mag` and enabling node status (32 characters).

6. Add a Switch node to the workspace. Edit its properties and configure it to check the property `msg.payload` with a test of `>= change` it to test on a number and the value 7. Click Done to close and add a wire from the Change node to this Switch node.

7. Add a Change node, wired to the output of the Switch node. Configure it to set `msg.payload` to the string `ON`.

8. Wire a new Debug node to the output of the Change node.

9. Wire a new MQTT out node to the output of the Change node with the following configuration:

10. Deploy the flow by clicking the Deploy button.

In the Debug Sidebar, if there were any quakes with a magnitude greater than 7 you will also see debug messages like:

```json
msg.payload : string(2)
"ON"
```

You could change the switch value of 7 to a smaller one to test your program. Remember to click on deploy after the change. **NOTE:** This tutorial is based on the Second-Flow tutorial from Node-RED.
This section describes the extra functionalities provided by the **CAULDRON** tool. The messages shown in the node by default are the input messages. However, in nodes without any input, the output is shown. Also, these nodes without inputs are not able to inject messages or use breakpoint functionalities.

![Diagram of CAULDRON tool](image)

1) Debug Button - toggle the debug functionalities.

2) Messages Plot - has two different modes, when a payload is a number it displays a line plot, as visible in figure 1.2, otherwise displays a scatter plot with message frequency, as shown in figure 1.3.

![Figure 1.2](image) ![Figure 1.3](image)

3) Show More Button - toggle all information above the button.

4) Messages Buttons:
   a) Trash Button - Clear all the current saved messages
   b) Expand Button - Show in a popup all the messages according to the input/output, as shown in figure 1.4:
c) Inject Message - Allow inject a message for debug purposes, as shown in Figure 1.5.

![Node: 90fee01d.75132](image)

**Inject Message**

Payload: **23**

Figure 1.5

5) Current Message:
   a) Time - Show the arrival time of a message
   b) Value - Show the payload of a message, however, if this payload is an object it only displays [Object object]

6) Breakpoint Buttons:
   a) Trash Button - Clear queued messages when the breakpoint is activated
   b) Play/Pause Button - Allow stopping message flow in the current node, queuing them
   c) Step Button - Allow to send a message at a time and also editing them
Problem 1

John Doe has developed a system capable of automating the treatment of a strawberry plantation inside a greenhouse. However, strawberries require some special care such as:

- Exposure to sunlight;
- The soil pH must be contained between 5.5 to 7;
- Absence of wind;
- Temperature between 20-23 °C;
- Humidity above 70%.

Currently, the system is capable of:

- Keeping the soil moisture close to 75%. Through a humidity-temperature sensor that communicates its values periodically, if the value is lower than expected, the irrigation system is switched ON, otherwise OFF;
- Maintaining the temperature through the heating system. This system is switched ON if the temperature is below 20 °C until it reaches 23°C. Also, when it's ON, it automatically closes the roof window.

Note: You can access the status of the actuators in: http://darkmagic.ddns.net:3000/simulator/agriculture/actuators. Also, In each task, there are tables that present information about the actuators and sensors that can be used or are already used in the system.

Task 1

John Doe has had huge headaches with the heating system that has not worked properly. Also, the water system isn't working at all.

a) Open http://darkmagic.ddns.net:3000/nodered-feedback
b) Identify and fix bugs so that the system works as described.

Actuators

<table>
<thead>
<tr>
<th>Name</th>
<th>Topic</th>
<th>Command to send</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roof Window</td>
<td>agriculture/roof-window/command</td>
<td>&quot;OPEN&quot;/&quot;CLOSE&quot;</td>
</tr>
<tr>
<td>Irrigation System</td>
<td>agriculture/water-system/command</td>
<td>&quot;ON&quot;/&quot;OFF&quot;</td>
</tr>
<tr>
<td>Heating System</td>
<td>agriculture/heat-system/command</td>
<td>&quot;ON&quot;/&quot;OFF&quot;</td>
</tr>
</tbody>
</table>

Sensors

<table>
<thead>
<tr>
<th>Name</th>
<th>Topic</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>Humidity-Temperature Sensor</td>
<td>agriculture/temp-hum-readings</td>
<td>{ &quot;node-id&quot;:&quot;sensor-node-3&quot;, &quot;sensor&quot;: &quot;dht11&quot;, &quot;hum-percent&quot;: 60, &quot;temp-C&quot;: 18, &quot;timestamp&quot;: 1585673989 }</td>
</tr>
</tbody>
</table>
Task 2

Strawberries need to have sunlight daily. Implement a feature that, based on weather forecasts, turns OFF the UV lamps when it is a sunny day (you should use the icon property from forecasting data with the value clear-day). Otherwise, turn ON the UV lamps and close the roof window.

Actuators

<table>
<thead>
<tr>
<th>Name</th>
<th>Topic</th>
<th>Command to send</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roof Window</td>
<td>agriculture/roof-window/command</td>
<td>“OPEN”/”CLOSE”</td>
</tr>
<tr>
<td>Roof UV Lamps Controller</td>
<td>agriculture/roof-lux-controller/command</td>
<td>“ON”/”OFF”</td>
</tr>
</tbody>
</table>

Sensors

<table>
<thead>
<tr>
<th>Name</th>
<th>Topic</th>
<th>Payload</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forecasting</td>
<td>agriculture/forecast</td>
<td>{&quot;latitude&quot;: 41.1611, &quot;longitude&quot;: -8.614, &quot;timezone&quot;: &quot;Europe/Lisbon&quot;, &quot;time&quot;: 1586554471, &quot;icon&quot;: &quot;clear-day&quot;, &quot;precipIntensity&quot;: 0, &quot;temperature&quot;: 18.1, &quot;dewPoint&quot;: 12.13, &quot;humidity&quot;: 0.60, &quot;pressure&quot;: 1022.4, &quot;windSpeed&quot;: 1.32, &quot;ozone&quot;: 306.3}</td>
<td>Icon property will have one of the following values: clear-day, clear-night, rain, snow, sleet, wind, fog, cloudy, partly-cloudy-day, or partly-cloudy-night.</td>
</tr>
</tbody>
</table>
Problem 2

Now implement a system for a smart-home with some basic features:

- At any time, if there is movement in the kitchen, turn ON the lights.
- Every day at 6am, turn ON the water heater and the coffee machine.

Tip: Inject Node - It must be used to automatically trigger flows at regular intervals.

Note: You can access the status of the actuators in: http://darkmagic.ddns.net:3000/simulator/smart-home/actuators. Also, in the tables below, there are information about the actuators and sensors that can be used.

### Actuators

<table>
<thead>
<tr>
<th>Name</th>
<th>Topic</th>
<th>Command to send</th>
</tr>
</thead>
<tbody>
<tr>
<td>Water Heater Controller</td>
<td>smart-home/water-heat-controller/command</td>
<td>“ON”/“OFF”</td>
</tr>
<tr>
<td>Kitchen Lights</td>
<td>smart-home/kitchen-lights/command</td>
<td>“ON”/“OFF”</td>
</tr>
<tr>
<td>Coffee Machine</td>
<td>smart-home/coffee-machine/command</td>
<td>“ON”/“OFF”</td>
</tr>
</tbody>
</table>

### Sensors

<table>
<thead>
<tr>
<th>Name</th>
<th>Topic</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kitchen Motion Sensor</td>
<td>smart-home/kitchen-motion-readings</td>
<td>{ &quot;node-id&quot;:&quot;sensor-node-3&quot;, &quot;sensor&quot;: &quot;pir-motion&quot;, &quot;motion-bool&quot;: 0, &quot;timestamp&quot;: 1585673980 }</td>
</tr>
</tbody>
</table>

Note: motion-bool property will have one of the following values: 1 if there is movement, 0 otherwise.
FAQ

Q: How to not overload a certain device? By overload, it means receiving messages at a higher rate.
A1: Using a Delay node configured to rate-limit the messages passing through it (Slow down messages passing through a flow).
A2: Using the same node but configured to rate-limit the messages passing through it with the option to drop intermediate messages enabled (Handle messages at a regular rate).

Q: How to not overload the device by constantly sending the same command?
A: Using a RBE node configured to send a message only when the value changed (Drop messages that have not changed value).

Q: How to not incessantly turn on/off a device when receiving slightly differences in data readings?
A1: Using a RBE node configured to only send a message if respect a certain threshold (e.g., check if our input data has changed by more than 20%).

Q: What can I do to automate my home?
A: 25 Home Automation Ideas: Ultimate Smart Home Tour!
Appendix D

Validation Environment

This appendix includes the Docker Compose configuration file concerning the setup configured in the private server for the user study. Node-RED and the sensor/actuator simulator were slightly adapted to retrieve some statistical metrics (i.e., number of clicks in CAULDRON features).

```yaml
version: "3"

services:
  nginx:
    image: nginx:alpine
    restart: on-failure
    container_name: nginx
    volumes:
      - ./configs/.htpasswd:/etc/nginx/.htpasswd
      - ./configs/nginx.conf:/etc/nginx/conf.d/default.conf:ro
    ports:
      - 80:80
    depends_on:
      - nodered
      - nodered-cauldron
      - validation-web

  validation-web:
    image: nginx:alpine
    restart: on-failure
    container_name: validation-web
    volumes:
      - ./validation-web:/usr/share/nginx/html

  nodered:
    image: nodered/node-red:1.0.5
    container_name: nodered
    restart: on-failure

  nodered-cauldron:
```

77
Listing D.1: The stack needed for the user study is presented in this file. A nginx container is used as a reverse proxy to the other containers, abstracting the existence of other internal services. A validation-web container hosts a simple web page to facilitate the verification requests process (i.e., check the correctness of the tasks). The original Node-RED and our development solution are also provided. Finally, there are services for the sensor/actuator simulator, alongside a Mosquitto broker.
Appendix E

Validation Questionnaire

This appendix includes the validation questionnaire corresponding to the group with CAULDRON. The difference to the other group is in the first three questions where we asked if they would like to have had such features.
Node-RED - Post Experience

Some feedback about the experience made.

*Obrigatório

How satisfied are you with this experience? *

1 2 3 4 5

Very Unsatisfied  □ □ □ □ □  Very Satisfied

Suggestions for this experience

A sua resposta

Suggestions for the tool used

A sua resposta

Seguinte
Node-RED - Post Experience

*Obrigatório*

The input's message in each node improved the users ability to understand and develop these systems. *

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly Disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The message frequency (chart) in each node helped to perceive if a given node is receiving messages and at what pace. *

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly Disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The breakpoints, in each node, help to debug and understand the system. *

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strongly Disagree</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Typed connections between nodes would have been useful in the development of these systems (e.g., giving indication to the user that the destination node expects a number instead of a string). *

Highlighting the node path of a message would help to debug these systems (e.g., the orange wire in the image below). *
References


[57] Adrian Santos, Markku Oivo, and Natalia Juristo. Moving beyond the mean: Analyzing variance in software engineering experiments. In Marco Kuhrmann, Kurt Schneider, Dietmar Pfahl, Sousuke Amasaki, Marcus Ciolkowski, Regina Hebig, Paolo Tell, Jil Klünder, and
REFERENCES


