

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Orchestration for Automatic Decentralization in Visually-defined IoT

Ana Margarida Oliveira Pinheiro da Silva

WORKING VERSION



Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Hugo Sereno Ferreira

Second Supervisor: André Restivo

July 1, 2020

Orchestration for Automatic Decentralization in Visually-defined IoT

Ana Margarida Oliveira Pinheiro da Silva

Mestrado Integrado em Engenharia Informática e Computação

July 1, 2020

Abstract

The Internet-of-Things (IoT) is an ever growing network of devices connected to the Internet. Such devices are heterogeneous in their protocols and computation capabilities. With the rising computation and connectivity capabilities of these devices, the possibilities of their use in IoT systems increases. Concepts like smart cities are the pinnacle of the use of these systems, which involves a big amount of different devices in different conditions.

There are several tools for building IoT systems; some of these tools have different levels of expertise required and employ different architectures. One of the most popular is Node-RED. It allows users to build systems using a visual data flow architecture, making it easy for a non-developer to use it.

However, most of these mainstream tools employ centralized methods of computation, where a main component — usually hosted in the cloud — executes most of the computation on data provided by edge devices, *e.g.* sensors and gateways. There are multiple consequences to this approach: (a) edge computation capabilities are being neglected, (b) it introduces a single point of failure, and (c) local data is being transferred across boundaries (private, technological, political...) either without need, or even in violation of legal constraints. Particularly, the principle of Local-First — *i.e.*, data and logic should reside locally, independent of third-party services faults and errors — is blatantly ignored.

Previous work attempt to mitigate some of these consequences, usually through tools that extend existing visual programming frameworks, such as Node-RED. They go as far as to propose a solution to decentralize flows and its execution in fog/edge devices. So far, achieving such decentralization requires that the decomposition and partitioning effort be manually specified by the developer when building the system.

Our goal is to extend Node-RED to allow automatic decomposition and partitioning of the system towards higher decentralization, by inferring computational boundaries. Furthermore, through automatic detection of abnormal run-time conditions, we also intend to provide dynamic self-adaptation. The prototype developed will be first validated with real devices and later with simulations.

As a result, we expect to achieve a more robust and efficient execution of IoT systems, by leveraging edge and fog computational capabilities present in the network, and improving overall reliability.

Keywords: Internet of Things, Visual Programming, Edge Computing

Resumo

A Internet-of-Things (IoT) é uma rede de dispositivos conectados à Internet em constante crescimento. Estes dispositivos são heterogêneos nos seus protocolos e capacidades de computação. Com o crescimento das capacidades de computação e conectividade destes dispositivos, as possibilidades do seu uso em sistemas IoT aumentaram. Conceitos como Cidades Inteligentes são o pináculo do uso destes sistemas, que envolvem um grande número de dispositivos diferentes em diferentes condições.

Existem várias ferramentas para construir sistemas IoT; algumas destas ferramentas requerem diferentes níveis de perícia e usam diferentes arquiteturas. Uma das ferramentas mais populares é Node-RED. Esta permite aos seus utilizadores construir sistemas usando uma arquitetura visual de *data flow*, tornando o processo mais fácil para um utilizador não programador.

No entanto, a maioria das ferramentas convencionais usam métodos centralizados de computação, onde um componente principal - normalmente alocado na *cloud* - executa a maioria da computação nos dados provenientes dos dispositivos *edge*, *e.g.* sensores e *gateways*. Com esta abordagem estão associadas múltiplas consequências: (a) capacidades de computação de dispositivos *edge* estão a ser negligenciadas, (b) introduz um único ponto de falha, e (c) data local está a ser transferida através de limites (privados, tecnológicos, políticos...) sem necessidade ou violando restrições legais. Especificamente, o princípio de *Local-First* - *i.e.*, dados e lógica devem residir localmente, independentemente de falhas e erros de serviços terceiros - é totalmente ignorado.

Trabalhos feitos até agora tentam mitigar algumas destas consequências, construindo ferramentas que estendem ferramentas existentes de programação visual, como Node-RED. Algumas propõem uma solução que consiste na descentralização de *flows* e a sua execução em dispositivos de *fog* e *edge*. Atualmente, para obter a este tipo de descentralização é necessário que o esforço de decomposição e partição seja manualmente efetuado pelo programador quando este constrói o sistema.

O nosso objetivo é estender a ferramenta Node-RED para permitir a decomposição e partição automática do sistema com o fim de obter uma maior descentralização. Para isso é necessário deduzir os limites de computação do sistema. Para além disso, também pretendemos que o sistema se adapte automaticamente às mudanças do ambiente, detectando automaticamente condições anormais em *run-time*. O protótipo construído será validado, numa primeira fase, com dispositivos reais e, mais tarde, com o uso de simulações.

Como resultado, esperamos construir uma execução de sistemas IoT mais robusta e eficiente, aproveitando as capacidades de computação presentes nos dispositivos *edge* e *fog* da rede, e melhorando a confiança e segurança do sistema.

Keywords: Internet of Things, Visual Programming, Edge Computing

Acknowledgements

There is a big number of people that made it possible for me to finish my *Master's Degree* and this dissertation.

First of all, I would like to thank my supervisors Hugo Sereno Ferreira and André Restivo, who guided me through the dissertation while supplying knowledge and insights that improved my knowledge and work. I also appreciated the electronics lessons given which might have created my new hobby. I would also like to thank João Pedro Dias for the insights and suggestions given, which greatly improved the final work.

I would like to thank my friends, who supported me through all this time and provided the best quality jokes for me to laugh about. A special thanks to my *dissertation budies* from *IoTices*, who always listened to my ramblings and provided support and companionship.

To Pedro Reis, my boyfriend, for putting up with my many moods, motivating me, and reminding me when I needed to rest.

Lastly, a huge thank you to my parents and brother, who supported me through all my life and pushed me to be the best I can be.

Ana Margarida Silva

*“While it is always best to believe in oneself,
a little help from others can be a great blessing.”*

Uncle Iroh

Contents

1	Introduction	1
1.1	Context	1
1.2	Problem Definition	2
1.3	Motivation	3
1.4	Goals	3
1.5	Document Structure	3
2	Background	5
2.1	Internet-of-Things	5
2.1.1	IoT architectures	6
2.2	Visual Programming Languages	8
2.3	Node-RED	9
2.4	Summary	10
3	State of the Art	11
3.1	Systematic Literature Review	11
3.1.1	Methodology	11
3.1.2	Results	14
3.1.3	Expanded Search	22
3.1.4	Analysis and Discussion	23
3.1.5	Conclusions	25
3.2	Decentralized Architectures in Visual Programming Tools applied to the Internet-of-Things paradigm	26
3.2.1	DDF	26
3.2.2	<i>FogFlow & uFlow</i>	27
3.2.3	<i>FogFlow</i>	29
3.2.4	DDFlow	30
3.2.5	Analysis	31
3.2.6	Conclusion	32
3.3	Summary	33
4	Problem Statement	35
4.1	Current Issues	35
4.2	Desiderata	36
4.3	Scope	36
4.4	Main Hypothesis	37
4.5	Experimental Methodology	37
4.6	Summary	37

5	Solution	39
5.1	Overview	39
5.2	Implementation Details	40
5.2.1	Devices Setup for Decentralization Support	41
5.2.2	Decentralized Node-RED Computation	42
5.2.3	Summary	50
6	Evaluation	51
6.1	Scenarios and Experiments	51
6.2	Discussion	54
6.2.1	ES1: Sanity Checks	55
6.2.2	ES1: Experimental Tasks	59
6.2.3	ES2: Experimental Tasks	67
6.3	Hypothesis Evaluation	68
6.4	Lessons Learned	69
6.5	Conclusions	70
7	Conclusions	71
7.1	Difficulties	71
7.2	Future Work	72
7.3	Conclusions	73
A	Scenario 2 Results	75
B	Paper Submitted	79
	References	103

List of Figures

2.1	Fog Computing Architecture	7
2.2	Node-RED environment	9
2.3	Example of a Node-RED flow	10
3.1	Pipeline overview of the SLR Protocol.	14
3.2	Belsa et al. [10] solution architecture. The Modeller is a Node-RED flow editor canvas where new flows can be created by connecting nodes that correspond to the available services (Service Catalog and Discovery) which are then stored in the Flow Repository. The Orchestrator is responsible managing and running the specified flows as required (by running several instances of the Node-RED runtime) and converting Node-RED calls to the different IoT Platforms <i>native calls</i> (aided by the Semantic Mediator).	15
3.3	FRED [13] high-level architecture. FRED is based on Node-RED and addresses the limitation of running several flows in parallel (multiple runtimes) by orchestrating several instances of Node-RED (FRED-IS) using their FRED Proxy.	19
3.4	Node-RED [32] high-level architecture, identifying its development interface, runtime and <code>node.js</code> dependency. The <i>flows</i> can be versioned and organized in projects and new modules (<i>i.e.</i> , <i>nodes</i>) can be added using the <code>node.js</code> dependency manager tool (<i>i.e.</i> , <code>npm</code>).	22
3.5	Publications and tools of VPL tools applied to IoT per year.	24
3.6	Coordination between nodes in D-NR	27
3.7	Partition and assignment of parts of the flow	28
3.8	<i>FogFlow</i> architecture	28
3.9	<i>FogFlow</i> high level model	29
3.10	DDFlow architecture	30
5.1	Solution's overview, presenting three devices as orchestration <i>targets</i>	40
5.2	Firmware component diagram.	41
5.3	Simple Node-RED flow.	43
5.4	Node assignment example.	49
5.5	Sequence of events for orchestration.	50
6.1	Node-RED implementation of scenario 1	52
6.3	ES1-SC1 <i>node</i> assignment.	55
6.2	ES1-SC1 measurements.	56
6.4	ES1-SC1 script delivery time.	56
6.5	ES1-SC1 delivery times.	57
6.6	ES1-SC2 measurements.	58
6.7	ES1-SC2 script delivery time.	58

6.8	ES1-A measurements	60
6.9	ES1-B measurements	61
6.10	ES1-C measurements	62
6.11	ES1-D measurements	63
6.12	ES1-E measurements	64
6.13	ES1-F measurements	64
6.15	Number of devices active and inactive	65
6.14	Nodes assignment distribution	66
6.16	Node-RED implementation of scenario 2	67
6.17	ES2 results.	68

List of Tables

3.1	Inclusion and exclusion criteria.	13
3.2	Visual programming solutions applied to IoT and their characteristics. Small circles (●) mean <i>yes</i> , hyphens (-) means <i>no information available</i> , empty means <i>no</i> and asterisk (*) means more than one.	21
3.3	Characterization of VPLs applied to IoT from survey.	23
3.4	Decentralized visual programming solutions applied to IoT and their characteristics.	31
5.1	Comparison between the Espressif Systems ESP32 and ESP8266 systems on chip.	41
6.1	Scenario 2 results	67
A.1	Node-RED original results	75
A.2	Node-RED + MQTT results	75
A.3	Node-RED modified + Dockers (same host) results	76
A.4	Node-RED modified + Dockers (different host) results	76
A.5	Physical + MQTT results	76
A.6	Node-RED modified + MQTT + Physical + Firmware	77

Abbreviations

API	Application Programming Interface
CPSCN	Cyber Physical Social Computing and Networking
CPU	Central Processing Unit
HTTP	Hypertext Transfer Protocol
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
IFTTT	<i>If This Then That</i>
IoT	Internet-of-Things
JSON	JavaScript Object Notation
MQTT	Message Queuing Telemetry Transport
MTTR	Mean Time To Recover
QoS	Quality of Service
RaaS	Robot as a Service
IoIT	Internet of Intelligent Things
REST	Representational State Transfer
URL	Uniform Resource Locator
VPL	Visual Programming Language
WWW	<i>World Wide Web</i>

Chapter 1

Introduction

1.1	Context	1
1.2	Problem Definition	2
1.3	Motivation	3
1.4	Goals	3
1.5	Document Structure	3

This chapter introduces the motivation and scope of this project, as well as the problems it aims to solve. Section 1.1 details the context of this project in the area it is based on. Section 1.2 defines the problem we aim to solve. Then, Section 1.3 explains the reason why this work and the area it belongs to is important and the goals of this dissertation are described in Section 1.4. Finally, Section 1.5 describes the structure of this document and what content it contains.

1.1 Context

The Internet-of-Things (IoT) paradigm states that all devices, independently of their capabilities, are connected to the Internet and allow for the transfer, integration and analytic of data generated by them [17]. This paradigm has several characteristics, such as the heterogeneity and high distribution of devices as well as their increasing connectivity and computational capabilities [7]. All these factors allow for a great level of applicability, enabling the realization of systems for the management of cities, health services, and industries [20].

The interest in Internet-of-Things has been growing massively, following the rising of connected devices along these past years. According to Siemens, there are around 26 billion physical devices connected to the Internet in 2020 and predictions are pointing at 75 billion in 2025 [6]. Although this allows for more opportunities, it is important to note that these devices are very different in

their hardware and capabilities, which causes several problems in terms of developing the systems, as well as their scalability, maintainability, and security.

Visual Programming Languages (VPLs) allow the user to communicate with the system by using and arranging visual elements that can be translated into code [19]. It provides the user with an intuitive and straightforward interface for coding at the possible cost of losing functionality. There are several programming languages with different focuses, such as education, video game development, 3D building, system design, and even Internet-of-Things [56]. Node-RED¹ is one of the most famous open-source visual programming tools, originally developed by IBM's Emerging Technology Services team and now a part of the JS Foundation, which provides an environment for users to develop their own Internet-of-Things systems.

Non-functional attributes in a system are very important, specially attributes such as resiliency, fault-tolerance, and self-healing in Internet-of-Things systems. All these attributes mean that when an error or problem occurs, the system can adapt and overcome them dynamically and automatically.

Node-RED, mentioned above, is a centralized system, as well as most of the visual programming environments applied to IoT. A centralized architecture has a central instance that executes all computational tasks on the data provided by the other devices in the network. On the other hand, in a decentralized architecture the central instance, if it exists, partitions the computational tasks in independent blocks that can be executed by other devices. In IoT, these decentralized architectures are mentioned in Fog and Edge computing.

1.2 Problem Definition

Most mainstream visual programming tools focused on Internet-of-Things, Node-RED included, have a centralized approach, where the main component executes most of the computation on data provided by edge devices, e.g. sensors and gateways. There are several consequences to this approach: (a) computation capabilities of the edge devices are being ignored, (b) it introduces a single point of failure, and (c) local data is being transferred across boundaries (private, technological, political...) either without need or even in violation of legal constraints. The principle of Local-First [42] - i.e, data and logic should reside locally, independent of third-party services faults and errors - and NoCloud [55] - i.e, on-device and local computation should be prioritized over cloud service computation - is being ignored.

Besides being a single point of failure, centralized systems can be less efficient than decentralized ones and in this context, it might be the case, since there are computation capabilities that aren't being taken advantage of.

Chapter 4 expands on the problem definition, explaining it in bigger detail, defining its scope, desiderata, use cases and research questions.

¹<https://nodered.org/>

1.3 Motivation

Internet-of-Things is a rapidly growing concept that is being applied to several areas, such as home automation, industry, health, city management, and many others. Given the number of existing systems with different protocols and architectures, it becomes difficult for a user to build a system that is in accordance with standards [5].

With the appearance of visual programming languages focused in IoT, more specifically Node-RED, users can build their own systems in an easier and streamlined way, removing the overhead of learning advanced programming concepts and protocols. These tools must be resilient, in order to withstand flaws and non-availability of devices as well as failure in the network. However, the majority of these tools are centralized, including Node-RED, and this type of architecture hinders the resiliency of the system. Given the existence of only one unit that executes most or all the processing of data, if this device fails, the system becomes nonfunctional. A possible solution would be increasing the redundancy of the system, creating more than one instance of the main unit. However, this approach has several costs, not only monetary but also in the increase in complexity.

1.4 Goals

The main goal of this dissertation is to leverage the computation capabilities of the devices in the network, increasing efficiency, fault-tolerance, resiliency and scalability in an Internet-of-Things system.

To achieve this goal, a prototype will be developed, extending or rewriting Node-RED, which enables IoT devices to communicate their "computational capabilities" back to the orchestrator. In its turn, the orchestrator is able to partition the computation and send "tasks" to the nodes, which are the devices in the network, leveraging their computation power and independence.

As a secondary goal, several other challenges will be tackled, viz: (i) computational capabilities of the devices in the network, (ii) detecting non-availability and using alternative computation resources, and (iii) exploring different alternatives of leveraging current IoT devices, including using firmwares that allow the execution of programs written in Lua, Javascript, Python, etc., amongst others.

1.5 Document Structure

Chapter 2 introduces the background information and explanation about concepts necessary for the full understanding of this dissertation. Chapter 3 describes the state of the art regarding the ecosystem of this project's scope, including a Systematic Literature Review on the state of the art of visual programming applied to the Internet-of-Things paradigm. Chapter 4 presents the problem this dissertation aims to solve, as well as the approach taken to solve it. Chapter 5 details how the solution was implemented and all the decisions and efforts taken to answer the problem statement mentioned before. Chapter 6 analyzes the evaluation process and demonstrates the validation and

evaluation of the developed solution. Finally, Chapter 7 concludes this dissertation with a reflection on the success of the project by presenting the a summary of the developed work and detailing the difficulties and future work.

Chapter 2

Background

2.1 Internet-of-Things	5
2.2 Visual Programming Languages	8
2.3 Node-RED	9
2.4 Summary	10

This chapter describes the necessary foundations regarding visual programming tools for the Internet-of-Things context. Section 2.1 describes the background of the Internet-of-Things paradigm and important concepts in that area, with a description of IoT architectures, including Fog and Edge, in Section 2.1.1. Section 2.3 describes the Node-RED programming tool and its architecture and uses. Finally, Section 2.2 mentions visual programming languages, their uses as well as their benefits and drawbacks.

2.1 Internet-of-Things

Internet-of-Things paradigm is defined by the committee of the International Organization for Standardization and the International Electrotechnical Commission [1] as:

"An infrastructure of interconnected objects, people, systems and information resources together with intelligent services to allow them to process information of the physical and the virtual world and react."

IoT systems are, mostly, networks of heterogeneous devices attempting to bridge the gap between people and their surroundings. According to Buuya [37], the applications of IoT systems can be divided into four categories: (i) *Home* at the scale of a few individuals or domestic scenarios, (ii) *Enterprise* at the scale of a community or larger environments, (iii) *Utilities* at a national or

regional scale and (iv) *Mobile*, which is spread across domains due to its large scale in connectivity and scale.

One might think that IoT only relates to machines and interactions between them. Most of the devices we use in our day-to-day — *e.g.*, mobile phones, security cameras, watches, coffee machines — are now computation capable of making moderately complex tasks and are continually generating and sending information. This relates to the *human-in-the-loop* concept, where humans and machines have a symbiotic relationship [51].

2.1.1 IoT architectures

Internet-of-Things systems deal with big amounts of data from different sources and have to process it in an efficient and fast fashion. Typical IoT systems are composed of three tiers, which are:

Cloud Tier mostly composed of data centers and servers, normally running remotely. It is characterized by having high computation power and latency.

Fog Tier composed of gateways and devices that are normally between the cloud servers and the edge devices. This tier has less latency than the cloud, more heterogeneity, and, typically, is more geographically distributed.

Edge Tier composed of all the peripheral devices (*e.g.*, sensors, embedded systems, light sources and air conditioners). These devices have several limitations in computational capabilities, but have less latency.

Complementary to these tiers, we can also partition IoT systems into an Application Layer, a Network Layer, and a Perception Layer [47]. At first sight, these might seem compatible with the tiers mentioned above (in the same order); however, not all devices in each tier map to their respective layer. One example is a third-party service that gives readings. It can be contained in the Perceptive Layer, but it is not included in the Edge Tier.

New paradigms of computing appeared related to each of these tiers. The majority of IoT systems use a Cloud Computing architecture, taking advantage of centralized computing and storage. This approach has several benefits, such as increased computational capabilities and storage, as well as easier maintenance. However, it comes with several problems such as (1) high latency, and (2) high use of bandwidth, due to the need to send the data generated from the sensors to the centralized unit [43]. Systems that only use cloud computing face several challenges [2], especially real-time applications, which are sensitive to increased latency. With the increasing computation capabilities of edge devices and the requirement of reduced latency, two new paradigms appeared: Fog Computing and Edge Computing.

2.1.1.1 Fog Computing

With the improvement of wireless technologies and the increasing computational power (and reducing costs) of lower-tier devices (*i.e.*, fog and edge), it became possible to improve the computational

execution of IoT systems. By not depending so much on the cloud tier, communication and resource sharing between devices can occur with lower latency. The central coordinator (on-premises or cloud-based), which in Cloud Computing was responsible for all the computation, now serves as a scheduler and state manager of the communication between devices, occasionally providing necessary resources. This new paradigm, where fog and edge devices are leveraged as computational entities (and not only merely sensing, actuating and gateway devices in the network) is called Fog Computing, which aims to bring computing closer to the perception tier, bringing the computation nearer to the edge of the network [44]. It focuses on distributing data throughout the IoT system, from the cloud to the edge devices, making the system distributed.

According to Buuya [17], Fog Computing has several advantages: (1) reduction of network traffic by having edge devices filtering and analyzing the data generated and sending data to the cloud only if necessary, (2) reduced communication distance by having the devices communicate between them without using the cloud as a middleman, (3) low-latency by moving the processing closer to the data source rather than communicating all the data to the cloud for it to be processed, and (4) scalability by reducing the burden on the cloud, which could be a bottleneck for the system.

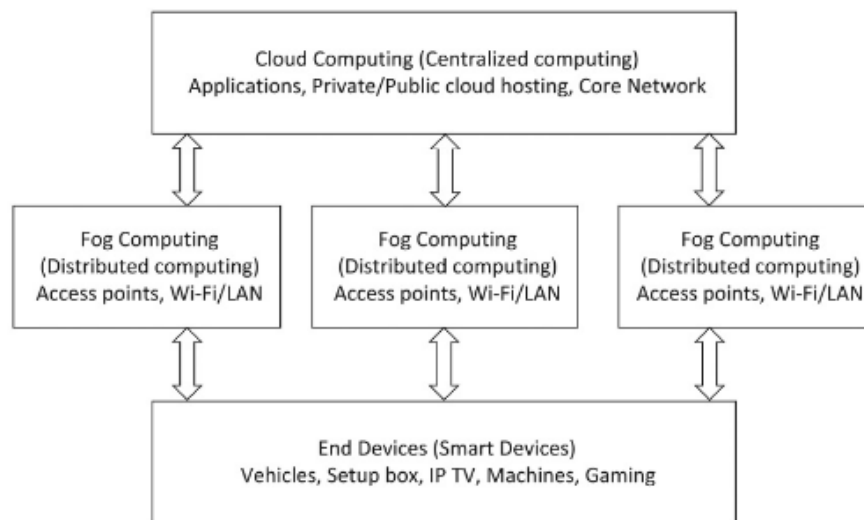


Figure 2.1: Fog Computing Architecture [17]

Despite all the advantages, Fog Computing has several requirements and difficulties. To make a successful and efficient distribution of computation and communication, it requires knowledge about the resources of the connected devices. The complexity is also more significant than Cloud Computing since it needs to work with heterogeneous devices with different capacities.

2.1.1.2 Edge Computing

Edge Computing, also known as Mist Computing, is a distributed architecture that uses the devices' computational power to process the data they collect or generate. It takes advantage of the Edge tier, which contains the devices closer to the end-user, such as smartphones, TVs and sensors. The goal

of this paradigm is to minimize the bandwidth and time response of IoT systems while leveraging the computational power of the devices in them. It reduces bandwidth usage by processing data instead of sending it to the cloud to be processed, which is also correlated to reduced latency since it does not wait for the server response. In addition to these advantages, and related to their cause, Edge Computing also prevents sensitive data from leaving the network, reducing data leakage and increasing security and privacy [46, 61].

In this paradigm, each device serves both as a data producer and a data consumer. Since each device is constrained in terms of resources, this brings several challenges such as system reliability and energy constraints due to short battery life and overall security. Other issues consist of the lack of easy-to-use tools and frameworks to build cloud-edge systems, non-existent standards regarding the naming of edge devices and the lack of security edge devices have against outside threats such as hackers [60].

There is some confusion in the research community regarding the concepts of Fog and Edge computing. The publication from Iorga et al. [40] was used to inspire the definitions of these terms. Edge Computing focuses on executing applications in constrained devices, without worrying about storage or state preservation. On the other hand, Fog Computing is hierarchical and includes devices with more capabilities, capable of control activities, storage, and orchestration.

2.2 Visual Programming Languages

Visual Programming, as defined by Shu [62], consists of using meaningful graphical representations in the process of programming. With this definition, we can consider Visual Programming Languages (VPLs) as a way of handling visual information and interaction with it, allowing the use of visual expressions for programming. According to Burnet and Baker [15], visual programming languages are constructed to *"improve the programmer's ability to express program logic and to understand how the program works"*.

There are several applications of visual programming languages in different areas, such as education, video game development, automation, multimedia, data warehousing, system management, and simulation, with this last area being the area with most use cases [56].

Visual programming languages have several characteristics, such as a concrete process and depiction of the program, immediate visual feedback and require the knowledge of fewer programming concepts [15].

VPLs can be categorized by their visual paradigms and architecture [14]:

Purely Visual Languages where the system is developed using only graphical elements and the subsequently debugging and execution is made in the same environment.

Hybrid text and visual systems where the programs are created using graphical elements, but their executions is translated into a text language.

Programming-by-example systems where a user uses graphical elements to teach the system.

Constraint-oriented systems where the user translates physical entities into virtual objects and applies constraints to them, in order to simulate their behaviour in reality.

Form-based systems which are based on the architecture and behaviour of spreadsheets.

The categories mentioned can be present in a single system, making them not mutually exclusive.

2.3 Node-RED

Node-RED [32] is a programming tool applied to the development of Internet-of-Things systems. It was first developed with to manipulate and visualize mappings between MQTT topics in IBM's Emerging Technology Services group. It then expanded into a more general open-source tool, which is now part of the JS Foundation.

It is a web-based tool consisting of a run time built with the Node.js framework and a browser-based visual editor. This tool provides the end-user with a simple interface to connected devices and APIs, using a flow-programming approach. Programs are called *flows*, built with *nodes* connected by wires. Each node corresponds to an action, such as input, output, data processing, etc.

The Node-RED interface has three components: (1) Palette, (2) Workspace and (3) Sidebar. The Palette contains all the nodes installed and available to use, divided into categories. They can be used by dragging them into the workspace and additional features for each node are accessible by double-clicking them. The Workspace is where the flows are created and modified. It is possible to have several *flows* and *sub-flows* accessible with the use of tabs. Lastly, the Sidebar contains information about the nodes, the debug console, node configuration manager and the context data. Figure 2.2 showcases the visual interface of Node-RED and its elements.



Figure 2.2: Node-RED environment

One example of a *flow* can be seen in Figure 2.3, where a request is being made in intervals of 5 minutes to an HTTP URL that returns a CSV with the feed of significant earthquakes in the last 7 days. The data from the CSV is then printed to the debug console and, if the magnitude is equal or bigger than 7, the message "PANIC!" is printed to the console.

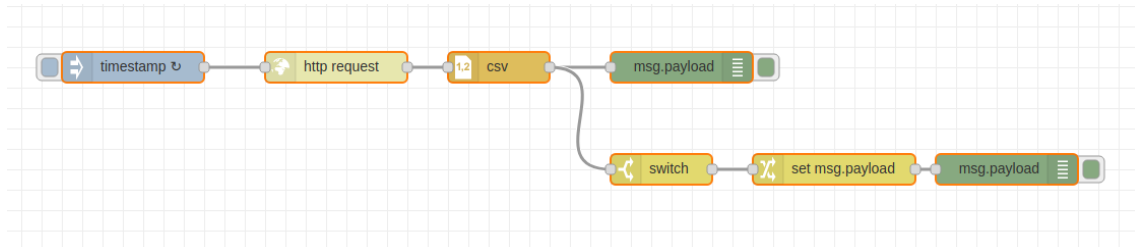


Figure 2.3: Example of a Node-RED flow

Regarding the architecture of Node-RED, the `Node` base class is a subclass of Node.js event APIs `EventEmitter`. This class implements an observer design pattern that maintains a subscriber list of all the nodes connected to it by *wires* and emits events to them. When a node finishes processing data from external sources or another node, it calls the methods `send()` with a Javascript object. In its turn, this method calls the `EventEmitter emit()` method that sends named events to the subscribed nodes.

Being open-source, Node-RED takes advantage of a large community that contributes with new nodes and improvements to the tool. It is the most popular open-source visual programming tool for IoT, with more than 9,300 stars on Github.

2.4 Summary

This chapter introduces two areas that are fundamental to the understanding of this dissertation. Internet-of-Things is defined, as well as its use cases and categories. Fog and Edge computing paradigms are explained, which will be mentioned throughout this document. Node-RED is introduced as a visual programming tool for IoT and its architecture is explained. Finally, a definition and categorization of visual programming languages are introduced and explained.

Chapter 3

State of the Art

3.1	Systematic Literature Review	11
3.2	Decentralized Architectures in Visual Programming Tools applied to the Internet-of-Things paradigm	26
3.3	Summary	33

This chapter describes the state of the art in visual programming tools in the Internet-of-Things context, as well as decentralized methods of work distribution in flow-based architectures. Section 3.1 presents a systematic literature review on the topic of visual programming tools applied to the Internet-of-Things paradigm, which aims to answer the research questions defined in Section 3.1.1.1. Section 3.1.2 contains the results of the Systematic Literature Review, as well as their categorization. Section 3.1.3 contains the additional tools found in a survey and their analysis. The discussion and analysis of the tools found as well as the answering of the research questions made previously are made in Section 3.1.4. The Systematic Literature Review conclusions are presented in Section 3.1.5. Lastly, Section 3.2 contains the state of the art of visual programming tools applied to IoT that implement a decentralized architecture.

3.1 Systematic Literature Review

A Systematic Literature Review was made to gather information on the state of the art of visual programming applied to the Internet-of-Things paradigm. The goal of a systematic literature review is to synthesize evidence with emphasis on the quality of it [53].

3.1.1 Methodology

During this SLR, a specific methodology was followed to reduce bias and produce the best results [53]. We started by defining the research questions to be answered as well as choosing data

sources to search for publications.

3.1.1.1 Research Questions

To reveal the current practice, research and studies related to orchestration in the Internet-of-Things that leverage visual approaches, which enable us to find the current, and pending research challenges, we outline the following research questions (RQ):

RQ1 *What relevant visual programming solutions applied to IoT orchestration exist?* Internet-of-Things is a paradigm with several years, and its integration with visual programming languages makes their development easier for the end-user. The tools that integrate these two paradigms are useful and reduce the overhead of programming or prototyping IoT systems.

RQ2 *What is the tier and architecture of the tools found in RQ1?* IoT systems can belong to one or more of tiers — Cloud, Fog and Edge — as well as implement a centralized or decentralized architecture. A visual programming tool applied to IoT orchestration can be used to facilitate the development of systems that operate on these tiers. Each tier and type of architecture offers vantages and disadvantages, which are essential to understand the usages and characteristics of a system.

RQ3 *What was the evolution of visual programming solutions applied to IoT orchestration over the years?* To understand the field of visual programming applied to IoT, more specifically, its orchestration, it is essential to perceive its evolution.

Answering these questions will provide insights that can be valuable for both practitioners — in terms of summarizing what the current practices on the usage of visual programming methodologies for IoT orchestration are — and researchers — showing current challenges and issues that can be further researched.

3.1.1.2 Databases

The publications retrieved during this research were retrieved from the following databases:

- IEEE
- ACM
- Scopus

These electronic databases contain some of the most relevant digital literature for studies in the area of Computer Science, thus being considered reliable sources of information.

Table 3.1: Inclusion and exclusion criteria.

I/E	ID	Criterion
Exclusion	EC1	Not written in English.
	EC2	Presents just ideas, tutorials, integration experimentation, magazine publications, interviews or discussion papers.
	EC3	Presents a tool, framework or approach that does not support the orchestration of multiple devices.
	EC4	Has less than two (non-self) citations when more than five years old.
	EC5	Duplicated articles.
	EC6	Articles in a format other than camera-ready (PDF).
Inclusion	IC1	Must be on the topic of visual programming in Internet-of-Things.
	IC2	Contributions, challenges and limitations are presented and discussed in detail.
	IC3	Research findings include sufficient explanation on how the approach works.
	IC4	Publication year in the range between 2008 and 2019.
	IC5	Is a survey that focus visual programming in IoT or

3.1.1.3 Candidate Searching and Filtering

Our systematic literature review protocol followed the inclusion and exclusion criteria detailed in Table 3.1 and is outlined in Figure 3.1 (p. 14).

We begun our search in these data sources using a query that captured the most probable keywords to appear in our target candidates, namely *visual programming*, *node-red*, *dataflow*, and *Internet-of-Things*. This led us to specify variants of the following query that are understood by the mentioned databases: `((vpl OR visual programming OR visual-programming) OR (node-red OR node red OR nodered) OR (data-flow OR dataflow)) AND (IoT OR Internet-of-Things OR internet-of-things)`. This search was performed in October of 2019 and the number of results produced can be seen in the first step of Figure 3.1 (p. 14).

The evaluation process of the publications then followed eight steps with specific purposes:

1. **Automatic Search:** Run the query string in the different scientific databases and gather results;
2. **Filtering** (*EC1*, *IC4*, and *EC6*): Publications are selected regarding its (1) language, being limited to the ones written in English language, (2) publication date, being limited to the ones published between 2008 and 2019, and (3) publication status, being selected only the ones that are published in their final versions (camera-ready PDF format);
3. **Filtering to remove duplicates** (*EC5*): The selected papers are filtered to remove duplicated entries;
4. **Filtering by Title and Abstract** (*EC2–EC4*, and *IC1–IC3*): Selected papers are revised by taking into account their *Title* and *Abstract*, by observing the (1) stage of the research, only

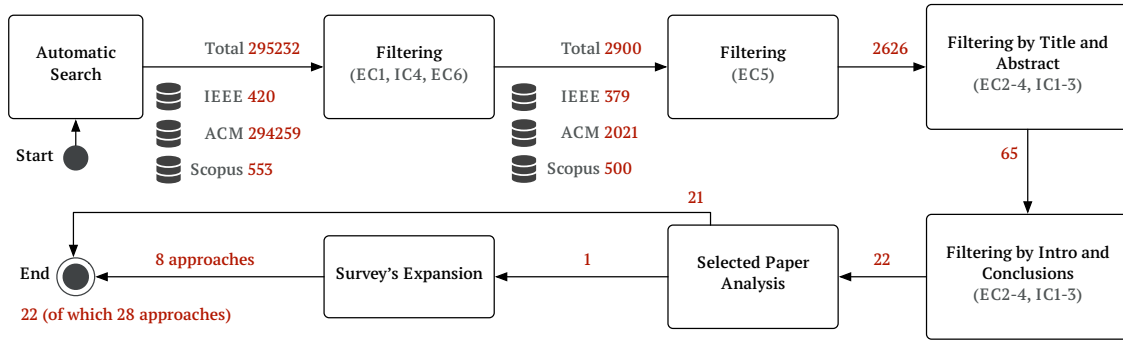


Figure 3.1: Pipeline overview of the SLR Protocol.

selecting papers that present approaches with sufficient explanation, some experimental results and discussion on the paper contributions, challenges and limitations, (2) contextualization with recent literature, filtering papers that have less than two (non-self) citations when more than five years old, and (3) leverages the use of visual notations for orchestrating and operating multi-device systems.

5. **Filtering by Introduction and Conclusions** (EC2–EC4, and IC1–IC3): The same procedure of the previous point is followed but taking into consideration the *Introduction* and *Conclusion* sections of the papers;
6. **Selected Papers Analysis**: Selected papers are grouped, and surveys are separated; their content is analyzed in detail.
7. **Surveys Expansion**: For the survey papers found, the enumerated solutions are analyzed and filtered taking into account their scope and checking if they are not duplicates of the current selected papers.
8. **Wrapping**: Approaches and solutions gathered from the *Selected Papers Analysis* (individual papers) and from the *Survey Expansion* are presented and discussed.

The total number of publications was 2698, and, after the evaluation process, 22 publications were selected as can be seen in Figure 3.1. From those, one was a survey and the others presented approaches relevant to our research questions.

3.1.2 Results

After analyzing the 22 publications, we organized them by categories; of these, one was a survey [56], and the remaining 21 were papers describing papers that address our research questions. In that survey, the authors make an in-depth review of 13 visual programming languages in the field of IoT, comparing them using four attributes: (1) programming environment, (2) license, (3) project repository and (4) platform support. We used this survey to complement our research in Section 3.1.3 (p. 22).

The selected 21 articles described approaches that use visual programming in the IoT context having orchestration considerations. One of the tools is described in two papers, which showcases its evolution. The 20 unique solutions are:

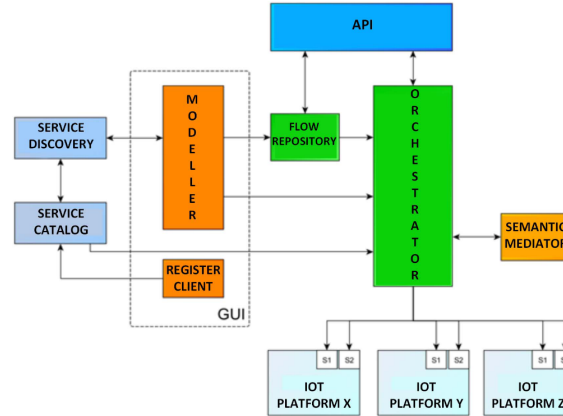


Figure 3.2: Belsa et al. [10] solution architecture. The Modeller is a Node-RED flow editor canvas where new flows can be created by connecting nodes that correspond to the available services (Service Catalog and Discovery) which are then stored in the Flow Repository. The Orchestrator is responsible for managing and running the specified flows as required (by running several instances of the Node-RED runtime) and converting Node-RED calls to the different IoT Platforms *native calls* (aided by the Semantic Mediator).

- **Belsa et al. [10]** present a solution for connecting devices from different IoT platforms, using Flow-Based Programming with Node-RED, depicted in Figure 3.2. Its motivation is based on the limitation imposed by the IoT platform on communication between components and extensibility, which limits the possibility to interact with other platforms' services. To validate their solution, they implemented a use case in the domain of transportation and logistics, with a service that uses five different types of applications. The developed tool offers access to available services in a centralized visual framework, where end-users can use them to build more complex applications.
- **Ivy [26]** proposes the next step toward visualization applied to IoT with a visual programming tool that allows its users to link devices, inject logic, and visualize real-time data flows using immersive virtual reality. It provides the end-users with an immersive virtual reality that allows them to visualize the data flow, access to debugging tools, and real-time deployment. Each programming construct called node - data flow architecture - has a distinct shape and colour, which facilitates the understanding of the system being built or debugger for the user. The experiences made to validate the prototype were positive, with the participants being receptive to Ivy and proposing new use cases.
- **Ghiani et al. [33]** proposition is to build a collection of tools that allow non-developer users to customize their Web IoT applications using trigger-actions rules. The proposed solution provides a web-based tool that allows users to specify their trigger-action rules using *IFTTT*,

as well as a context manager middleware that can adapt to the context and events of the devices and apply rules to the system. To validate the developed tool, an example home automation application that displays sensor values and directly controls appliances were built. The results were, for the most part, positive, and the issues found are related to usability and visual clues.

- **ViSiT** [4] uses the jigsaw puzzle metaphor [39] to allow its end-users to implement a system of connected IoT devices. It provides a web-based visual tool connected with a web-service that, given a jigsaw representation, generates an executable implementation. Their goal is achievable by adapting model transformations used by software developers into intuitive metaphors for non-developers to use. They validated the developed tool with a usability evaluation, which was overall positive, with a significant percentage considering the tool useful and providing real-life scenarios where they could implement it.
- **Valsamakis and Savidis** [69] propose a framework for Ambient Assisted Living (AAL) using IoT technologies, which allows for customized automation. It uses visual programming languages to facilitate their end-users - carers, family, friends, elderly - to build and modify automation. They built a visual programming framework that introduces smart objects grouping in tagged environments and real-time smart-object registration through discovery cycles. It runs on typical smartphones and tablets and is built in Javascript, allowing it to run in browsers. Their future work focuses on integrating different visual programming paradigms to accomplish the requirements of the end-user fully.
- **WireMe** [52] is a solution for building, deploying, and monitoring IoT systems, built with non-developer end-users in mind but also extensible for advanced users to build over it. The developed solution makes use of Scratch, a visual programming interface, to provide its users with a customizable dashboard where they can monitor and control their IoT system as well as program automation tasks. It has a Main Control Unit responsible for communicating the device's status to the dashboard via MQTT, which is programmable using their visual interface and Lua programming language. Their tool was validated in an empirical study with students around 16 years old and engineering students without programming experience. The results were not positive, with some students not being able to create the required simple logic. Future work consists of improving programming blocks to become more intuitive.
- **VIPLE** [23], Visual IoT/Robotics Programming Language Environment, is a new visual programming language and environment. It provides an introduction to topics such as computing and engineering and tools for more technical domains like software integration and service-oriented computing. It focuses on complex concepts such as robot as a service (Raas) units and Internet of Intelligent Things (IoIT) while studying the programming issues of building systems classified as such. The developed tool has been tested and used in several universities since 2015 due to its large set of features and use cases.

- **Smart Block** [9] is a block-based visual programming language and visual programming environment applied to IoT systems, that allows non-developer users to build their systems quickly. Their solution is specific to the home automation domain, like Smart Things. The language was designed using IoTa calculus, used to generalize Event-Condition-Action rules for home automation. The environment was built using a client-side Javascript library called Blockly, which allows for the creation of visual block languages. Future work for this project consists of supporting device grouping and security by expanding custom blocks, as well as extending the tool for other domains besides home automation.
- **PWCT** [31] is a visual programming language applied to build IoT, Data Computing, and Cloud Computing systems. Its goal consists of reducing the cost of development of these types of systems by providing a comfortable and more productive development tool. The language was meant to compete with text-based languages such as Java and C/C++. It makes use of graphical elements to replace code. It has three main layers: (1) the VPL layer, composed of graphical elements, (2) the middleware layers, responsible for connecting the VPL layer to the system's view, which is the (3) System Layer, responsible for dealing with the source code generated by the first layer. The created solution received positive feedback from the community, with more than 70,000 downloads and 93% of user satisfaction.
- **DDF** [36] is a Distributed Dataflow (DDF) programming model for IoT systems, leveraging resources across the Fog and the Cloud. They implemented a DDF framework extending Node-RED, which, by design, is a centralized framework. Their motivation comes from the possibility to develop applications from the perspective of Fog Computing, leveraging these devices for efficiency and reduced latency, since there is a significant amount of resources such as edge devices and gateways in IoT systems. They evaluated their prototype using a small scale evaluation, which was positive. The results showed that their DDF framework provides an alternative for designing and developing distributed IoT systems, despite having some open issues such as not having a distributed discovery of devices and networks.
- **GIMLE** [67], Graphical Installation Modelling Language for IoT Ecosystems, is a visual language that uses visual elements to model domain knowledge using significant ontological requirements. The goal of this language is to fill the gap of modeling requirements on the physical properties of IoT installations by proposing a new process for configuring industrial installations. It makes use of flow-based and domain-based visual programming to isolate the requirements' logical flow from their details. The developed tool supports reuse within the models, which is valuable due to the repetitive nature of industrial installations. However, it still needs to clarify its scope within the current practice and its use in production settings.
- **DDFlow** [50] is a macro-programming abstraction that aims to provide efficient means to program high quality distributed apps for IoT. The authors point to a lack of solutions for complex IoT systems programming, causing developers to build their systems, which leads to a lack of portability/extensibility and results in a lot of similar systems that do the same thing

but are “different” because different programmers created them. Developers use Node-Red to specify the application functionalities, and DDFlow handles scalability and deployment. The authors describe DDFlow’s goal to allow developers to formulate complex applications without having to care about low-level network, hardware, and coordination details. This is done by having the DDFlow accompanying runtime dynamically scaling and mapping the resources, instead of the developer. DDFlow gives developers the possibility to inject custom code on nodes and has custom logic if the available nodes are not enough for some tasks.

- **Kefalakis et al. [41]** proposition consists of a visual environment that operates over the OpenIoT architecture and allows for the development of IoT applications with reduced programming effort. Modeling IoT services with the developed tool is made by specifying a graph that corresponds to an IoT application, which can be validated and have its code generated and performed over the OpenIoT middleware platform. It aims to fill the gap of tools that provide support for the development and deployment of integrated IoT applications. The approach taken presents several advantages: (1) it leverages standards-based semantic model for sensor and IoT context, making it easier to be widely adopted, (2) it is based on web-based technologies which open the possibilities of applications from developers and (3) it is open source.
- **Eterovic et al. [29]** propose an IoT visual domain-specific modeling language based on UML, with technical and non-technical users in mind. The authors defend that, with the evolving nature of IoT, the future end-user will be a non-technical person, with no programming knowledge. Attaining the issues that this can create in the future, it is crucial to create a visual language easy enough to be understood by non-technical people but expansible enough to represent complex systems. To evaluate the proposed solution, they invited 11 users of different levels of UML expertise to model a simple IoT system with the developed language. The System Usability Score was positive, as well as the Tasks Success Rate. Despite the positive score, some future actions would be the testing of the language with a more complex task as well as the integration of advanced UML notations.
- **FRED [13]** is a frontend for Node-RED, a development tool that makes it possible to host multiple Node-RED runtimes (Figure 3.3 (p. 19)). It can be used to connect devices to services in the cloud, manage communication between devices, create new web app applications, APIs and event-integrated services. To provide all these features, FRED allows the running of flows for multiple users, in which all flows get fair access to resources such as CPU, memory, storage, as well as secure access to flow editors and the flow runtime. The authors concluded that FRED is useful for users learning about Node-RED and allows users to prototype cloud-hosted applications rapidly.
- **WoTFlow [12]** is proposed as a cloud-based platform that aims to provide an execution environment for multi-user cloud environments and individual devices. It aims to take advantage of data flow programming, which allows parts of the flow to be executed in

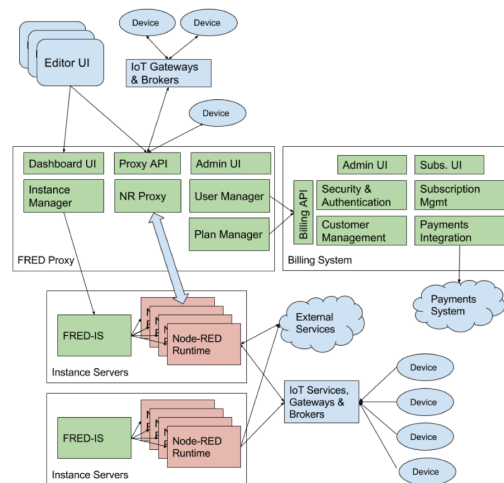


Figure 3.3: FRED [13] high-level architecture. FRED is based on Node-RED and addresses the limitation of running several flows in parallel (multiple runtimes) by orchestrating several instances of Node-RED (FRED-IS) using their FRED Proxy.

parallel in different devices. Based on this, the tool will take advantage of the ability to split and partition the flows and distribute them by edge devices and the cloud. The state of the developed tool was in the early stages, with future expansions based on the use of optimization heuristics, automatic partitioning based on calculated constraints, security, and privacy.

- **Besari et al.** [59, 11] proposes an IoT-based GUI that aims to control sensors and actuators in an IoT system using an android application, in which the users use a visual programming language to configure and interact with the IoT system. The system was tested with a Pybot, a robot that is programmable like an IoT system, with sensors and actuators. After testing and evaluating the system, the authors came to a score of 72.917 (out of 100) for the Pybot software, which is considered “good”. The overall acceptability of the system was "ACCEPTABLE", which led the authors to consider the application accepted by users.
- **CharIoT** [66] is a programming environment that promises its end-users a solution that unifies and supports the configuration of IoT environments. It provides three blocks of support: capturing higher-level events using virtual sensors, construction of automation rules with a visual overview of the current configuration and support for sharing configuration between end-users using a recommendation mechanism. Two types of virtual sensors were developed to capture higher-level events. The programmed virtual sensor provides more accessible and understandable abstractions (defining that a room is “cold” if the temperature is below 20°C). The demonstrated virtual sensors are more complex, requiring the user to provide a demonstration of the occurrence and lack of occurrence of the event (for example, the event of someone knocking on the door and the absence of someone knocking on the door). This last one requires the training of a Random Forest classifier. This programming environment is similar to IFTTT but goes one step further, with smarter event capturing

and reusing of configurations, allowing the end-user to build faster and more robust IoT installations.

- **Desolda et al. [24]** proposition uses a tangible programming language that allows non-programmers to configure smart objects' behaviour to create and customize smart environments. The main goal was to create, with the developed technology, a scenario of a smart museum. The authors defend that the synchronization of smart devices cannot limit the personalization of a smart environment, and it may require experts to build their narrative, much like a museum said. With this in mind, they introduced custom attributes to assign semantics to connected objects to empower and simplify the creation of event-condition-action rules. This is ongoing research focused on developing new technology with an interaction paradigm that supports the input of domain experts in the creation of smart environments. The fact that this technology uses expensive material (tabletop surface as a digital workspace) does not allow a regular user to use it, as stated in the introduction.
- **Eun et al. [30]** proposes an End-User Development (EUD) tool that allows users to develop their applications. It uses the dataflow approach, which allows for a more generalized programming experience as well as the facility to build more complex programs with simple modules. The proposed tool has three main components: Service Template Authoring Tool, Service Template Repository, and Smartphone Application. The first one allows the end-user to build more complex methods using atomic templates (components with simple functionality, like opening a curtain if it receives a command). The Service Template Repository contains the proprietary atomic templates as well as ones built by the user. Lastly, the Smartphone Application runs and manages the applications built by the user, as well as their requirements and dependencies. The developed EUD tool was compared with *IFTTT* and Zapier; other tools focused on end-user development. *IFTTT* and the developed tool are similar, focusing on consumer development, IoT, and home environments, with Zapier focusing on business environments. Both Zapier and *IFTTT* use the Trigger-Action paradigm (TAP), which differs from the dataflow paradigm used in this paper's tool.

The mentioned frameworks and tools were divided into the following categories, according to several characteristics:

1. **Scope.** Some tools have specific use cases in mind. Therefore, knowledge of the scope of a tool is useful to assess if it solves a problem or fills a specific gap in the literature. Example values consist of *Smart Cities*, *Home Automation*, *Education*, *Industry* or *Several* if there is more than one.
2. **Architecture.** Visual programming tools applied to the Internet-of-Things can have a centralized or decentralized architecture, based on their use of Cloud, Fog or Edge Computing architecture. Possible values are *Centralized*, *Decentralized* and *Mixed*.

3. **License.** The license of software or tool is essential in terms of its usability. Normally, an open-source software reaches a bigger user base and allows them to expand and contribute to it. Possible values are the name of the tool license or N/A if it does not have one.
4. **Tier.** IoT systems, as explained in Section 2.1.1 (p. 6) is composed of three tiers - *Cloud*, *Fog* and *Edge*. A tool can interact in several of these tiers, which shapes the features it contains and how it is built.
5. **Scalability.** Defines how the tool or framework scales. It can be calculated based on metrics used to test the performance of the system. In this case, we considered scalability in terms of number and different types of devices supported. Possible values are *low*, *medium*, *high* or N/A, in case there is no sufficient information.
6. **Programming.** According to Downes and Boshernitsan [14] and also mentioned in Section 2.2 (p. 8), visual programming languages can be classified in five categories: (1) Purely Visual languages, (2) Hybrid text and visual systems, (3) Programming-by-example systems, (4) Constraint-oriented systems and (5) Form-based systems. These classifications are not mutually exclusive. It is important to know which type, so that might be possible to assess the type of experience the tool provides to the user and its architecture.
7. **Web-based.** Defines if the visual programming language and/or environment can be used in a browser. It is useful in terms of the usability of the tool.

Table 3.2: Visual programming solutions applied to IoT and their characteristics. Small circles (●) mean *yes*, hyphens (-) means *no information available*, empty means *no* and asterisk (*) means more than one.

Tool	Scope	Architecture	License	Tier	Scalability	Programming	Web-based
Belsa et al. [10]	Several	Centralized	-	Cloud	High	Hybrid	●
Ivy [26]	Several	Centralized	-	Cloud	Medium ⁷	Purely visual	
Ghiani et al. [33]	Home Automation	Centralized	-	Cloud	-	Form-based	●
ViSiT [4]	Several	Centralized	-	Cloud	High	Hybrids	●
Valsamakis and Savidis [69]	Ambient Assisted Living	Centralized	-	Cloud	-	Hybrid	●
WireMe [52]	Education, Home Automation	Centralized	-	Cloud	-	Hybrid	
VIPLE [23]	Education	Centralized	-	Cloud	-	Hybrid	
Smart Block [9]	Home Automation	Centralized	-	Cloud	-	Hybrid	●
PWCT [31]	Several	Centralized	GNU GPL v2.0	- ¹	High	Hybrid	
DDF [36]	-	Decentralized	Apache 2.0	Fog	High	Hybrid	●
GIMLE [67]	Industry	Centralized	-	Cloud	High	Hybrid	●
DDFlow [50]	Security	Decentralized	-	Fog and Edge	-	Hybrid	●
Kefalakis et al. [41]	-	Centralized	LGPL V3.0 ³	Cloud	-	Hybrid	
Eterovic et al. [29]	Home Automation	- ⁴	-	-	-	Hybrid	-
FRED [13]	Several	Centralized	- ⁵	Cloud	High	Hybrid	●
WoTFlow [12]	-	Decentralized	-	Fog and Edge	-	Hybrid	●
Besari et al. [11] [59]	Education	Centralized	-	Cloud	-	Hybrid	
CharIoT [66]	Home Automation	Centralized ⁶	-	Cloud and Edge ⁶	High ⁶	Form-based	●
Desolda et al. [24]	Smart Museums	-	-	-	-	Hybrid	
Eun et al. [30]	Home Automation	Centralized	-	-	-	Form-based	●

¹ Used for several purposes, did not specify the tier it is located in regarding IoT.

² Since it uses Node-RED, this information was based on its architecture.

³ Under the same license of OpenIoT.

⁴ No information is given regarding the architecture of the environment created, only the VPL.

⁵ No information about the license is given, but further research discovered that it had paid plans and no source code available.

⁶ CharIoT uses the Giotto stack [3] from where we retrieved this information.

⁷ Certainty regarding this information is low.

3.1.3 Expanded Search

The results of the Systematic Literature Review were disclosed in the previous section. However, some tools were found in a non-systematic survey [56] that are not present in the selected papers. We consider that this divergence may result from tools that have no academic publications associated with them, thus not being present in the publication databases mentioned in Section 3.1.1.2 (p. 12). One famous example is *Node-RED* [32]. The results from the survey [56] were analyzed, the described tools were assessed against the evaluation process defined in Section 3.1.1 (p. 11) and characterized by the categories mentioned in Section 3.1.2 (p. 14). Using the methodology described, the results are:

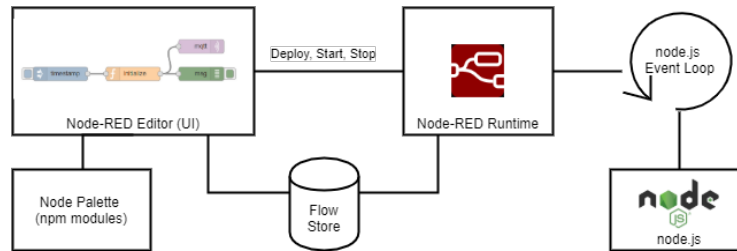


Figure 3.4: Node-RED [32] high-level architecture, identifying its development interface, runtime and `node.js` dependency. The *flows* can be versioned and organized in projects and new modules (*i.e.*, *nodes*) can be added using the `node.js` dependency manager tool (*i.e.*, `npm`).

- **Node-RED** [32] is a visual programming environment applied to the IoT paradigm. It makes use of flow-based development (connecting communication and computation *nodes* in *flows*), supporting a wide range of devices and APIs. It has two main modules: (1) a development interface which consists of a flow drawing canvas and a *node* palette, and (2) a runtime module that leverages the Node.JS event-loop to pass messages between the different *nodes* (*cf.* Figure 3.4). Due to being open-source and extendable, its large community contributes with features that enrich the tool, some of them talked about in Section 3.1.2 (p. 14) (*e.g.*, FRED [13] and DDF [36]).
- **NETLab Toolkit** [68] is a visual environment that makes use of *drag-and-drop* actions to allow users to build IoT applications. It provides a web interface to connect sensors, actuators, and others for quick prototypes.
- **NooDL** [49] is a platform that provides a visual programming interface for prototyping applications. It allows for the creation of interfaces, using live data, and supporting several types of hardware. Although it is not specific to IoT, NooDL covers the programming of IoT systems. It makes use of MQTT broker agents for connecting devices and visual paradigms such as *nodes*, *connections*, and *hierarchies* to allow the user to build its system.
- **DGLux5** [25] for DSA is a *drag-and-drop* visual language and environment that allows its users to build applications tailored for Distributed Services Architecture (DSA) IoT

middleware. It provides a dashboard for analyzing and controlling device data in real-time and builds the system only using visual elements.

- **AT&T Flow Designer** [8] is a visual tool incorporated in a cloud development environment, applied to the development of IoT systems. Its visual paradigm is similar to Node-RED, with the notion of *nodes* and *wires*. This tool provides an easy iteration and improvement of a product, as well as an easy deployment.
- **GraspIO** [45] is a Graphical Smart Program for Inputs and Outputs that contains a block *drag-and-drop* visual paradigm that allows its users to build applications for the *Cloudio* hardware. It offers a Cloud Service that connects and manages all *Cloudio* devices, making them available at the user's mobile device.
- **Wylodrin** [70] is a browser-based visual programming environment that allows the development of IoT systems of several devices, such as Raspberry Pi, Arduino, Intel Galileo, Intel Edison, and others. It provides a *drag-and-drop* environment, as well as support for text-based languages. A dashboard for visualizing the data collected is provided.
- **Zenodys** [18] provides a *drag-and-drop* interface to build application backends as well as user interfaces. Its computing engine can run in several types of devices, from the cloud to chips, devices, and distributed computers. Zenodys contains a visual debugger as well as support for text-based programming and code generation.

Tool	Scope	Architecture	License	Tier	Scalability	Programming	Web-based
Node-Red [32]	Several	Centralized	Apache 2.0	Cloud and Edge	High	Hybrid text and visual system	•
NETLab Toolkit [68]	-	-	GNU GPL	Edge ²	-	Hybrid text and visual system	•
NoodDL [49]	Several	-	NoodDL End User License ¹	Cloud ²	-	Hybrid text and visual system	
DGLux5 [49]	Several	-	DGLux Engineering License	Cloud and Fog ²	High ²	Purely visual language	
AT&T Flow Designer [8]	Several	-	GNU GPL3	Cloud ²	High ²	Hybrid text and visual system	•
GraspIO [45]	Education	-	BSD	Cloud ²	-	Purely visual language	
Wylodrin [70]	Several	-	GNU GPL3	All ²	-	Hybrid text and visual system	•
Zenodys [18]	Several	-	GNU GPL3	Cloud ²	High ²	Hybrid text and visual system	•

Table 3.3: Characterization of VPLs applied to IoT from survey [56]. Small circles (•) mean *yes*, hyphens (-) means *no information available*, empty means *no* and asterisk (*) means more than one.

¹ Available at <https://www.noodl.net/eula>

² Certainty regarding this information is low.

3.1.4 Analysis and Discussion

The tools presented in this Systematic Literature Review passed the evaluation process defined in Section 3.1.1.3 (p. 13). Tools that only supported one device were left out, as well as tools that extended a VPL applied to IoT.

3.1.4.1 Evolution Analysis

To understand the evolution of visual programming languages applied to IoT, the publication years of the tools found in Section 3.1.2 (p. 14), as well as the launch years of the survey tools of Section 3.1.3 (p. 22), were analyzed. Figure 3.5 contains the their evolution, where we can observe that there was a more substantial amount of work related to this topic in the years 2017 and 2018. The year 2019 still does not have conclusive data.

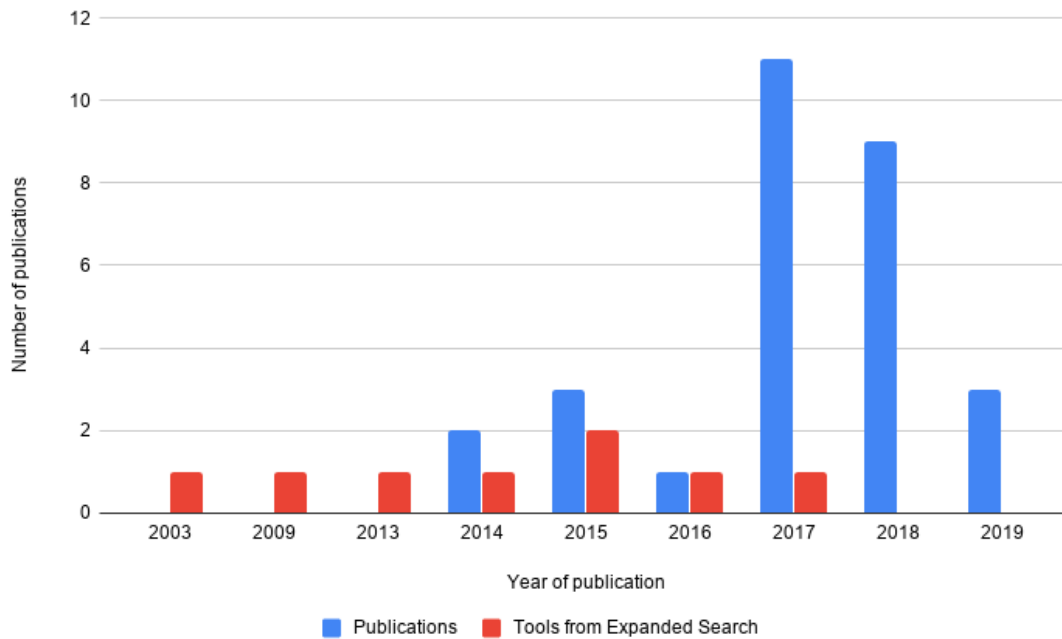


Figure 3.5: Publications and tools of VPL tools applied to IoT per year.

3.1.4.2 Result Analysis

Scope Most of the tools found have several scopes, such as education, industry or home automation. From the 28 tools, six were specific to home automation, 4 to education, 3 to specific domains, and 1 for the industry; the remainder 14 had a wide range of use cases.

Architecture From the 28 tools found, 16 tools have a centralized architecture, three are decentralized, and the remaining nine do not present enough information to conclude on this topic.

License Most of the tools did not mention a license and the ones who did were in its majority open-source (*e.g.*, GNU GPL2, GNU GPL3, Apache 2.0 and LGPL3).

Scalability The majority of tools analyzed do not have scalability metrics analyzed, more specifically, the number of devices supported by them. The ones that do have high scalability, which seems to indicate that scalability is only analyzed when the tool has support for it.

Programming From the 28 analyzed tools, 22 employ a hybrid text and visual system visual programming paradigm, while 3 use a purely visual and the other three a form-based one.

Web-based The majority of tools analyzed are web-based, being accessible with the use of a browser. Only one tool did not provide an environment, only a specification of a visual programming language.

3.1.4.3 Research Questions

The research questions presented in Section 3.1.1.1 (p. 12) served as a way of directing the research of this Systematic Literature Review and obtain answers to relevant questions regarding the available tools that apply visual programming languages to the IoT domain. These answers are:

RQ1 *What relevant visual programming solutions applied to IoT orchestration exist?* From the analyzed tools in Sections 3.1.2 and 3.1.3, we found 28 visual programming tools applied to IoT orchestration.

RQ2 *What is the tier and architecture of the tools found in RQ1?* Tables 3.2 and 3.3 give an overview of the characteristics of all the tools found. In these tables and subsequent analysis in Section 3.1.4.2 (p. 24) it is concluded that the majority of the tools have a centralized architecture and work in the Cloud tier.

RQ3 *What was the evolution of visual programming solutions applied to IoT orchestration over the years?* As it can be observed in Section 3.1.4.1 (p. 24) and more specifically in Figure 3.5 (p. 24), there are visual programming tools applied to the orchestration of IoT since 2003, and in 2017 and 2018 there was a bigger number of publications with a focus on building these type of tools.

3.1.5 Conclusions

In this Systematic Literature Review, 2698 publications were analyzed from IEEE, ACM and Scopus databases, resulting in 20 visual programming tools applied to the Internet-of-Things. A survey made on the visual programming solutions applied to IoT found during the research process resulted in 8 more tools, making a total of 28.

The results show that there is a significant number of tools that allow end-users to build IoT systems using visual programming in several different scopes. The majority of these tools have a centralized architecture and operate in the Cloud tier. Despite the considerable amount of tools, most of them do not have their source code accessible nor have a license. The results from the expanded search are more positive in this aspect, with the majority of them being open-source, such as Node-RED [32], NETLab Toolkit [68] and others. However, this poses a problem since there is an evident lack of open source tools.

In summary, the majority of tools found do not possess a license, employ a centralized architecture, operate in the Cloud tier and use a hybrid text and visual programming system. Thus, it

propels the possibility of future research on designing and building a visual programming tool applied to IoT that is (1) open-source, (2) has a decentralized architecture and (3) also operates in the Fog and/or Edge tiers.

3.2 Decentralized Architectures in Visual Programming Tools applied to the Internet-of-Things paradigm

Although the substantial amount of solutions found during our systematic literature (Section 3.1.2), only a small fraction of those aim to offer a truly decentralised solution to visual orchestration for Internet-of-Things systems. These solutions are now analysed in detail, followed by a comparison and discussion.

3.2.1 DDF

The work made by in WoTFlow [12], DDF [36] and subsequent works [34, 35] consists of a system built on the Node-RED framework and focused on the use case of Smart Cities. Their goal is to make a tool more suitable for the development of fog-based applications that are dependent on the context of the edge devices where they operate.

In DDF [36], the authors started by extending Node-RED and implementing D-NR (Distributed Node-RED), which contains processes that can run across devices in local networks and servers in the Cloud. The application, called flow, is built in the visual programming environment, which is running in a development server. All the other devices running D-NR subscribe to an MQTT topic that contains the status of the flow. When a flow is deployed, all devices running D-NR are notified and subsequently analyse the given flow. Based on a set of constraints, they decide which nodes they may need to deploy locally and which sub-flow (parts of a flow) must be shared with other devices. Each device has a set of characteristics, from its computational resources such as bandwidth, available storage to its location. The developer can insert constraints into the flow, by specifying which device a sub-flow must be deployed in or the computational resources needed. Further, each device must be inserted manually into the system by a technician.

Subsequent work to the previously mentioned tool focused on support for the Smart Cities domain. In a 2018 publication [34], the problems addressed were the deployment of multiple instances of devices running the same sub-flow, as well as the support for more complex deployment constraints of the application flow. With this, the developer can specify requirements for each node on device identification, computing resources needed (CPU and memory) and physical location. In addition to these improvements, the coordination between nodes in the fog was tackled by introducing a coordinator node. This node is responsible for synchronising the context of the device with the one given by the centralised coordinator. In Figure 3.6 (p. 27) it is possible to see the four possible states of a coordinator node: (1) NORMAL, where the node passes the data to its output, (2) DROP, in which the node does not pass the data to other node and instead drops it, (3) FETCH_FORWARD, where the node gets the input from an external instance of its supposed

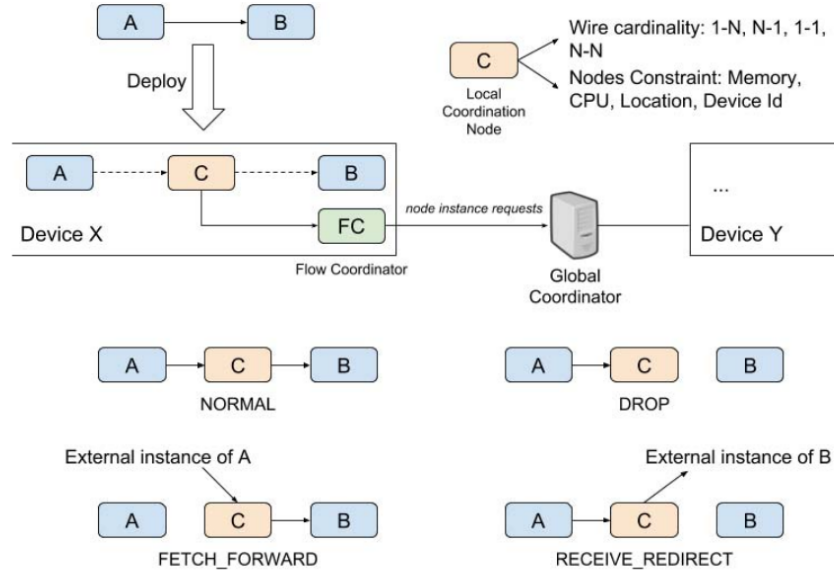


Figure 3.6: Coordination between nodes in D-NR[34].

input and (4) RECEIVE_REDIRECT in which the node sends the data to an external instance of its output node.

In more recent work [35], support for CPSCN (Cyber-Physical Social Computing and Networking) was implemented, making it possible to facilitate the development of large scale CPSCN applications. Additionally, to make this possible, the contextual data and application data were separated, so that the application data is only used for computation activities and the contextual data is used to coordinate the communication between those activities.

3.2.2 FogFlow & uFlow

Another approach was made in the publication by Szydło et al. [65], where they focused on the transformation and decomposition of data flow. Parts of the flow can be translated into the executable parts, such as Lua code. Their contribution includes the concepts of data flow transformation, a new run-time environment called *uFlow* that can be executed on a variety of resource-constrained embedded devices and the integration with the Node-RED platform.

The solution consisted of the transformation of a given data flow, where the developer chooses the computing operations that will be run on the devices. The operations that will run directly on the devices are implemented in the form of embedded software, using the developed framework *uFlow*, which allows parts of the flow to be run on heterogeneous devices. All this is integrated with Node-RED. The communication between the devices is made only through the Cloud, with no support for peer-to-device communication. The results were promising, with a decrease in the number of measurements made by the sensors. However, there was room for improvement, with the

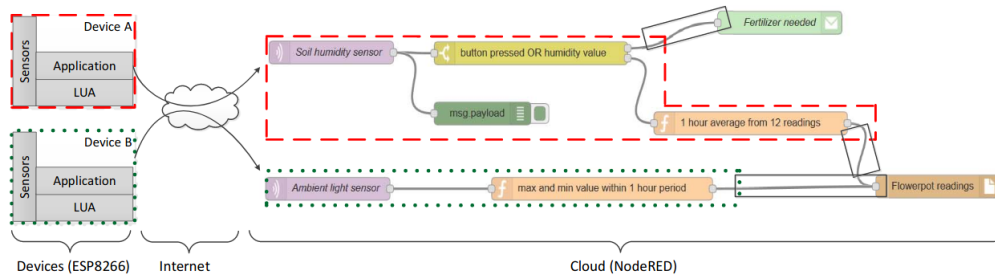


Figure 3.7: Partition and assignment of parts of the flow[65].

automation of the decomposition and partitioning of the initial flow. Another improvement would be the detection bottlenecks which will move computations accordingly from the cloud to the fog.

Figure 3.7 represents a situation of partitioning and assignment of tasks. There are two IoT devices and a Node-RED instance running in the Cloud. The system's goal is to measure soil humidity and ambient light. If a button is pressed or fertiliser is needed, an e-mail is sent to the gardener. The partition of computation is made with the assumption that the closer a selected process is to the source of data, the higher the amount of data transmitted between computing operations. After parts of the flow are assigned to specific devices, they are altered to be executed by *uFlow* and Node-RED. It is possible to observe in Figure 3.7 the results of the transformation process, where the parts of the flow surrounded by colour are executed in the device having the same colour.

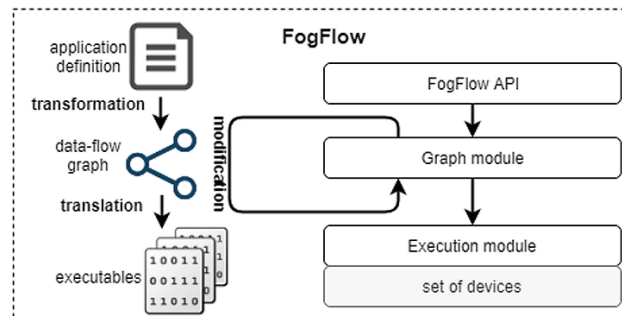


Figure 3.8: *FogFlow* architecture[58].

In a new publication [58], they built the model and engine *FogFlow*, which enables the design of applications able to be decomposed onto heterogeneous IoT environments according to a chosen decomposition schema. To achieve a level of decentralisation and heterogeneity, they abstract out the application definition from its architecture and rely on graph representation to provide an unambiguous, well-defined model of computations. The application definition should be infrastructure-independent and contain only data processing logic, and its execution should be possible on different sets of devices with different capabilities. Several algorithms for flow decomposition are mentioned [48, 38], but none were specified in terms of results. Figure 3.8

represents the *FogFlow* architecture, which is composed by three modules: (1) the *FogFlow* API, which enables the creation of the application definition, (2) the Graph Module, responsible for processing and transforming the application definition into a data flow graph and finally the (3) Execution Model, which translates the graph and generates executables ready to be run on the assigned devices.

3.2.3 *FogFlow*

There is another tool with the same name *FogFlow* but created by Cheng et al. [63]. In the first publication related to this tool [22], the contributions made were the implementation of a standards-based programming model for Fog Computing and scalable context management. The first contribution consists in extending the dataflow programming model with hints to facilitate the development of fog applications. The scalable context management introduces a distributed approach, which allows overcoming the limits in a centralised context, achieving much better performance in terms of throughput, response time and scalability. The *FogFlow* framework focuses in a Smart City Platform use case, separated in three areas: (1) Service Management, typically hosted in the Cloud, (2) Data Processing, present in cloud and edge devices and (3) Context Management, which is separated in a device discovery unit hosted in the Cloud and IoT brokers scattered in Edge and Cloud.

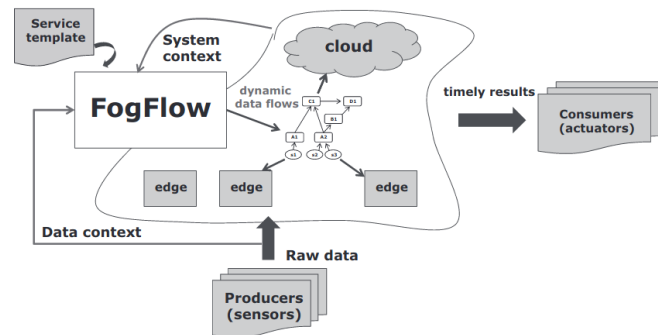


Figure 3.9: *FogFlow* high level model[21].

In more recent work [21], *FogFlow* was improved to deliver infrastructure providers with an environment that allows them to build decentralised IoT systems faster, with increased stability and scalability. The architecture can be seen in Figure 3.9, where dynamic data representing the IoT system flows that are orchestrated between sensors (Producers) and actuators (Consumers). The application is first designed using the *FogFlow* Task Designer, a hybrid text and visual programming environment, which results in an abstraction called Service Template. This abstraction contains specifics about the resources needed for each part of the system. Once the Service Template is submitted, the framework will determine how to instantiate it using the context data available. Each task is associated with an operator (a Docker image), and its assignment is based on (1) how many resources are available on each edge node, (2) the location of data sources, and (3) the prediction

of workload. Edge nodes are autonomous since they can make their own decisions based on their local context, without relying on the central Cloud.

3.2.4 DDFlow

DDFlow [50], first mentioned in Section 3.1.2 (p. 14), presents another distributed approach by extending Node-RED with a system run-time that supports dynamic scaling and adaption of application deployments. The coordinator of the distributed system maintains the state and assigns tasks to available devices, minimizing end-to-end latency. Dataflow notions of *node* and *wire* are expanded, with a *node* in DDFlow representing an instantiation of a task that is deployed in a device, receiving inputs and generating outputs. *Nodes* can be constrained in their assignment by optional parameters, *Device*, and *Region*, inserted by the developer. A *wire* connects two or more nodes and can have three types: *Stream* (one-to-one), *Broadcast* (one-to-many) and *Unite* (many-to-one).

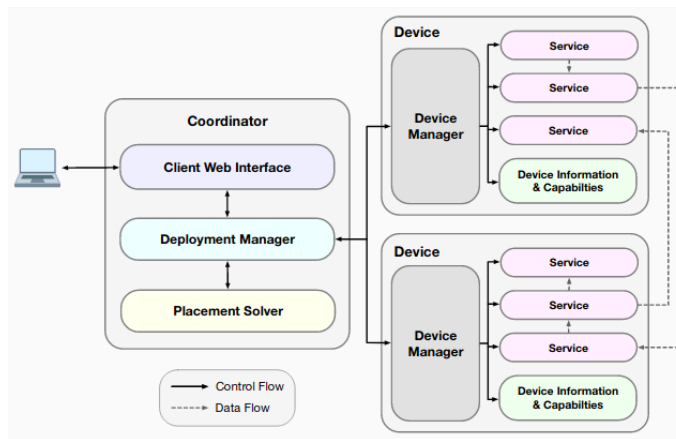


Figure 3.10: DDFlow architecture [50].

In a DDFlow system, each device has a set of capabilities and a list of services that correspond to an implementation of a *Node* (cf. Figure 3.10). The devices communicate this information through their Device Manager or a proxy if it is a constrained device. The coordinator is a web server responsible for managing the DDFlow applications and is composed of three parts, which can be seen in Figure 3.10: (1) a visual programming environment where DDFlow applications are built, (2) a Deployment Manager that communicates with the Device Managers of the devices and (3) a Placement Solver, responsible for decomposing and assigning tasks to the available devices. When an application is deployed, a network topology graph and a task graph are constructed based on the real-time information retrieved from the devices. The coordinator proceeds with mapping tasks to devices by minimizing the task graph's end-to-end latency of the longest path. Dynamic adaptation is supported by monitoring the system and adapting to changes. If changes in the network are detected, such as the failure or disconnection of a device, adjustments in the assignment of tasks are made. In addition to this, the coordinator can be replicated onto many devices to improve the reliability and fault-tolerance of the system.

In the evaluation made to DDFlow, the system can recover from network degradation or device overload, whereas in a centralised system, this would cause its total failure.

3.2.5 Analysis

The mentioned tools were characterized based on their mentions or support for the following features and characteristics:

1. **Leverage devices.** A decentralised architecture takes advantage of the computational power of the devices in the network, assigning them tasks. However, some tools can have limitations on the type of devices, making constraints or only focusing on the devices of the Fog tier and not Edge.
2. **Capabilities communication.** The devices need to communicate to the orchestrator their capabilities so that it can make an informed decision regarding the decomposition and assignment of tasks.
3. **Open-source.** The license of software or tool is essential in terms of its usability. Open-source allows access to the code, making it possible for its analysis, improvement, and reuse.
4. **Computation decomposition.** To implement a decentralised architecture, it is important to decompose the computation of the system into independent and logical tasks that can be assigned to devices. This is made using algorithms, which can be specified or mentioned.
5. **Run-time adaptation.** A system needs to adapt to run-time changes, such as non-availability of devices or even network failure. The system notices these events and can take action to circumvent the problems and keep functioning.

Table 3.4: Small circles (●) mean *yes*, hyphens (-) means *no information available*, empty means *no* and asterisk (*) means more than one.

Tool	Leverage devices	Capabilities communication	Open-source	Computation decomposition	Run-time adaptation
DDF [36, 12, 34, 35]	Limited ¹	●	●	Limited ²	●
FogFlow & uFlow [65, 58]	●	Limited ³		Limited ²	Limited ³
FogFlow [63, 22, 21]	●	-	●	Limited ²	●
DDFlow [50]	Limited ⁴	●		Limited ²	●

¹ Assumes that all devices run Node-RED, which limits the type of devices.

² Do not specify the algorithm used.

³ Communication between devices is made through the Cloud.

⁴ Assumes that all devices have a list of specific services they can provide.

From the analysis and the characteristics of Table 3.4, we can conclude that the current research in decentralised architectures in visual programming tools applied to IoT is incomplete. All the tools leverage the devices in the network but in different ways. DFF [36] assumes that all devices run Node-RED, which limits the type of devices that can be leveraged since it needs to

have minimum resources to run it. *FogFlow* and *uFlow* [58, 65] is the only tool that specifies how it truly leverages constrained devices, with the transformation of sub-flows into Lua code, with DDFlow [50] assuming that all devices have a list of specific services they can provide, that should match the node assigned to them.

Regarding the method used to decompose and assign computations to the available devices, DDFlow describes the process with the use of the longest path algorithm focused on reducing end-to-end latency between devices. *FogFlow* and *uFlow* [58, 65] mention several algorithms that could be used, but do not specify which one was implemented. Both DDF [36] and *FogFlow* [22, 21] do not specify the algorithm used besides some constraints but are the only tools with their source code accessible and with an open-source license. All the tools claim to have support for run-time adaptation to changes in the system, such as device failures.

3.2.6 Conclusion

Several solutions are available that provide a decentralized architecture in visual programming tools applied to the Internet-of-Things paradigm. However, some of these tools solve specific problems or make assumptions regarding the scale of the system and the constraints of the devices. We can highlight the following research challenges that remain to be addressed by the research community:

1. **Leveraging devices in the network:** since most tools use a centralized architecture, including Node-RED, they do not leverage the devices in the network. Fog Computing introduces a decentralized solution, one that can be applied to Node-RED by distributing the computational tasks across the edge devices.
2. **Communication of computational capabilities:** some of the current tools require the developer to manually introduce the resources of each device in the network, which is not a scalable solution. Others have a specific list of services, manually inserted that the devices can provide. Information about the computational capabilities of the devices in the network is vital for the successful distribution of computation across the devices.
3. **Detecting unavailability:** when a device fails or becomes unavailable, the system needs to realize and adapt automatically. The majority of current solutions do not possess this feature, which is vital if a system aims to adapt to changes in the environment dynamically.
4. **Code generation of sub-flows:** to truly leverage constrained devices, it is important to convert sub-flows or "tasks" into executable code. Devices that support simple firmware capable of executing code can be used to execute blocks of code, despite their limited capabilities.
5. **Provide self-adaption of the system:** devices can fail, as well as the connection between them or even the network. The system needs to discover and identify these changes and adapt to them at run-time in order to keep functioning.

Addressing these research challenges can improve the current state of the Internet-of-Things ecosystem, where systems are mostly centralized, leading to the proliferation of single point of failure (SPOF) (*e.g.*, dependency on the Cloud can render systems unusable if there is an Internet-connectivity problem). Further, high amounts of computational power are unused (*e.g.*, response times could be improved by using on-premises and already existent resources), and can address some pending security and privacy issues [55].

We should also mention that although Partha's survey [56] conclude that there exists some advantages of using visual programming languages, such as the ease of visualizing programming logic (which useful for rapid prototyping), and less burden on handling syntax error, as well as mentioning some negative aspects, such as the significant amount of time required for building simple IoT applications, we cannot find support or a rationale on how this observation results from the attributes they analysis (*e.g.*, licence and project repository).

3.3 Summary

Section 3.1 (p. 11) presents a Systematic Literature Review of visual programming tools applied to the Internet-of-Things. Each tool derived from the research is summarized and characterized to understand the state of the art regarding this topic of interest. Section 3.2 (p. 26) describes visual programming tools for building IoT systems that employ a decentralized architecture, pointing out their advantages but also their shortcomings.

Chapter 4

Problem Statement

4.1	Current Issues	35
4.2	Desiderata	36
4.3	Scope	36
4.4	Main Hypothesis	37
4.5	Experimental Methodology	37
4.6	Summary	37

This chapter describes the problem, as can be seen in Section 4.1. In Section 4.2 it is presented the wanted features for the proposed solution and in Section 4.3 the scope of the project is defined. Section 4.4 contains the hypothesis this dissertation presents. The experimental methodology is outlined in Section 4.5. Finally, this chapter is summarized in Section 4.6 with an overview of the topics mentioned before.

4.1 Current Issues

Chapter 3 (p. 11) contains several solutions that provide a decentralized architecture in visual programming tools applied to the Internet-of-Things paradigm. However, some of these tools solve specific problems or make assumptions regarding the scale of the system and the constraints of the devices. We can define the problem in these issues:

1. **Leveraging devices in the network:** since most tools use a centralized architecture, including Node-RED, they do not leverage the devices in the network. Fog Computing introduces a decentralized solution, one that can be applied to Node-RED by distributing the computational tasks across the edge devices.
2. **Communication of computational capabilities:** some of the current tools require the developer to manually introduce the resources of each device in the network, which is

not a scalable solution. Others have a specific list of services, manually inserted, that the devices can provide. Information about the computational capabilities of the devices in the network is vital for the successful distribution of computation across the devices.

3. **Detecting non-availability:** when a device fails or becomes unavailable, it is important for the system to automatically realize and adapt. The majority of current solutions do not possess this feature, which is vital if a system aims to dynamically adapt to changes in the environment.
4. **Code generation of sub-flows:** to truly leverage constrained devices, it is important to convert sub-flows or "tasks" into runnable code. Devices that support simple firmware capable of executing code can be used to execute blocks of code, despite their limited capabilities.
5. **Provide self-adaption of the system:** devices can fail, as well as the connection between them or even the network. It is important for the system to discover and identify these changes and adapt to them at run-time, in order to keep functioning.

4.2 Desiderata

Desiderata is a Latin word that translates to "*things wanted*". In the context of this document, this section contains requirements wanted in a solution that aims to solve all the issues identified in Section 4.1 (p. 35). The requirements are the following:

- D1: Communicate computational capabilities of devices connected** so that this information can be sent to an orchestrator that will decompose the total computation workload based on this data.
- D2: Decomposition and partition of computation** so that the total computational requested can be distributed through all the devices in the network, using information about the computational capabilities and availability of the devices in the network.
- D3: Convert computational tasks into runnable code** so that each computational task can be executed in edge and fog devices, which contain limited resources.
- D4: Provide self-adaptation of the system** so that it can adapt to the non-availability of resources or even appearances of new devices.

4.3 Scope

The focus of this dissertation is the development of a prototype that allows for a decentralized orchestration of an IoT system. Despite security being a critical feature, it is considered a secondary goal, allowing the dissertation to focus on its primary goals. The scope of the project is a

home automation system, where its devices have firmware capable of running MicroPython code and accepting custom code. They also need to be able to communicate their capabilities to an orchestrator.

4.4 Main Hypothesis

This dissertation is built around the following hypothesis:

“Given an IoT system with several heterogeneous devices connected, capable of running custom code, a decentralized architecture is more resilient, efficient and scalable than a centralized one.”

The attributes presented in the hypothesis will be measure against a system using the current development branch of Node-RED. These attributes consist of:

- **Resilience** means the system’s capability to adapt to failures and changes. It will be measured by injecting failures and measuring the recovery patterns.
- **Efficiency** how fast the system can execute the logic of the system and communicate between nodes. The efficiency of a system is measured by the latency when reacting to system events.
- **Elasticity** specifies how a system can grow and shrink. This attribute will be tested by increasingly adding or removing devices in different scenarios and assessing the overall system’s behavior.

4.5 Experimental Methodology

In the interest of validating whether or not the solution implemented achieves the *desiderata* and solves the current issues, we will develop test scenarios and experiments that use both virtual and physical devices. Each one of the test scenarios will measure the one or more requirements proposed in Section 4.2. Lastly, the attributes mentioned in Section 4.4 will be evaluated with the implement solution, in order to validate the given hypothesis.

4.6 Summary

Section 4.1 (p. 35) starts by presenting issues and lack of features not fully present in the current tools presented in the State of the Art Chapter 3 (p. 11). Section 4.2 (p. 36) presents a *desiderata* that aims to fix the issues presented in Section 4.1 (p. 35). The hypothesis of this dissertation is presented in Section 4.4, as well as an experimental methodology to prove it, in Section 4.5.

Chapter 5

Solution

5.1 Overview	39
5.2 Implementation Details	40

This chapter describes how the problem presented in Chapter 4 was solved by stating the solution implemented and the reasons for the choices taken. Section 5.1 provides an overview of the developed solution, which is detailed in Section 5.2. Finally, in Section 5.2.3, a summary of the developed solution is presented, which mentions some limitations of the developed solution and problems that is able to solve.

5.1 Overview

In our solution, we use Node-RED for both (1) defining programs (as flows) and (2) orchestrate the decentralization and send tasks to other devices in the network, acting as an orchestration controller. The devices in the network make themselves known by announcing their address and capabilities to a registry *node* running in Node-RED. Consequently, Node-RED assigns *nodes* to devices taking into account their capabilities and communicates each node's assignment via HTTP. Due to the devices' limitations, they cannot run an instance of Node-RED, so Node-RED needs to translate the *nodes* code in JavaScript to another language that can be interpreted by these devices.

Node-RED was modified to meet the distributed computation communication demands by replacing the built-in communication by an MQTT-based one. Two main components, as *nodes*, were introduced to the Node-RED Palette: (1) the *Registry node* which maintains a list of available devices and their capabilities and, (2) the *Orchestrator node* which partitions and assigns computation tasks to the available devices. Additionally, support was added to Node-RED to generate MicroPython-compatible code from custom *nodes*, *i.e.*, code-to-code transformation.

Additionally, a MicroPython-based firmware was developed that can receive and run arbitrary Python code scripts generated by Node-RED, and communicate with other devices or Node-RED itself using MQTT.

An high-level overview of the system can be seen in Figure 5.1. Each module will be analyzed in detail in the following sections.

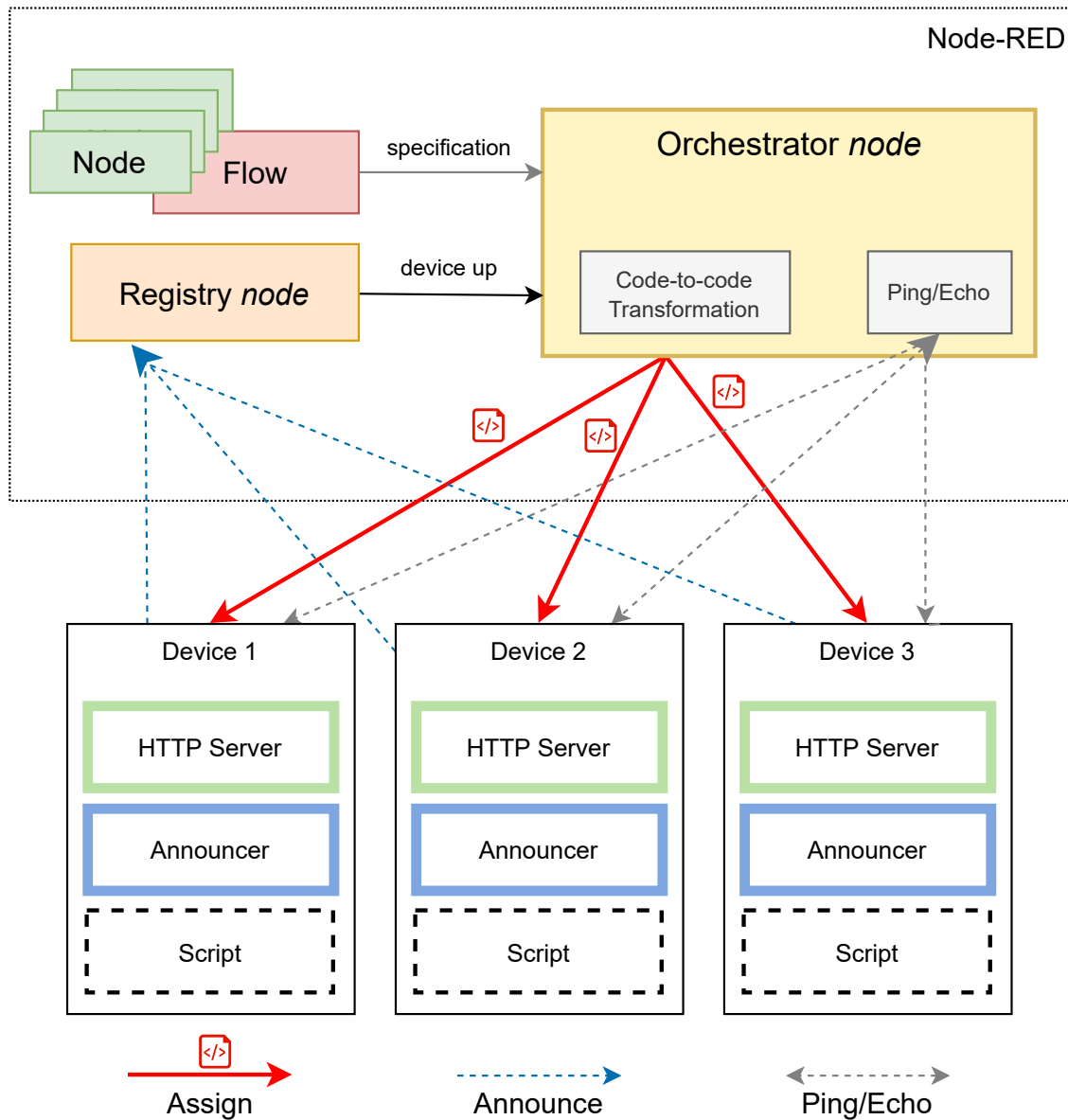


Figure 5.1: Solution's overview, presenting three devices as orchestration targets.

5.2 Implementation Details

The implemented solution consists of two co-dependent modules that are necessary for the functionality of the system. The first module consists of the changes implemented to Node-RED run-time

Table 5.1: Comparison between the Espressif Systems ESP32 and ESP8266 systems on chip.

	ESP8266	ESP32
MCU	Single-core 32bit	Dual-Core 32bit
Frequency	80 MHz	160 MHz
SRAM	160kBytes	512kBytes
Flash	SPI, 4MBytes	SPI, 4MBytes
802.11 b/g/n WiFi	Yes	Yes
Bluetooth	No	Yes
Programming Lang.	Lua, Python and C	

to allow for its decentralization, detailed in Section 5.2.2 (p. 42). The second module consists of the solution found to take advantage of external devices with limited capabilities, explained in Section 5.2.1.

5.2.1 Devices Setup for Decentralization Support

For this work, we considered constrained devices that are capable of running custom code. Amongst the available hardware solutions, taking into consideration both costs and features, we picked two IoT development devices based on the Espressif Systems ESP32 and ESP8266 systems on chip (SoC) [28, 27]. An overview of these devices hardware capabilities is given on Table 5.1.

The first challenge was to find a way to take advantage of the constrained devices by making them run arbitrary scripts of code and communicate with other devices. Since both selected devices can run MicroPython firmware, Python was the selected programming language. Further, MicroPython already packs a small-footprint HTTP server and packages are available to implement asynchronous operations — *i.e.*, `uasyncio` — and MQTT publisher-subscriber (*viz.*, pub-sub) communication — *i.e.*, `MicroPython-mqtt`.

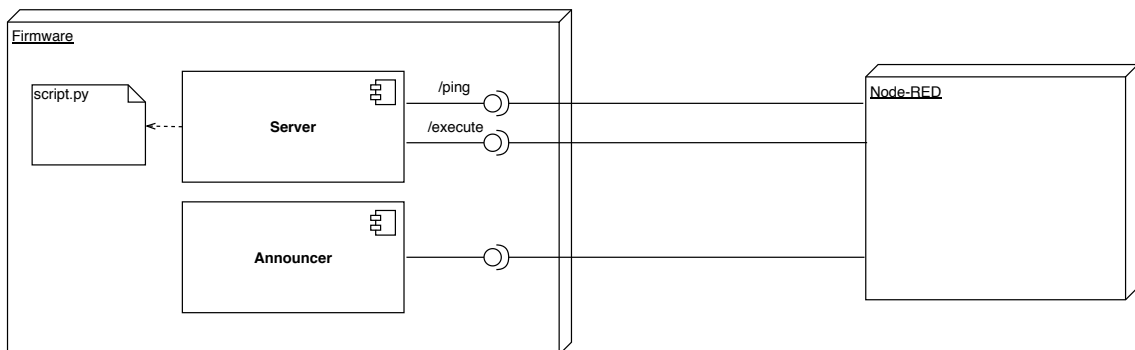


Figure 5.2: Firmware component diagram.

As the devices must be able to receive arbitrary Python scripts (sent by Node-RED) and run them, the HTTP server was used to receive the Python payloads, which are then saved in the device SPI Flash and can be later executed (loaded to the SRAM). Further, the same HTTP server was used

to implement an endpoint that returns the state of the device, as well as an announcing mechanism (cf. Section 5.2.2.4 (p. 47)). These features built as an integral part of the firmware that runs on the devices. An overview of the components of the firmware can be seen in Figure 5.2 (p. 41).

The firmware also includes a FAIL-SAFE mechanism, safeguarding against *Out-of-Memory* errors that may happen during the lifespan of the device (SRAM usage). This mechanism resets all running tasks and recovers the HTTP server and communication channels. This is an important feature due to the high probability of these error's occurrence since the devices have limited memory.

However, this solution is not without some limitations from which we highlight: (1) the developed firmware only supports MQTT QoS levels 0 and 1 due to `MicroPython-mqtt` limitations, and (2) ESP8266 severe memory limitations prohibit the use of the FAIL-SAFE mechanism if the given script is too big.

5.2.2 Decentralized Node-RED Computation

Node-RED is a centralized tool by design, which takes advantage of events to allow communication between *nodes* in a *flow*. To implement a decentralized architecture, some changes were made to the Node-RED runtime. These changes consisted mainly in (1) implementing a new way of communication between *nodes*, (2) add code-to-code generation features (i.e., JavaScript to MicroPython), (3) . These are described in the following sections.

5.2.2.1 Node-RED Node-to-Node Communication

Node-RED *nodes* communicate using events — `node.js EventEmitter`, where a *node* only communicates with *nodes* it is wired to. The communication is one-way only (forward message passing), with the *node* only sending data to the *nodes* it is connected to by output. These output wires are used to access the *nodes* the message must be sent to, and their `receive()` method is called. This method triggers the event `emit()` which will be caught by a specific method of each *node*, implementing its own logic.

This implementation is local and JavaScript specific, making it impossible to be used in a decentralized architecture where *nodes* will be executed outside of Node-RED. It was necessary to implement a way of communicating between *nodes* external to Node-RED that could be supported by low capacity devices. The solution found was MQTT, which fits as a good solution by its low-footprint and high-popularity amongst IoT solutions [64].

Thus, the Node-RED `Node` class was modified to use MQTT pub-sub communication instead of the in-place communication. Each *node* publish messages to an unique and addressable topic generated at the flow start, to which the next *node* in the *flow* subscribes to. This happens for every *node* with the exception of *producer nodes* that only publish messages and *consumers* that only subscribe to topics.

Since the modifications were made at the class (from which every *node* derives from) all the existent *nodes* and sub-flows are compatible with this modification without any changes in their

code. However, as it will be described next, if we want a *node* to be orchestrable, the code of the *nodes* themselves needs to be changed.

5.2.2.2 Code Generation

To orchestrate Node-RED *nodes* computation amongst devices, there is the need to generate MicroPython-compatible code from the existent JavaScript (*i.e.*, code-to-code transformation). Additionally, it is necessary to support multiple *nodes* in one script (*i.e.*, condensate); thus, a generalized strategy was defined that could fit any type of *node*.

This was accomplished by adding specific code generation methods to each orchestrable *node*, which provide (1) their functionality, and (2) input/output capabilities. Since every *flow* communication is now MQTT-based, the only input and output a *node* can have is in its topics. An exception to this is in *nodes* that are producers, meaning that they generate input and do not receive it.

The code generation happens after the *Orchestrator node* defines an assignment between *nodes* and devices. This generation creates device-specific Python scripts that follow the result of the task assignment procedure (a script which can correspond to several *nodes*), adding wrapping code that ties up the script. This wrapping code is responsible for subscribing to all the input topics of all the nodes, stopping the script's processes and forwarding the MQTT messages to the respective node's code.

An example of a Node-RED flow and its respective python script can be seen in Figure 5.3 and Listing 5.1.

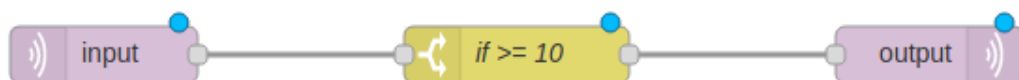


Figure 5.3: Simple Node-RED flow.

```

1 import gc
2 import sys
3 import ujson
4 import uasyncio as asyncio
5 mqtt_client = None
6 capabilities = []
7 client_id = None
8 nodes_str = "mqttin:3d70bdef542242 mqttout:40d7b1d7aca938 if:fd3cf13026958"
9 nodes_id = ["3d70bdef542242", "40d7b1d7aca938", "fd3cf13026958"]
10 input_topics = ["input", "topic1_node", "topic0_node"]
11 output_topics = ["topic0_node", "topic1_node"]
12

```

```

13 import ubinascii
14 input_topics_3d70bdef542242 = ["input"]
15 output_topics_3d70bdef542242 = ["topic0_node"]
16 node_datatype_3d70bdef542242 = "auto"
17 node_qos_3d70bdef542242 = 2
18
19 def on_input_3d70bdef542242(topic, msg, retained):
20     try:
21         if node_datatype_3d70bdef542242 == "base64":
22             msg = str(ubinascii.b2a_base64(msg))
23         elif node_datatype_3d70bdef542242 == "utf8":
24             msg = msg.encode("utf-8")
25         elif node_datatype_3d70bdef542242 == "json":
26             msg = ujson.loads(str(msg))
27         finally:
28             msg = dict(
29                 topic=topic,
30                 payload=msg,
31                 device_id=client_id
32                 qos=node_qos_3d70bdef542242,
33                 retain=retained
34             )
35             loop = asyncio.get_event_loop()
36             loop.create_task(
37                 on_output(
38                     ujson.dumps(msg),
39                     output_topics_3d70bdef542242
40                 )
41             )
42
43 import ujson
44 input_topics_40d7b1d7aca938 = ["topic1_node"]
45 output_topics_40d7b1d7aca938 = ["output"]
46
47 def on_input_40d7b1d7aca938(topic, msg, retained):
48     try:
49         msg = ujson.loads(msg)
50     else:
51         msg = dict(
52             topic=topic,
53             payload=msg["payload"],
54             device_id=client_id
55         )
56     loop = asyncio.get_event_loop()
57     loop.create_task(
58         on_output(
59             ujson.dumps(msg),
60             output_topics_40d7b1d7aca938
61         )

```



```

62     )
63
64 input_topics_fd3cf13026958 = ["topic0_node"]
65 output_topics_fd3cf13026958 = ["topic1_node"]
66 property_fd3cf13026958 = "payload"
67
68 def if_rule_fd3cf13026958_0(a, b = 10):
69     a = int(a)
70     return a >= b
71 def if_function_fd3cf13026958(a):
72     res = if_rule_fd3cf13026958_0(a)
73     return '%s' % res
74
75 def get_property_value_fd3cf13026958(msg):
76     properties = property_fd3cf13026958.split(".")
77     payload = ujson.loads(msg)
78
79     for property in properties:
80         try:
81             payload = ujson.loads(payload)
82             if payload[property]:
83                 payload = payload[property]
84             else:
85                 break
86         except:
87             break
88     return payload
89
90 def on_input_fd3cf13026958(topic, msg, retained):
91     msg = get_property_value_fd3cf13026958(msg)
92     res = if_function_fd3cf13026958(msg)
93     res = dict(
94         payload=res,
95         device_id=client_id
96     )
97     loop = asyncio.get_event_loop()
98     loop.create_task(
99         on_output(
100             ujson.dumps(res),
101             output_topics_fd3cf13026958
102         )
103     )
104     return
105
106 def on_input(topic, msg, retained):
107     topic = topic.decode()
108     if topic in input_topics_3d70bdef542242:
109         on_input_3d70bdef542242(topic, msg, retained)
110     elif topic in input_topics_40d7b1d7aca938:

```

```

111         on_input_40d7b1d7aca938(topic, msg, retained)
112     elif topic in input_topics_fd3cf13026958:
113         on_input_fd3cf13026958(topic, msg, retained)
114
115 async def conn_han(client):
116     for input_topic in input_topics:
117         await client.subscribe(input_topic, 1)
118
119 async def on_output(msg, output):
120     for output_topic in output:
121         await mqtt_client.publish(output_topic, msg, qos = 1)
122
123 def get_nodes():
124     return nodes_str
125
126 def stop():
127     for id in nodes_id:
128         func_name = "stop_" + id
129         if func_name in globals():
130             getattr(sys.modules[__name__], func_name)()
131
132 async def exec(mqtt_c, capabilities_array, c_id):
133     global mqtt_client
134     global capabilities
135     global client_id
136     mqtt_client = mqtt_c
137     capabilities = capabilities_array
138     client_id = c_id
139     for id in nodes_id:
140         func_name = "exec_" + id
141         if func_name in globals():
142             getattr(sys.modules[__name__], func_name)()
143     return

```

Listing 5.1: Code generated from the flow presented in Figure 5.3.

However, there are limitations to this solution. If two consecutive *nodes* are assigned to the same device, the communication between them is still dependent on MQTT. This is not an ideal solution since one *node* could call the other through code, passing its output as arguments.

5.2.2.3 Custom Nodes

As previously mentioned, all the existent *nodes* are compatible with the modified Node-RED. Nonetheless, for a *node* to be orchestrable, it must be modified to comply with the code generation needs. As proof-of-concept, the following *nodes* were modified or created to be orchestrable:

IF node which receives an input and verifies if it complies with all the given rules, returning true or false;

AND node which receives a given number of inputs and verifies if all of them are true or false, returning the corresponding boolean;

TEMP-HUM node that read the temperature and humidity from a DHT sensor present in a specific pin;

FAIL node that raises a `MemoryError` exception;

NOP node that simply redirects the received message in its input to its output;

MQTT IN and MQTT OUT nodes that subscribe and publish MQTT topics, respectively.

Additionally, each of these *nodes* has two available properties: Predicates and Priorities. Similar to the Kubernetes logic of assigning containers to machines [16], the predicates dictate constraints that cannot be violated, and priorities are requests that are advisable and recommended but can be violated if impossible to comply.

These *nodes* provide enough functionality to wire simple *smart home* scenarios and validate our approach.

5.2.2.4 Device Registry

IoT systems are typically built on top of heterogeneous parts, with different capabilities and resources, and their network can be highly-dynamic (devices appearing and disappearing according to factors such as battery levels, hardware/software failures and communication interference). To maintain the *list* of network available devices along with their capabilities, there is a need for a Device Registry [54].

When a device becomes available, it sends information about itself to an MQTT topic. This information contains the device's IP address, their capabilities and their status — if the device has failed before. In its turn, Node-RED contains a *Registry node* that listens to the announcements MQTT topics and saves the devices information. If this *node* is connected to an *Orchestrator node*, each new device is communicated so that the orchestration can be updated.

When a device has an OUT-OF-MEMORY error, it triggers a FAIL-SAFE, where it reboots the HTTP server, stops running any script and restarts all communications. After this action, the device announces itself again but with a flag that indicates that it has failed. This way, the *Orchestrator node* knows that a device is active but not running any code, being possible that it has failed due to too much work. In that case, it can dynamically adapt and assign fewer *nodes* to the device, reducing the chances of causing another OUT-OF-MEMORY error.

5.2.2.5 Computation Orchestration

Node-RED must be capable of distributing computation amongst available resources (*i.e.*, devices), thus, given a set of tasks, it must assign them to available devices, ensuring that they will be performed.

The requirements to achieve this are two-fold: first, a *node* should act as coordinator, which when provided with an available devices list, along with their respective capabilities (cf. Section 5.2.2.4 (p. 47)), should decide which device should execute specific computation *nodes* — *Orchestrator node* — and, second, the orchestrable *nodes* should provide both Predicates and Priorities that must be met to assure their correct execution (cf. Section 5.2.2.3 (p. 46)).

Algorithm 1: Greedy algorithm for *node* assignment.

```

Input   :deviceList
Output  :bestDevice

1 init
2   node: node
3   bestIndex: 0
4   bestDevice: null

5 onInput
6   for device in deviceList do
7       if not all node.predicates in device.tags then
8           return
9        $intersectionIndex \leftarrow \sum node.priorities \in device.tags / \sum priorities \in node$ 
10       $nodesPerDevice \leftarrow \sum nodes \in device$ 
11       $nodesPrioritiesPerDeviceRatio \leftarrow \sum node.priorities \in device.tags / \sum device \in tags$ 
12       $matchIndex \leftarrow intersectionIndex * 0.5 + (1/nodesPerDevice + 1) * 0.4 +$ 
           $nodesPrioritiesPerDeviceRatio * 0.1$ 
13      if matchIndex > bestIndex then
14          bestIndex  $\leftarrow matchIndex$  // Device is best for node
15          bestDevice  $\leftarrow device$ 
16  return bestDevice

```

The assigning algorithm uses the devices capabilities and each node's Predicates and Priorities to assign *nodes* to devices. With a greedy approach, the algorithm filters the devices that comply with each node's predicates and assigns the one with a higher value of a heuristic, cf. Algorithm 1. This heuristic takes into account the number of priorities the device can provide, as well as the number of already assigned *nodes* the device has. The goal is to assign each *node* to the best possible device, spreading the tasks through all the available devices. An example of a possible assignment can be seen in Figure 5.4 (p. 49), where the assignment matches the nodes' priorities with the devices' tags while spreading the *nodes* over the available devices.

After assigning all *nodes* to a specific device, a code script is generated for each device (cf. Section 5.2.2.2 (p. 43)). Due to the constrained memory of the devices, the number of *nodes* assigned to a device may exceed their resources. In that case, the device will FAIL-SAFE and return an error to the assignment request. The orchestrator will receive this information and repeat the process, assigning fewer *nodes* to the devices that returned an OUT-OF-MEMORY error. If a device does not return any response, the orchestrator will assume that the device is unavailable and not assign any *node* to it.

The *Orchestrator node* can be triggered — proceeding to a system (re)orchestration — by the following events: (1) start of the system, when there is already a defined flow in the configuration,

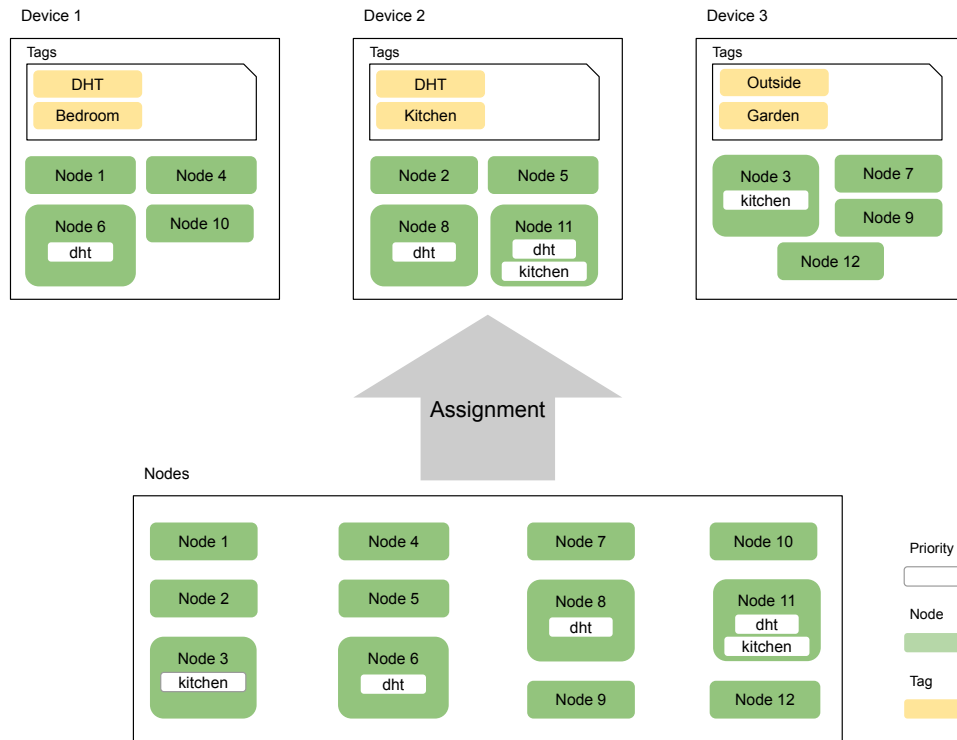


Figure 5.4: Node assignment example.

the assignment start after a period of 3 seconds, to give time for the devices to be registered by the registry node; (2) deployment of the entire flow using the Node-RED editor or API; (3) appearance of a new device detected by the *Registry node*; (4) failure or recovery of a device, which, working as a complement to the *Registry node*, is detected using PING/ECHO pattern [57] which periodically *pings* the devices in the system to assert their operational status. A visual representation of these events can be seen in Figure 5.5 (p. 50).

There are, however, some limitations in the assignment process, mostly due to the algorithm used. There are cases when the resulting assignment is not the best possible solution, and sometimes the orchestration can fail since it can not comply with the constraints imposed by *nodes*. As an example, given a scenario where the number of devices is too small for the number of *nodes*, the devices may be kept at their resources limits, *i.e.*, memory. If there is a *node* which constraints can only be compiled by one device, but that one device already has the maximum number of *nodes* it can handle, the assignment is not possible.

In addition to this, the assignment algorithm does not take into account the connections between *nodes*. As mentioned before, sequential *nodes* in the same device communicate via MQTT topics instead of calling themselves through code. However, if that were not the case, it would be advantageous if sequential *nodes* were assigned to the same device. This would allow better performance, less communication load and less dependency on an external MQTT client.

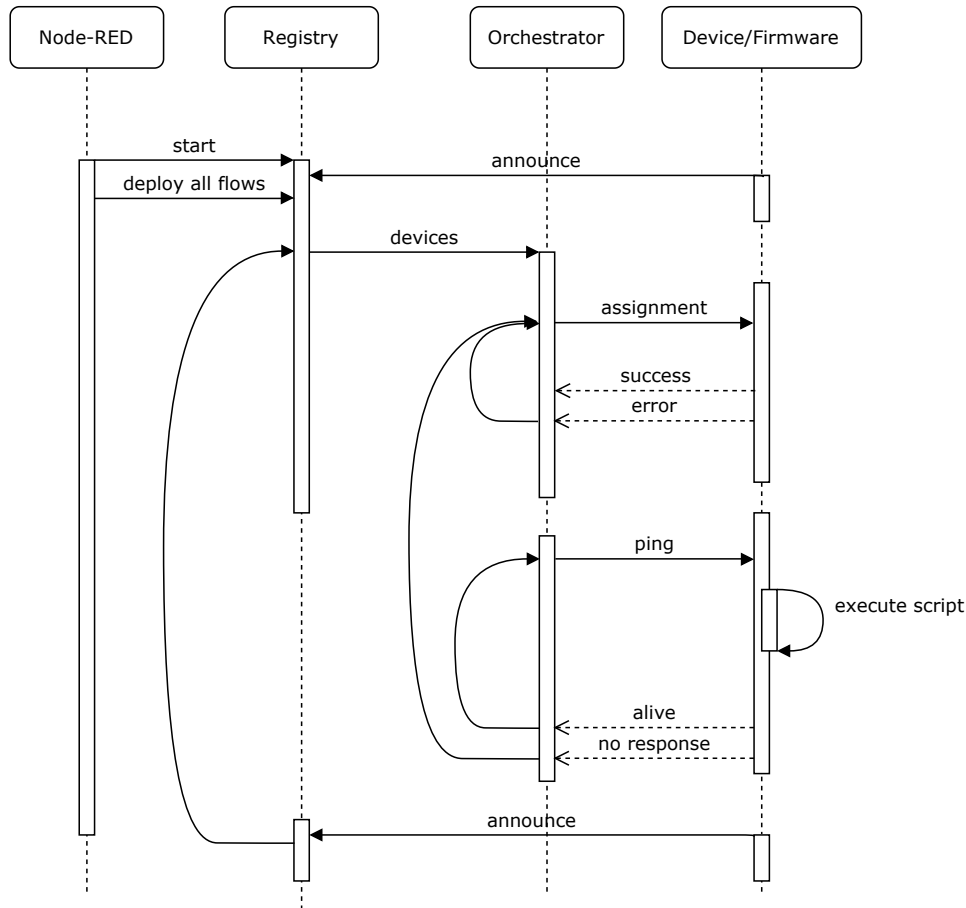


Figure 5.5: Sequence of events for orchestration.

5.2.3 Summary

The developed solution covers all the desired requirements established in Section 4.2 (p. 36). It identifies available devices in the network and decentralizes the given computation through them. It translates each computation task into something comprehensible for the devices. Additionally, our approach maintains a record of the state of the system, adapting to any change in the availability and constraints of the devices.

However, some limitations are not trivial. Some of them were already identified in previous sections: (a) the number of *nodes* that support MicroPython code generation is small, (b) there is a chance of duplicate MQTT messages that are not being handled, (c) the (re)orchestration using the editor or API is only possible by deploying the entire instance, and not specific *flows* or *nodes*, (d) *node* assignment does not favour the assignment of sequential nodes, which would improve efficiency if (e) the script generation did not force all communications between *nodes* to be through MQTT, instead of allowing a *node* to call other through code, passing its output as parameters.

Despite the limitations, the developed solution solves the issues identified in Section 4.1 (p. 35) and provides a decentralized option to a previously centralized product.

Chapter 6

Evaluation

6.1	Scenarios and Experiments	51
6.2	Discussion	54
6.3	Hypothesis Evaluation	68
6.4	Lessons Learned	69
6.5	Conclusions	70

This section evaluates how the solution developed proves the hypothesis proposed in Section 4.4 and fulfills the requirements presented in Section 4.2. Section 6.1 proposes scenarios and experiments that will be used to evaluate the developed solution. Section 6.2 discusses the results of the experiments and concludes the fulfillment of the requirement each test aims to assess. Section 6.3 verifies if the main hypothesis is proved with the evaluation data. Finally, Section 6.4 reflects on the lessons learned during the evaluation process.

6.1 Scenarios and Experiments

To validate to which extent we were able to reach the goals as mentioned earlier (*cf.* Section 1.3 (p. 3)), we proceed to evaluate our architecture and implementation *proof-of-concept* in both virtual — using Docker with a Unix-compatible version of MicroPython — and physical setups — using both ESP8266 and ESP32 connected in the same Wi-Fi network.

The experiments were run in a computer with an i5-6600K at 3.5GHz processor, with 16Gb of RAM and running Linux Manjaro kernel version 5.6.16. The base Node-RED was version 1.0.6, Mosquitto MQTT broker at version 1.6.10 and MicroPython firmware at version 1.12.

We outline the following two experimental scenarios as a foundation for our experiments:

ES1 A room has three sensors that give temperature and humidity readings every minute. There is a virtual sensor that compares the results (of both temperature and humidity) and triggers

depending on configured thresholds. An AC unit must be provided with the comparison result to define both (a) if it *switches on/off*, and (b) its operating mode: *cool*, *heat*, and *dehumidify*. The Minimal Working System (MWS) consists in (a) one temperature sensor, (b) one humidity sensor, (c) one *node* capable of making the decision, and (d) working communication channels amongst them.

ES2 A system contains 20 devices that are responsible for propagating an injected message to their output. In the end, the injected message must reach an output MQTT topic.

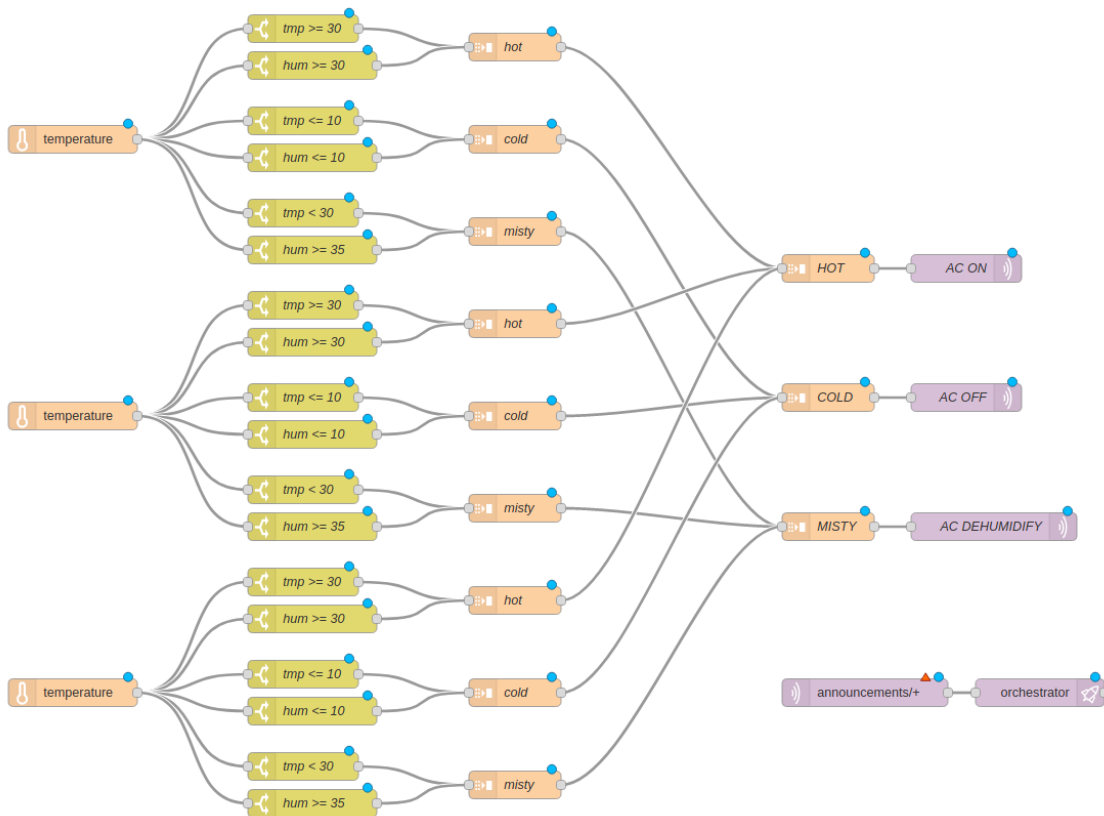


Figure 6.1: Node-RED implementation of scenario 1

ES1, which resembles to a real-world scenario, aims to test the features of the developed solution with a moderately simple Node-RED *flow* (cf. Figure 6.1), taking advantage of the *nodes* developed for MicroPython code generation support. As a complement, **ES2** allows the comparison of the developed solution to the already existing ones.

For each one of the experimental scenarios (*i.e.*, **ES1** and **ES2**) we defined a set of experimental tasks.

For **ES1** two Sanity Checks were performed, one over virtual devices — Sanity Check 1 (**ES1-SC1**) — and other with physical devices — Sanity Check 2 (**ES1-SC2**). A set of readings and message forwarding tasks were performed with no compensation or any other fault-tolerance strategy. Each sensor device only provides environmental readings to the system. Orchestration

is centralized. We expect all roundtrips to take less than the smallest part that can be resolved (measurement capability, which we estimate to be < 1 second).

We further defined a set of (re-)orchestration experiments for **ES1**, where the system must allocate computation tasks among the available resources (*i.e.*, devices), namely:

ES1-A MWS is achieved via multiple possible configurations by selective (provoked) device failure (fail-stop) using only virtual devices (*i.e.*, Docker);

ES1-B MWS is achieved via multiple possible configurations by selective (provoked) device failure (fail-stop) using physical devices;

ES1-C Inconsistent device behaviour, *e.g.*, appear and disappear in shorter intervals lower than the time needed for orchestrating convergence (OCT), that leads to activity impacting the MWS;

ES1-D With 20 devices, each one with different processing capabilities. During orchestration, some devices will develop an out-of-memory error because they cannot process all the processing tasks assigned to them, specifically the size of the script given. The orchestrator decides to send fewer tasks to these devices. The system will converge in a working solution. *This scenario will be implemented with a modified device script. When devices receive a script, it will generate a memory error if the length of the script passes a certain threshold. This simulates the memory constraints of devices when receiving a file too big.*

ES1-E With 20 devices, some of them have a memory leak with an unknown cause. After random time $\text{Random}(t_0, t_1)$, these problematic devices stop working with an out-of-memory error. The orchestrator thinks that the devices cannot handle the number of processing tasks assigned to them, so in the re-orchestration, it will assign fewer tasks. Since these devices will always break, the orchestrator will eventually not consider these devices in the assignment of nodes. *This scenario will be implemented with a modified device script that will trigger an out-of-memory error after a random period after executing the given tasks.*

ES1-F With 20 devices, there is a device that is sensitive to a particular node, which causes the device to give out an out-of-memory error. The orchestrator will potentially assign this *node* to the specific device. When the device gives out the out-of-memory error, the orchestrator will eventually converge in a solution where the *node* is not assigned to the particular device, and the system will converge. *These out-of-memory errors will be simulated with the use of a failure node that forces an `MemoryException` in the device.*

ES1-G With 50 devices, each second, the device has a probability of failing. This failure can go from 0 to 10 seconds, randomly chosen. The orchestrator must deal with the random failure of the devices and re-orchestrate the system. This experiment is considered a stress test, causing repeated failures and forcing constant re-orchestration.

With this set of experiments we can verify that the following constraints are met:

1. **Restrictions (predicates) are enforced.** Check that possible configurations lead to solutions that enforce defined predicates;
 - (a) Temperature and humidity might coexist in the same, or in dedicated, devices;
2. **Priorities are honored.** Check that all specified priorities were taken into account, and only violated if necessary;
 - (a) Priority is given to edge devices, but fog and cloud can be used;
 - (b) Priority is given to the maximum level of decentralization, but some centralization can be used.

Regarding **ES2**, a total of 20 devices were connected in a line topology. A message is sent to the starting device, which will propagate it to its output. All the devices implement this propagation logic, resulting in the initial message reaching the end of the line. The propagation time is measured, starting when the message is sent and ending when the message reaches the last node. This scenario was implemented with different experimental configurations, namely:

ES2-A Non-modified version of Node-RED, using the default *node-to-node* communication channel (`EventEmitter`), with all the *nodes* sharing the same runtime.

ES2-B Modified version of Node-RED that uses MQTT as the *node-to-node* communication channel, with all the *nodes* sharing the same runtime.

ES2-C MQTT-based modified Node-RED, where each *node* of the *flow* is assigned to a different virtual device (*i.e.*, a MicroPython-running Docker instance). The Docker instances and MQTT broker run in the same host machine.

ES2-D MQTT-based modified Node-RED, where each *node* of the *flow* is assigned to a different virtual device. The Docker instances share one host, but the MQTT broker is in a different one. All parts are connected to the same Wi-Fi network.

ES2-E Each physical device runs a simple script that performs the desired behaviour, on top of a non-modified MicroPython firmware image, communicating with which other over MQTT. Node-RED is not used, and there is no orchestration being performed.

ES2-F MQTT-based modified Node-RED, along with the modified MicroPython firmware running on physical devices. Each *node* of the *flow* is assigned to a different device. Each device is connected to the same Wi-Fi network and communicate between them using MQTT.

6.2 Discussion

The scenarios and respective experimental tasks were performed, and several metrics of the system were measured. The following paragraphs present and discuss the experimental results.

6.2.1 ES1: Sanity Checks

As mentioned previously (*cf.* Section 6.1, p. 51), the first scenario consists of a system that controls an A/C. This system takes into account readings of 3 temperature and humidity sensors to define if the room's temperature is too hot, cold or humid and sends commands to the A/C with the respective actions.

These experiments will prove that the devices can satisfy the *nodes*, meaning that the system works as intended once the assignment is complete. With this premise, this will not be verified again in the remaining experiments.

6.2.1.1 ES1-SC1

This experiment was used to test the overall functionality of the approach in a controlled way (the use of virtual devices reduce the proneness to hardware-provoked failures), the experiment resulting *node* orchestration can be observed in Figure 6.2 (p. 56), where the *Orchestrator node* allocated nine *nodes* to each device. Figure 6.3 demonstrates in which devices each *node* was assigned to.

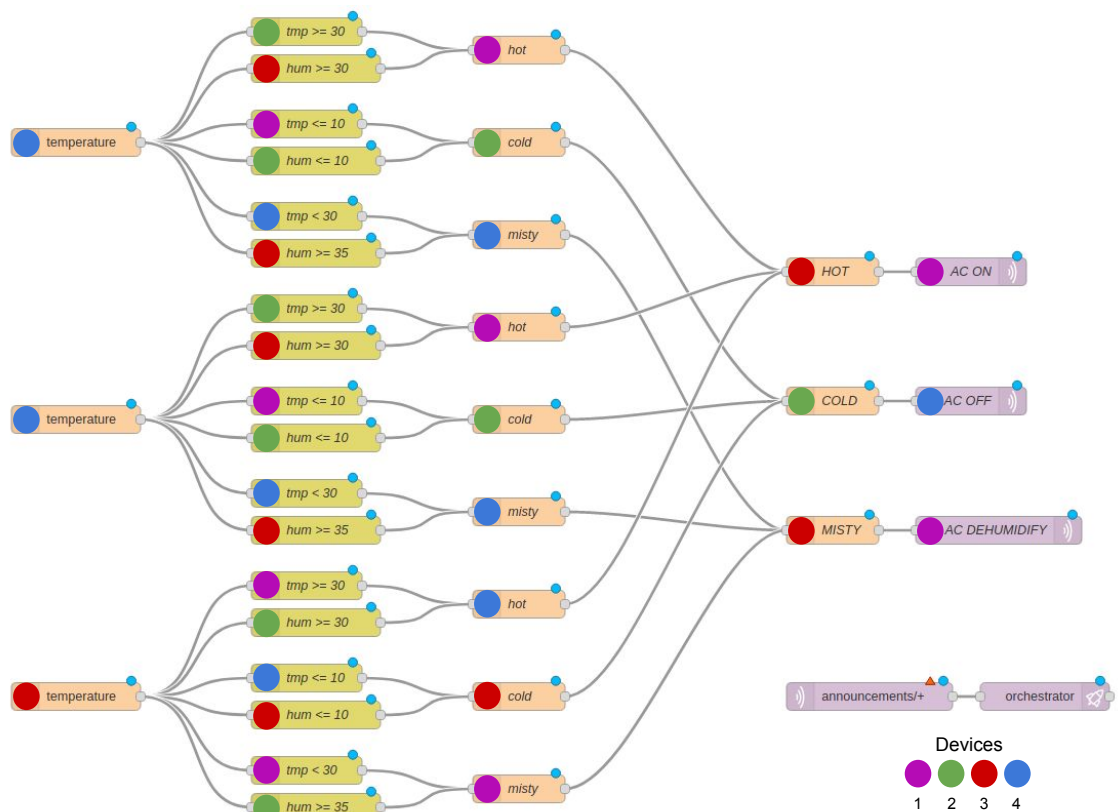


Figure 6.3: ES1-SC1 *node* assignment.

The usage of RAM was significant, varying from 60Kb to 200Kb (*cf.* Figure 6.2, p. 56). The flash size only decreases around 150000 bytes when the device receives a script for executing — matching the size of the payload received by the devices.

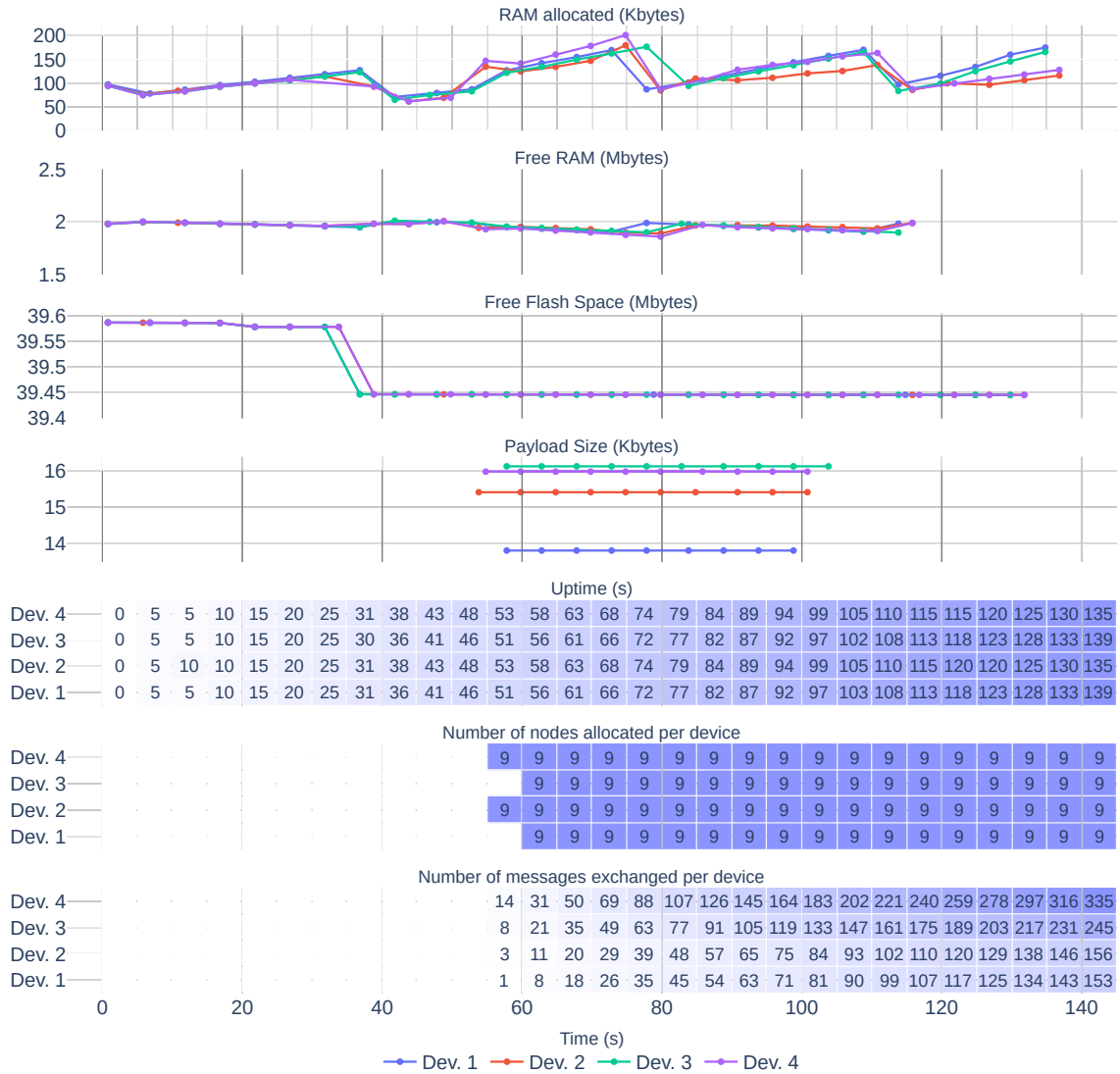


Figure 6.2: ES1-SC1 measurements.

As the orchestrator defines the *nodes* assignment, a script is built and send to the devices. The confirmation of this delivery is necessary for the system to conclude the assignment phase and start monitoring the state of the system. The time it takes to deliver the script can be consulted in Figure 6.4. The usage of virtual devices running in the same host as the Node-RED instance allows for shorter times, which are measured in milliseconds.

Time (s)	Device 1	Device 2	Device 3	Device 4
53.8	315	299	314	300

Figure 6.4: ES1-SC1 script delivery time.

Once the devices execute the script given to them, each *node* allocated to a device start to communicate with each other. All the messages of all communicating topics were capture to check if the system worked as expected. This allowed us to verify that all *nodes* are receiving the messages and producing the expected output messages. The number of communications can be consulted in Figure 6.2 (p. 56). As it can be observed, the number of messages produced by *Device 4* is bigger than any other. This is because in this device, two temperature-humidity *nodes* were allocated, which publish three messages which are more than produced by any other device. The *Device 3* contains the other temperature-humidity node (cf. Figure 6.3, p. 55).

To verify if the script delivery time is directly related with the payload size, this experiment was repeated 10 times and the script delivery time for each device was measured. By analyzing Figure 6.5, we can observe that the script delivery times between devices is very similar. Therefore, there is no relation between payload size and script delivery time. This verification allowed the making of an average of the delivery time, which is $0.303 \pm 0.165s$.

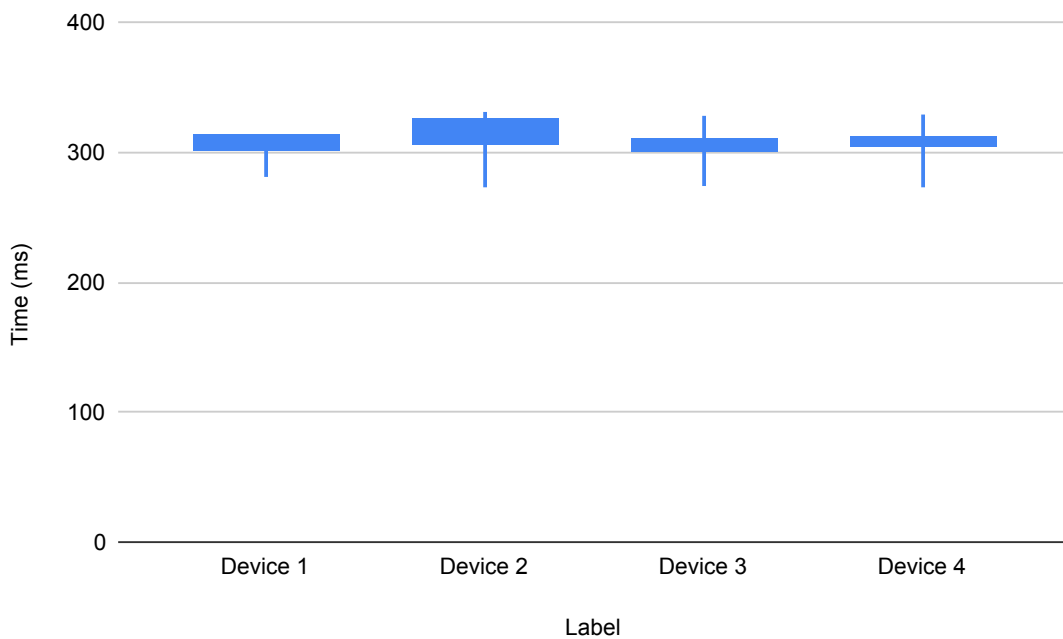


Figure 6.5: ES1-SC1 script delivery times.

The sanity check approach performs as expected in a virtual-only setup by (1) spreading the computation amongst available resources and (2) maintaining the system within expected behaviour.

6.2.1.2 ES1-SC2

The sanity check experiment was repeated using physical devices, more specifically 4 ESP32. Similar to the virtual devices, the assignment of *nodes* to devices spread the number of *nodes* equally, with each device running 9 *nodes* (cf. Figure 6.6, p. 58).

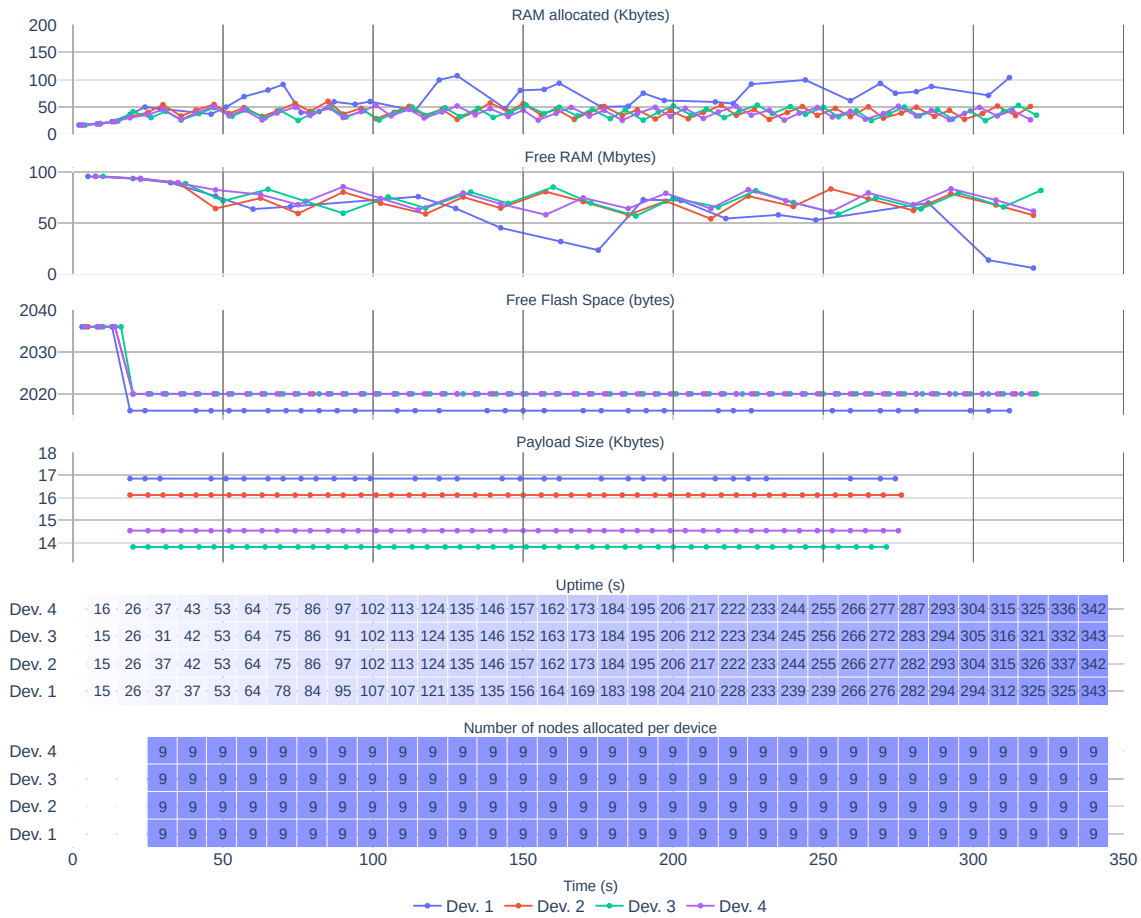


Figure 6.6: ES1-SC2 measurements.

The usage of RAM in physical devices is smaller than the one used by virtual devices, which can be explained with the possible optimization differences in the Docker-compatible and ESP-compatible MicroPython firmware and libraries.

The free flash space of the physical devices is smaller than the virtual ones, as expected. Fig. 6.6 shows that the device with the biggest payload, *Device 1*, ends up having less free flash space. The overall size of the payloads is very similar to the ones in the **ES1-SC1**.

Time (s)	Device 1	Device 2	Device 3	Device 4
23	null	null	6465	null
24	6610	6546	null	7486

Figure 6.7: ES1-SC2 script delivery time.

The script delivery time for physical devices is longer than their virtual counterpart, with an average of 6.776 ± 0.476 s (cf. Figure 6.7). Since things are not running in the same machine, the

Wi-Fi stack and the devices' hardware characteristics hinder the communication speed. The uptime is similar to the previous experiment since there were no hardware failures.

6.2.2 ES1: Experimental Tasks

These experiments focus on validating and evaluating the tool's capacity to adapt to devices' failures. Since the previous experiments (*cf.* Section 6.2.1, p. 55) already proved that the system executes the tasks that are given to them, this verification will not be repeated in this section's experiments.

6.2.2.1 ES1-A

This experiment evaluates if the system can re-orchestrate when a device either fails or (re-)appears (*i.e.*, new or recovered). During this experiment, devices were turned off one by one until only one was left running. It is expected that the system detects when a device has become unavailable and re-orchestrates, assigning *nodes* to the available devices. In the end, only one device is running, and all the *nodes* are assigned to it.

Figure 6.8 (p. 60) shows that the uptime of the devices stops increasing one by one, identifying the moment the device fails. Once a failure happens, the system re-orchestrates, assigning the *nodes* of the device to the other available devices, increasing their number (*cf.* Figure 6.8 (p. 60)). The increase in the number of *nodes* assigned to the available devices can also be observed in the payload size. When all devices fail except one, the one remaining is the only one that receives the payload, which is higher than any other previously received.

The information regarding the number of *nodes* is not updated to zero once the device fails, since it is no longer active to send the updated metric. This allows us to verify that the system identifies the failure of devices and takes actions to rectify it by repeating the assignment process, taking into account the available devices.

This allows us to verify that the system identifies the failure of devices and takes actions to rectify it by repeating the assignment process, taking into account the available devices.

6.2.2.2 ES1-B

Based on ES1-A (*cf.* Section 6.2.2.1), this experiment replaces the virtual devices by physical ones.

The payloads and number of *nodes* assigned through the experiment are very similar to the experiment ES1-A (*cf.* Figure 6.9, p. 61). However, it is noticeable that *Device 2*, the last remaining active device, fails when receiving the final-step payload — which contains the code for all the *nodes* of the system, since no other device is available.

The device constrained memory cannot handle the payload size, so it FAIL-SAFES, informing the system that there was an *Out-of-Memory* error, which results in *Orchestrator node* assigning fewer *nodes* to the device.

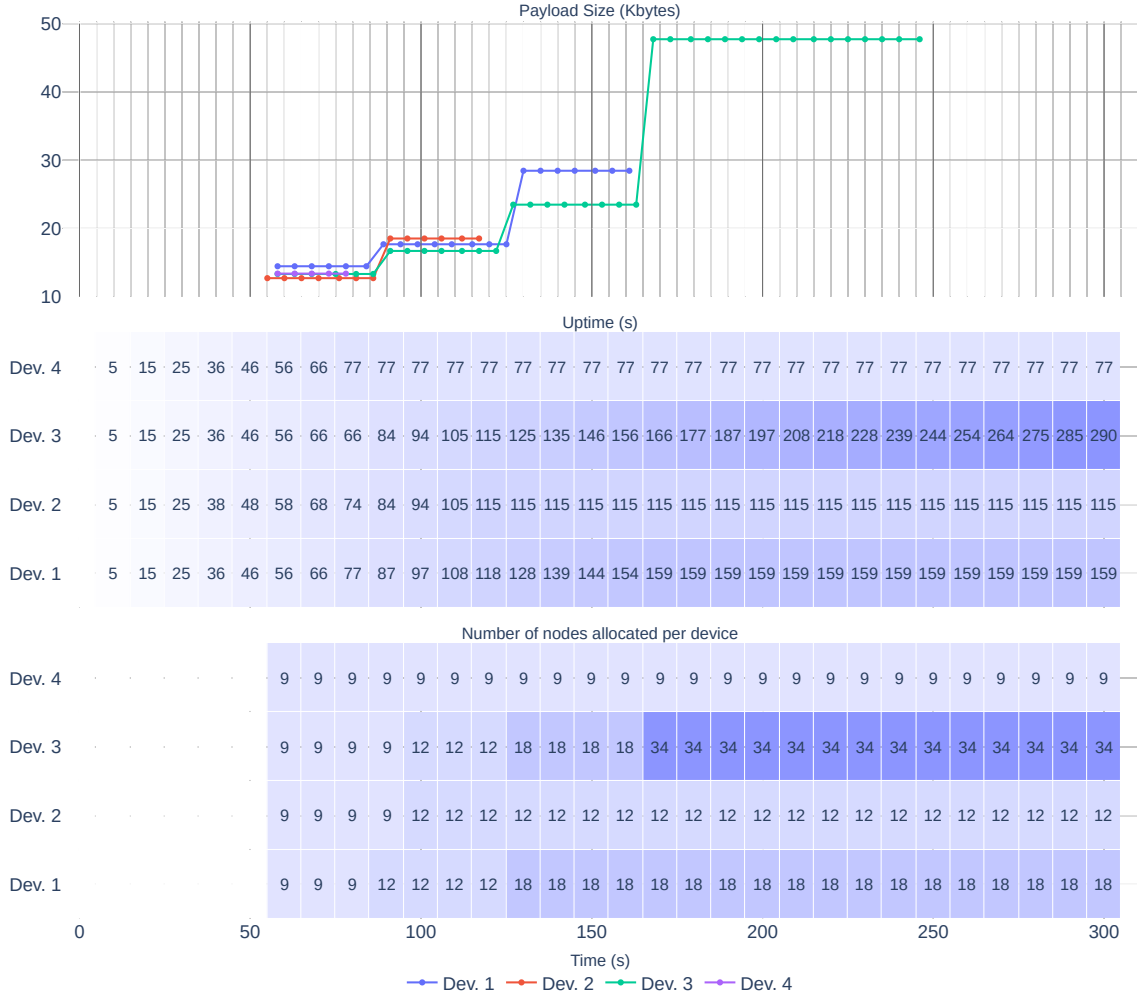


Figure 6.8: ES1-A measurements.

6.2.2.3 ES1-C

Similar to **ES1-A** and *ES1-B*, this experiment focuses on testing the system's ability to recover when devices fail and then recover. Figure 6.10 (p. 62), *Device 3* and *Device 4* fail early on and the system recovers, spreading the *nodes* assigned to them to other devices. *Device 4* recovers around the 100s, fails again and then recovers. The system did not catch this change since it was swift, and the system only re-orchestrates the second time *Device 4* recovers. During this experiment, *Device 3* and *Device 4* continue to fail and recover, and the system always re-configures itself.

This re-orchestration ability when a device recovers can be taxing to the functionality of the system. If a device is continuously failing and recovering, the system will always adapt itself, halting its functionality to orchestrate.

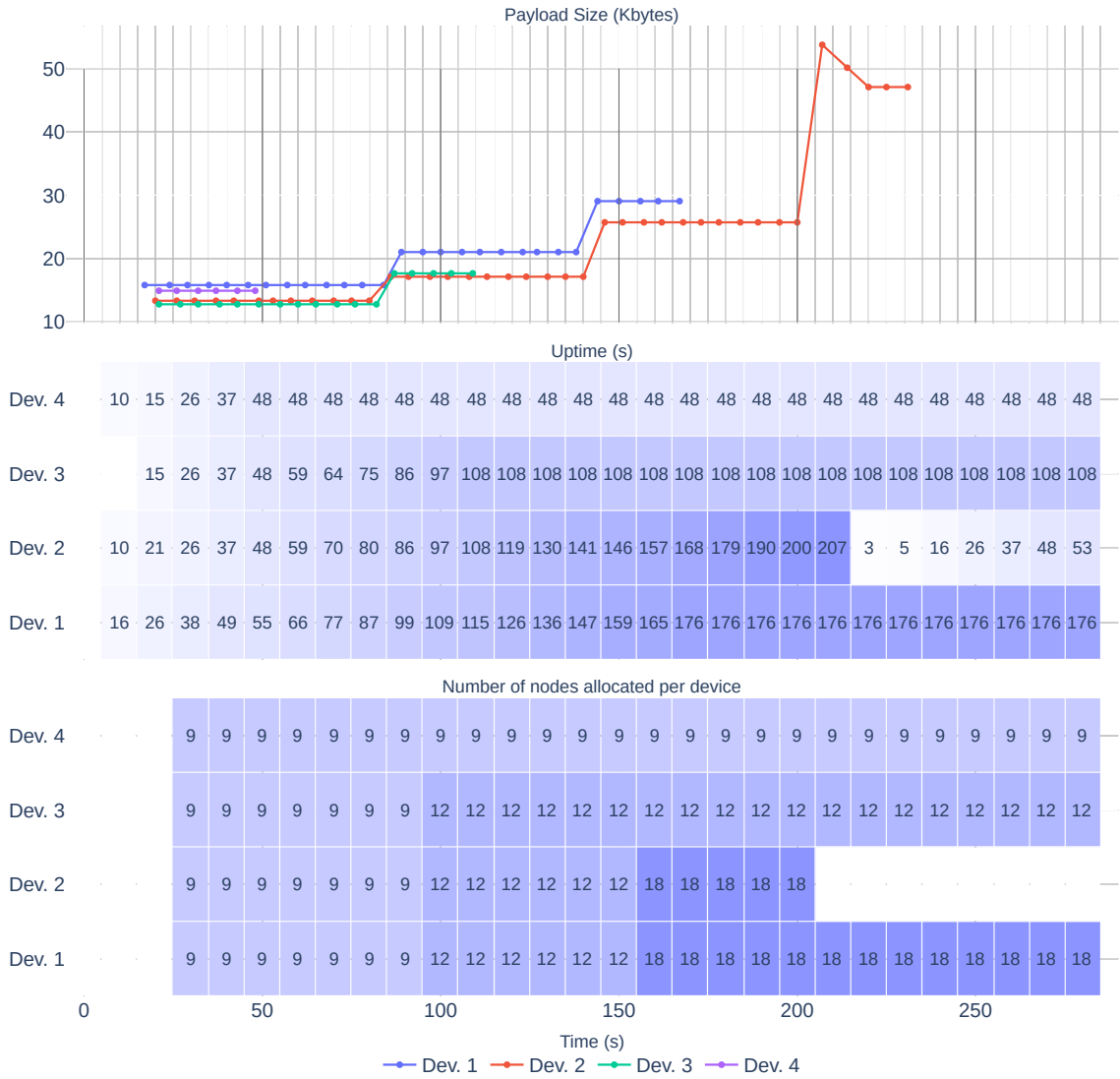


Figure 6.9: ES1-B measurements

6.2.2.4 ES1-D

The memory constraints of IoT devices can negatively impact the functioning of the system, by raising memory errors when writing the received script into the device SPI flash. This experiment verifies how the system recovers and adapt to the device's memory constraints.

Figure 6.11 (p. 63) shows the constrained memory of the *Device 2* and *Device 4*. When the first assignment is made, at approx. 50 seconds, both these devices FAIL-SAFE due to *Out-of-Memory* errors. The number of *nodes* present on these devices are the ones assigned after they communicate to the orchestrator their limitations.

To assess if the system saves information about the limitations of the devices, one of them was turned off and later turned on (*cf.* Figure 6.11, p. 63). As it can be observed, *Device 2* uptime stops increasing around the time of the event and its *nodes* are distributed by the other devices, except for

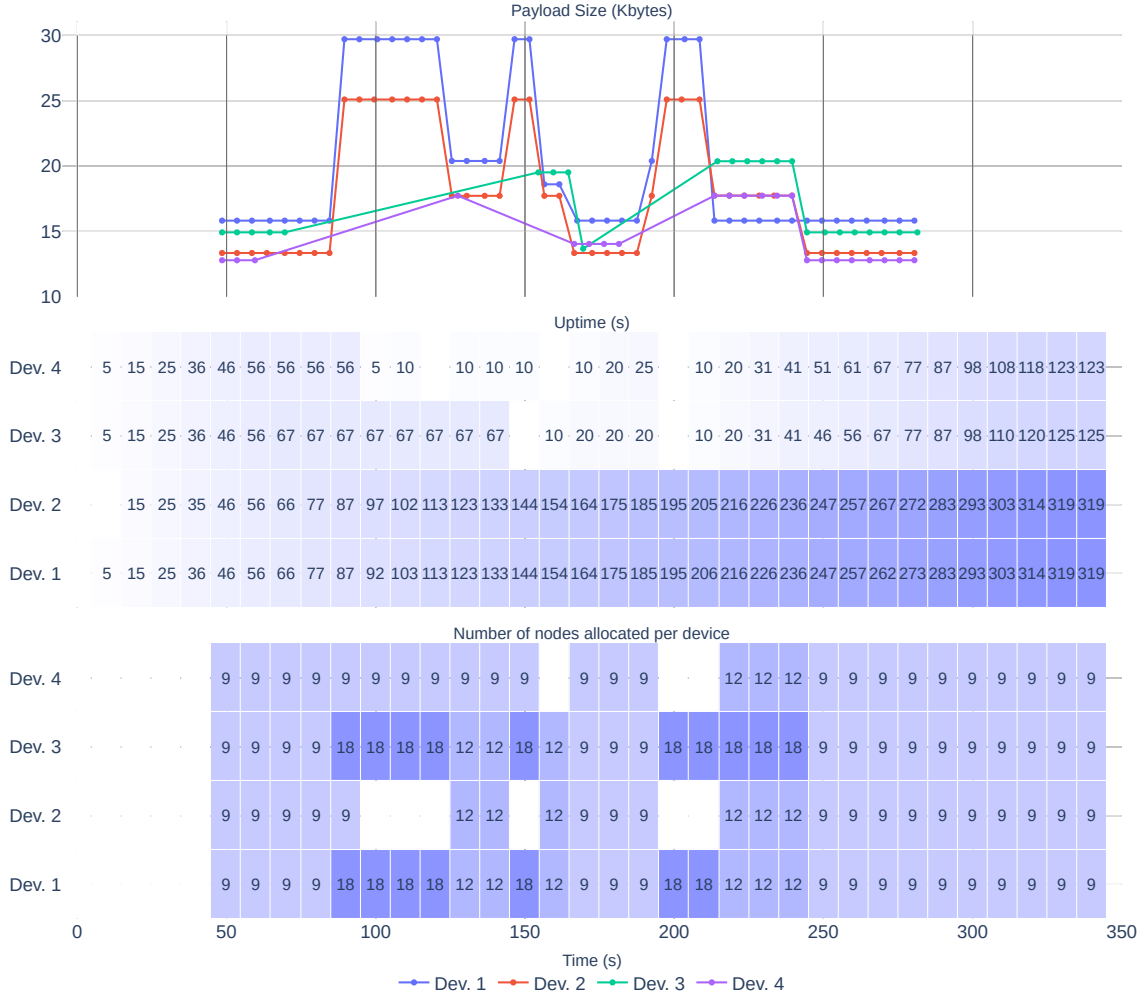


Figure 6.10: ES1-C measurements.

Device 4, which is memory constrained. After the recovery of *Device 2*, the system re-orchestrates and the same number of *nodes* is assigned to the devices. However, *Device 4* failed when *Device 2* recovered, which implies that the system repeated the process assignment process, ignoring the previously known information about memory constraints.

6.2.2.5 ES1-E

Additionally, to the handling of memory limitations, it is expected that the system can handle a damaged device which has a memory leak issue. *Device 2* was modified to always generate an *Out-of-Memory* error after a random period. The system should be able to exclude this device during the assignment process.

Figure 6.12 (p. 64) shows that *Device 2* is consistently failing after the first assignment of *nodes*, at approx. 75 seconds. The number of *nodes* assigned decreases, until no *node* is assigned and the device is excluded from consideration. This is an iterative process, in which the system will

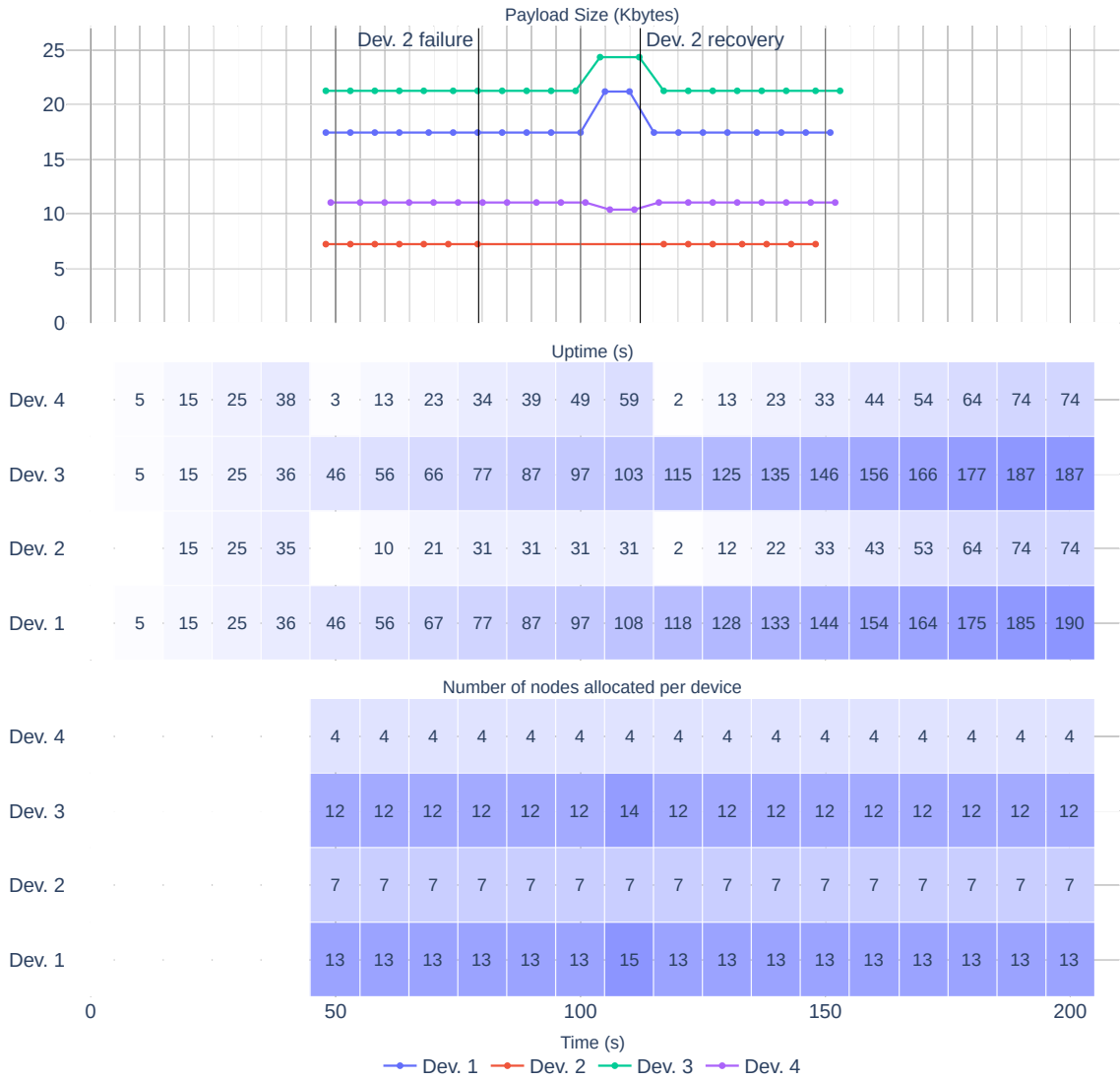


Figure 6.11: ES1-D measurements.

decrease the number of *nodes* it assigns to a device if the device communicates an *Out-of-Memory* to the orchestrator. Eventually, if the device does not handle any *node*, the minimum number of *nodes* the device can handle is zero, excluding the device from the assignment process.

6.2.2.6 ES1-F

To further assert the resilience of the system to *Out-of-Memory* errors, a *node* was deliberately injected that causes such error in specific devices. It is expected that the system re-orchestrate and converge in a solution where the specific *nodes* are assigned to devices not affected by them. In their turn, the devices affected by these *nodes* should have fewer *nodes* assigned to them. The system and devices do not know that a specific *node* is creating the *Out-of-Memory* errors and interpret the error as a device problem.

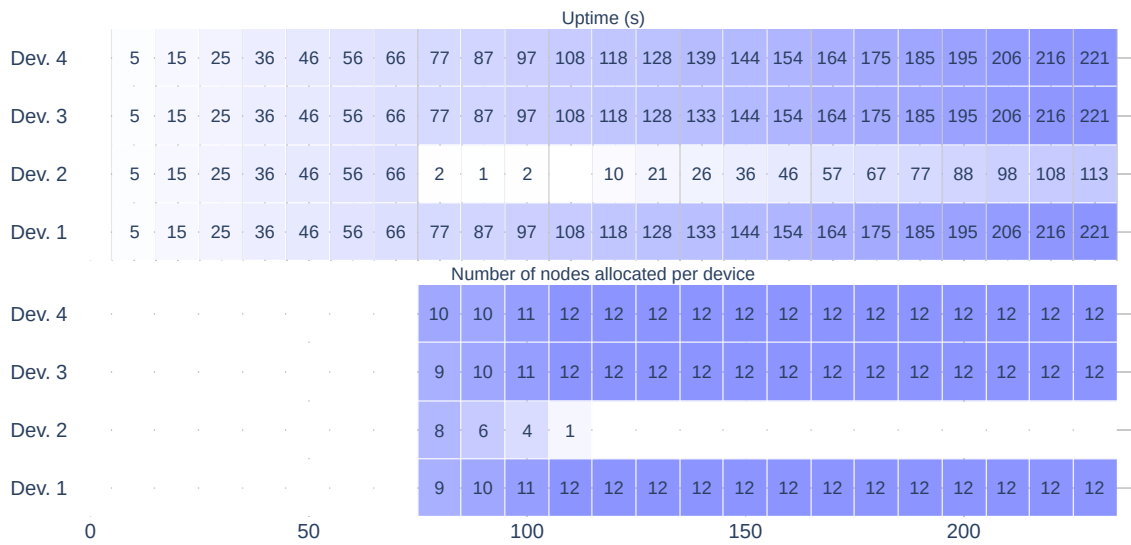


Figure 6.12: ES1-E measurements.

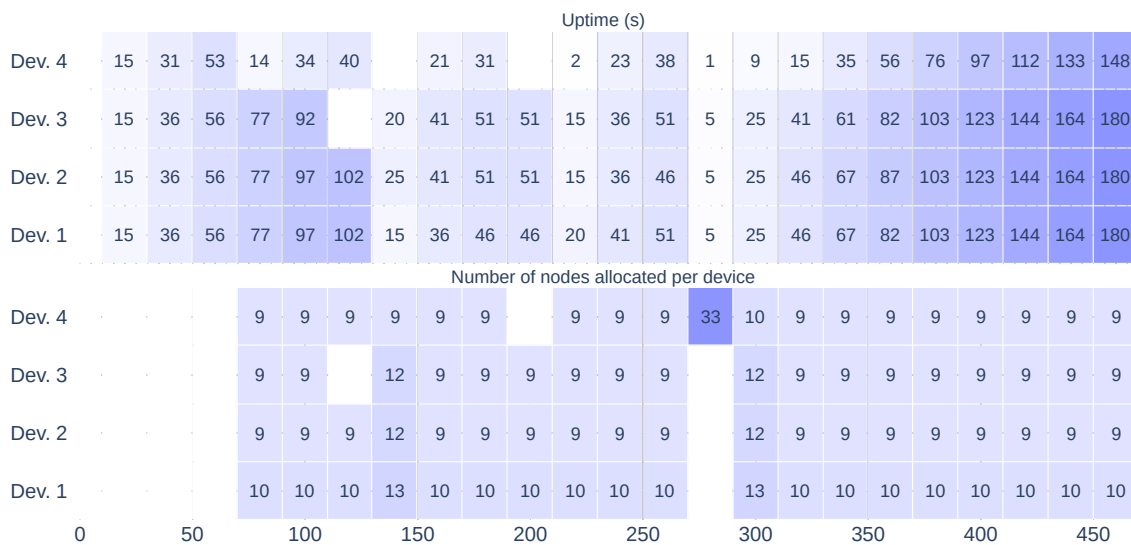


Figure 6.13: ES1-F measurements.

Since the first assignment can already be correct by default, meaning that these faulty *nodes* are assigned to devices not affected by them, some changes were made to force the system to re-orchestrate. The devices were all turned off and on in different order, this repeated three times. Figure 6.13 shows these off/on events at approx. 125, 200 and 275 second timestamps. It is important to note that the devices affected by the faulty *nodes* are *Device 2* and *Device 4*.

The event we aim to test occurs at approx. 300 seconds. As it can be seen (*cf.* Figure 6.13), *Device 4* is assigned 10 *nodes*. The uptime of *Device 4* resets in this small time period — the next uptime is less than 20 seconds — meaning that an *Out-of-Memory* occurred and the device

performed a FAIL-SAFE. The system updates, spreading the 10 *nodes* previously assigned to *Device 4* through all the available devices. Since Figure 6.13 (p. 64) shows the data in intervals of 20 seconds, the assignment in *Device 4* happens before the assignment present in the other devices.

When the system receives information that *Device 4* is available again, it already knows that it has a limitation, so it only assigns 9 *nodes* to it. It can be seen that missing *node* is assigned to *Device 1*. Since *Device 4* does not FAIL-SAFE, the *node* assigned to *Device 1* must have been the faulty one.

6.2.2.7 ES1-G

To assert the system limits, we proceed to inject constant failures in the available devices. Every second, each device has a 5% probability of becoming unavailable from 0 to 10 seconds. During this period, the device is unresponsive to the orchestrator requests and, when recovered, announces itself.

Figure 6.14 (p. 66) shows that the system is continuously re-orchestrating itself, and once the majority of devices failed, the system becomes unstable. It is important to note that, similar to previous experiments, once a device fails, the number of *nodes* does not update to zero. We then conclude that devices with the same number of *nodes* during the total execution of the system failed early on and continued to fail, not allowing another assignment by the orchestrator.

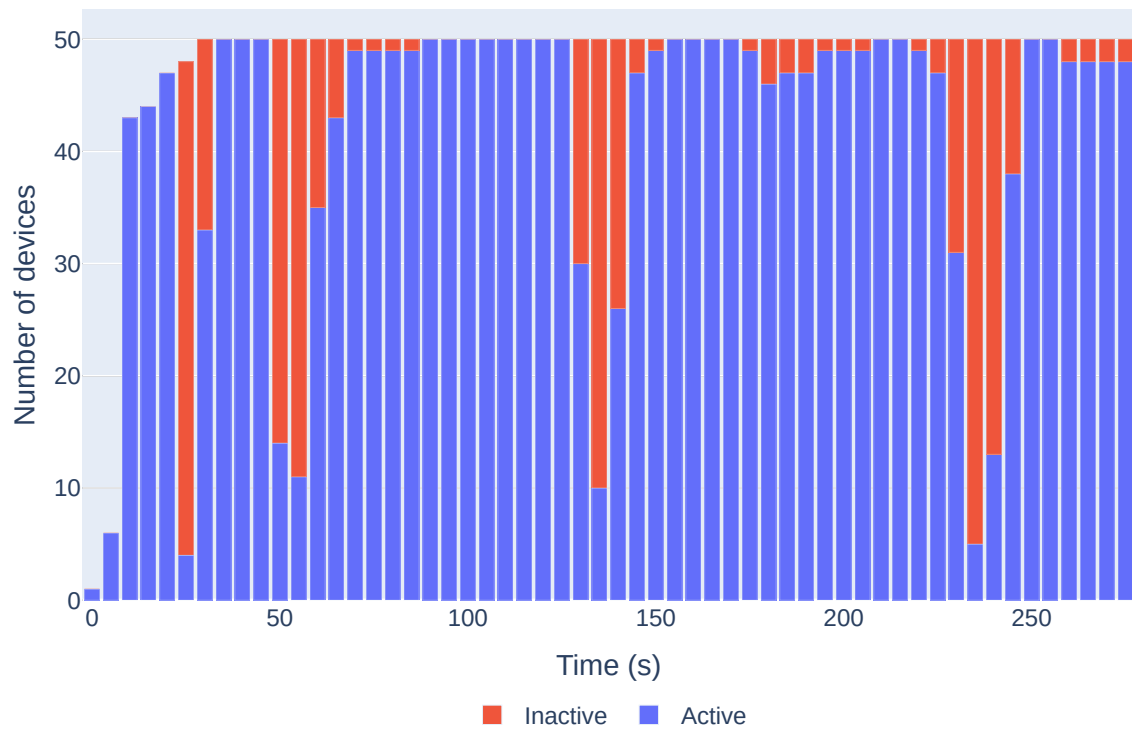


Figure 6.15: Number of devices active and inactive

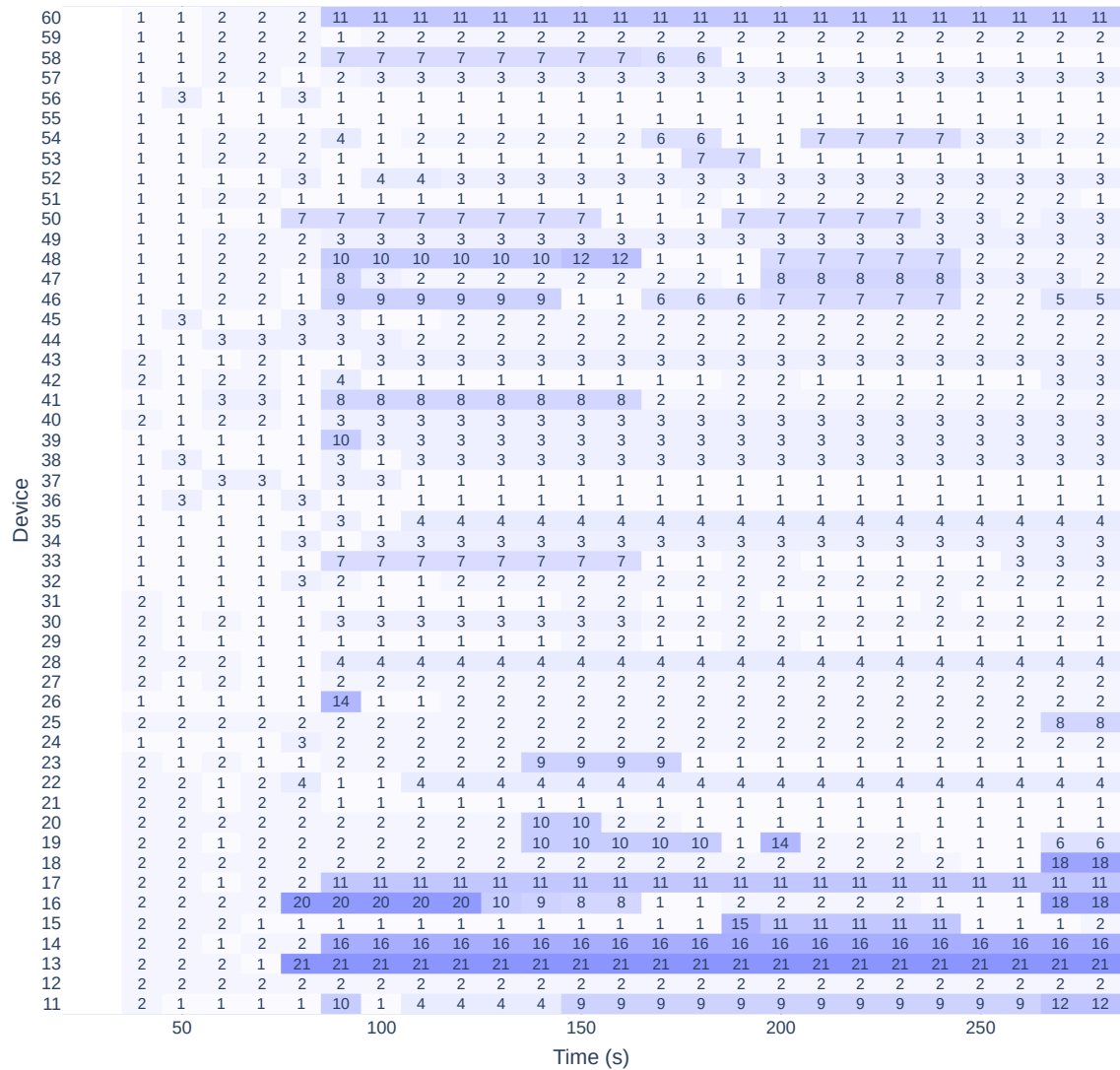


Figure 6.14: Nodes assignment distribution

At approx. 100 seconds, it can be noted in Figure 6.15 (p. 65) a period where all devices were available. However, the *node* assignment in Figure 6.14 does not converge during that time period. The reason for this behaviour is that the system will re-orchestrate when a device becomes available. Since each device announces itself individually, each announcement triggers a new orchestration. This process takes time and results in several failed orchestrations due to outdated data on the device's operating status.

The outdated data issue has been identified as a current limitation of the system, making it vulnerable to a possible Denial-of-Service (DoS) attacks in the form of an excess of status activity from the devices. This constant orchestration is also taxing for the devices, causing an overload of received assignments that will never make the system function as a whole.

6.2.3 ES2: Experimental Tasks

To benchmark, our approach against existing solutions we proceed to experiment with a *flow* that consists on passing a message through several devices, recording the elapsed time for the message to pass through all the devices.

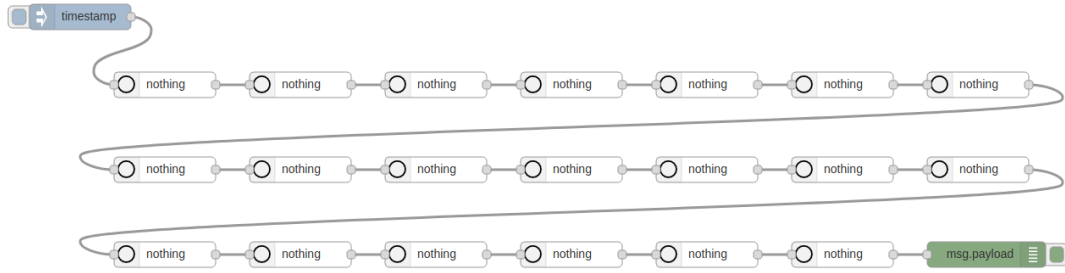


Figure 6.16: Node-RED implementation of scenario 2

The ES2 implementation in Node-RED can be seen in Fig. 6.16. The *NOP nodes* execution consists of only redirecting their input to their output. A message containing only the current timestamp is inserted into the system by "clicking" the *Inject node*, and the same message is expected to appear in the Node-RED *Debug* console (using the *Debug* node).

Label	Min	Q1	Q2	Avg	Q3	Max
ES2-A: Node-RED original	3	8	10	10	13	15
ES2-B: Node-RED + MQTT	134	353	431	489	711	883
ES2-C: Node-RED modified + Dockers (same host)	1217	1260	1318	1400	1574	1665
ES2-D: Node-RED modified + Dockers (different host)	1445	2332	2536	2392	2708	3059
ES2-E: Physical + MQTT	3616	4031	4142	4133	4372	4452
ES2-F: Node-RED modified + MQTT + Physical + Firmware	4168	4357	4569	4751	5088	5940

Table 6.1: Scenario 2 results

This experiment was run with different configurations (ES2-A to ES2-F) to assert the impact of each modification/module, as described in Section 6.1 (p. 51). Each experiment was replicated ten times, and the resulting time measurements are shown in Table 6.1.

Figure 6.17 (p. 68) demonstrates that the developed solution is considerably less efficient in communicating message between nodes. However, given the other experiences, it is possible to conclude that this lack of efficiency is caused not by the firmware created but because of the stack of communication the message as to go through, as well as the nature of MicroPython.

When the decentralization is applied inside Node-RED (*cf.* **ES2-B**), without running any MicroPython, it is possible to see that the introduction of the MQTT communication (Mosquitto broker) running in the same host causes some latency. The introduction of Dockers running the firmware in the same host as the Node-RED instance and MQTT causes additional latency (*cf.* **ES2-C**), making it possible to conclude that the MicroPython-based developed firmware also delays the communication. By repeating the same experience but with the broker running in another

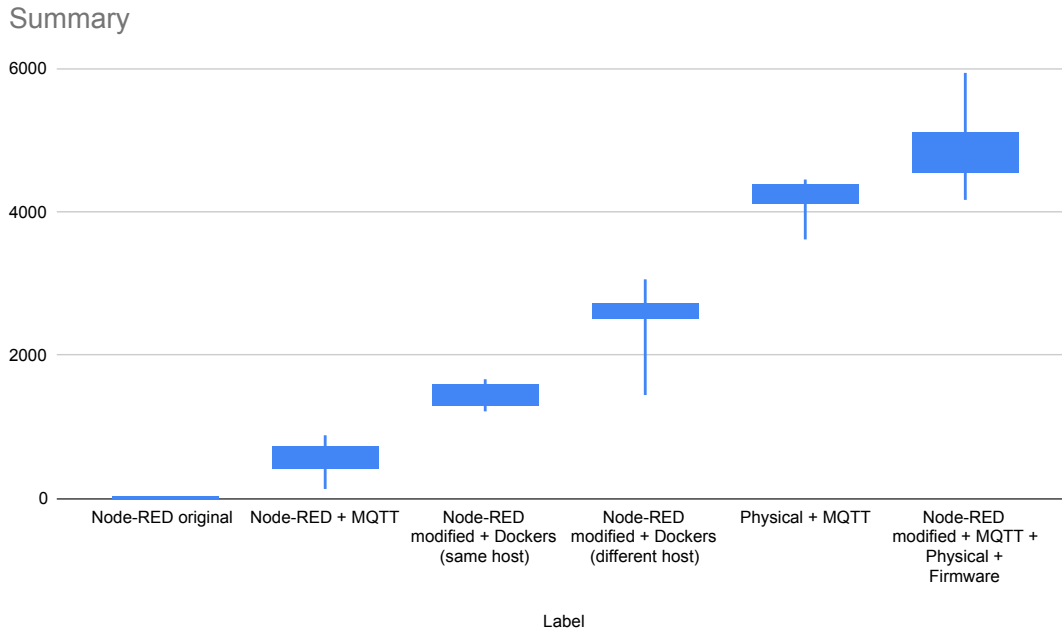


Figure 6.17: ES2 results.

machine (same network) (*cf.* **ES2-D**), it is noticeable that the times are more spread out and the overall latency of the system increases. As the Node-RED and the broker run in different machines connected over Wi-Fi, we conclude that this is the leading cause for the additional delay in communications.

The experiment was repeated in physical devices: (1) by running a simple code in the MicroPython flashed devices and injection of messages directly in the broker (*cf.* **ES2-E**), and (2) by using our approach as a whole, with the modified Node-RED and firmware in the devices (*cf.* **ES2-F**). The results allow us to conclude that the use of physical devices produces the worst times, but that the developed firmware has little impact, visible by the comparison of both their results.

We conclude that our approach, including the *node-to-node* communication change, is slower than the original Node-RED, but that mostly result of the nature of Wi-Fi communications and the base MicroPython firmware. Also, this modification makes Node-RED more *modular*, since any other pub-sub communication mechanism can easily replace MQTT.

6.3 Hypothesis Evaluation

This evaluation process aimed to prove the hypothesis presented in Section 4.4.

Given the results of the experiments with our approach proof-of-concept, we conclude that the challenges that we focus on this work were attained, more specifically the decentralization of computation, with the handling of the device's memory constraints and failures, and dynamic

adaptation of the system (self-reconfiguration). The attributes mentioned in Section 4.4 (p. 37) were evaluated, resulting in the following conclusions:

- **Resilience:** The developed solution is moderately robust, handling device failures and memory constraints dynamically at runtime. However, there are some limitations to this robustness. TAs demonstrated in Section 6.2.2.7, the system reaches a maximum point of adaptability when several devices fail and recover continually.
- **Efficiency:** The developed solution is slower than the original Node-RED system. However, the increased latency is due to the change in the communication channel, which introduces some extra latency. Despite this, the developed modules, such as the orchestrator and the device's firmware introduce little latency to the latency of the whole system.
- **Elasticity:** The solution handles a different number of devices, as it was demonstrated in the experiments. It also handles the number of devices changes throughout the lifespan of the system, adapting the orchestration to the number of devices available.

In summary, the developed solution is more scalable than a centralized one, dealing with a dynamic number of devices while taking advantage of their computational capabilities. It is also robust to handle the failures and constraints of these devices. Lastly, the developed solution is not more efficient, but its latency is due to factors external to the developed firmware, but it was mandatory for having a decentralized solution.

6.4 Lessons Learned

There were several changes made to the developed solution during the evaluation process. These changes were necessary to allow the capture of data that was later used to generate graphics and reach conclusions. The lessons learned during this process consisted in learning how to build a framework that supported sending and capture of data, as well as its aggregation and visualization.

Each device was modified to allow them to send metrics to MQTT topics, which in turn were captured by a bridge that populated an InfluxDB database. This database supplied a Grafana dashboard, where the data from the evaluation was exported from. In addition to this, both the devices and Node-RED sent data to a Logstash, which supplied both a Kibana instance using Elastic Search. These logs were sometimes useful to understand the events happening in the devices and orchestrator in more complex experiments.

The setup necessary for this process required learning tools and modifying the developed solution. It was an iterative process in which the number of metrics captured increased each time an experiment was made. Due to this fact, the evaluation process took more time than was expected.

6.5 Conclusions

This Chapter presents the results from the evaluation process of the developed solution. Section 6.1 (p. 51) starts by defining the scenarios and experiments that were used to test the tool and its compliance with the proposed requirements. Section 6.2 (p. 54) analyzes the results from the experiments and reaches conclusions about the requirement each test was measuring. Section 6.3 (p. 68) verifies if the data contrived from the evaluation process proves this dissertation hypothesis. Finally, Section 6.4 (p. 69) reflects on the lessons learned during the evaluation process.

Chapter 7

Conclusions

7.1 Difficulties	71
7.2 Future Work	72
7.3 Conclusions	73

This chapter presents an overview of this dissertation. Section 7.1 describes the difficulties faced during the development of the solution. Section 7.2 lists avenues that were not explored during the development phase and improvements to the final solution. Lastly, Section 7.3 contains an overview of the developed work.

7.1 Difficulties

During the development of the solution, there were several challenges to overcome to build the solution that better fitted the proposed requirements. Some of these difficulties were mentioned in previous sections, some were solved and others will be mentioned in Section 7.2 (p. 72).

The first difficulty was the modification of communication between Node-RED *nodes*. Since they originally used events, a solution exclusive to Javascript, and a centralized environment, changes were made to implement a decentralized alternative. One of the most challenging parts of this feature was the implementation of this communication for *sub-flows*. The way Node-RED implements this entity made it difficult to translate to the new way of communication. This was implemented but was seldom used in the constructed scenarios.

There were several difficulties when it comes to the devices' firmware. Early on in the development phase, it was noted the memory limitations of ESP8266 chips, which led to the construction of the FAIL-SAFE mechanism, after exploring several others. However, it was noted that if an assigned script passed a certain threshold, the device could not recover. This led to the exploration of several alternatives until the change in hardware from ESP8266 to ESP32. Besides

this difficulty, there were some problems regarding the differences between MicroPython Unix and ESP ports. This led to several changes in the developed firmware to allow compatibility in both environments.

Finally, the libraries used in the MicroPython firmware have some limitations, which were mentioned before in Section 5.2.1 (p. 41). These limitations consist of the MQTT client's inexistence of support for QoS.

7.2 Future Work

The solution developed during the course of this dissertation solved the more pressing issues identified in Section 4.1 (p. 35). However, the implementation contains limitations and introduces new problems, which can be expanded upon and solved in future work.

As mentioned previously in Section 6.2 (p. 54) experiments, the orchestrator is sensible to any change in the status of the devices. This characteristic leads the orchestrator to perform several orchestrations that are costly to the system. For example, one failed PING request should not lead to a complete (re-)orchestration of the system, since the device non-response may not mean total unavailability. Instead, there should be retries, where the system made sure that the device was indeed unavailable. This mechanism should be configurable by the user or based on existing algorithms used in distributed systems.

Currently, every time the orchestrator starts the assignment process, the system stops working until the process is finished. One optimization that would greatly increase the system's availability would be to implement a way of (re)orchestrating without forcing the system to stop. For example, instead of sending a script that contains all the *nodes*' code, send snippets for each node and each device would be responsible for adding it to its execution script. However, this is not a trivial problem and more exploration would have to be made.

The *node* assignment method used in the developed solution uses a greedy algorithm to assess the best device for each *node*. As mentioned in Section 5.2.2.5 (p. 47), this has limitations that can lead to impossible assignments. The assignment process could be greatly improved with the use of better algorithms, for example by using SAT solving algorithms. This improvement can lead to several different solutions, each with their advantages.

Lastly, to validate the current solution, only a small portion of *nodes* were developed that support MicroPython code generation. Besides the increase in the number of nodes that support this, it would be interesting if the system was expanded to support different types of devices' firmware. For example, supporting code generation for C and Lua.

We can conclude that the developed tool has space for improvement, not only in the expansion to new firmware and environments but also in its optimization and enrichment.

7.3 Conclusions

As the number of devices connected to the internet increases, it is important to leverage their capabilities and modify the way systems are built to take advantage of these resources. It is also important to allow end-users with no programming experience to build Internet-of-Things (IoT) systems, with the use of visual programming tools. These tools make the building process easier, reducing the need for knowledge of programming concepts.

Despite the existence of a considering number of visual programming tools applied to IoT, the majority of these tools are centralized. This centralization hinders the resiliency of the system, as the unit responsible for the execution of most or all of the computation is a single point of failure. If this unit or the network fails, the system stops being functional. In addition to this, there are several computational resources in the devices that are not being utilized by the system.

During the analysis of the state of the art, some issues and missing features were identified, which this dissertation aims to correct. The tools found that possess a decentralized architecture have limiting characteristics such as assumptions about what is a constrained device regarding computational capabilities, lack of open source licenses, and simplification of the approach taken to the decomposition and assignment of tasks.

The developed solution solves these issues by expanding an already popular visual programming tool, Node-RED, with a decentralized approach that focuses on leveraging all the devices, even ones that only support the execution of simple blocks of code. Node-RED was modified to allow communication between *nodes*, even in different devices, as well as support for *node*'s code generation and orchestration of computations. On the devices' side, firmware was developed that allows it to receive scripts of code for later execution as well as communicate its status.

Several mechanisms were built to deal with the devices' instability. The developed solution manages the state of the devices, triggering a new orchestration if any device becomes unavailable or if a new device announces itself. Besides this, the system handles possible memory constraints of the system, assigning fewer *nodes* to devices limited by memory capacity.

This dissertation contributes with a decentralized IoT system that is robust, elastic, and overall efficient.

Appendix A

Scenario 2 Results

Start	End	Delta
1591876328759	1591876328770	11
1591876329440	1591876329448	8
1591876329991	1591876329994	3
1591876330539	1591876330554	15
1591876331106	1591876331120	14
1591876331658	1591876331667	9
1591876332192	1591876332200	8
1591876332710	1591876332721	11
1591876333222	1591876333237	15
1591876333779	1591876333787	8

Table A.1: Node-RED original results

Start	End	Delta
1591877265187	1591877265346	159
1591877266172	1591877267055	883
1591877267564	1591877267698	134
1591877268318	1591877268955	637
1591877269424	1591877269783	359
1591877270361	1591877271117	756
1591877271635	1591877272012	377
1591877272630	1591877273132	502
1591877273645	1591877273996	351
1591877274541	1591877275277	736

Table A.2: Node-RED + MQTT results

Start	End	Delta
1591877987030	1591877988695	1665
1591877989911	1591877991177	1266
1591877992272	1591877993595	1323
1591877994286	1591877995817	1531
1591877996305	1591877997618	1313
1591877998049	1591877999307	1258
1591877999734	1591878001322	1588
1591878001638	1591878002855	1217
1591878003397	1591878004643	1246
1591878005113	1591878006703	1590

Table A.3: Node-RED modified + Dockers (same host) results

Start	End	Delta
1591908868087	1591908870410	2323
1591908871443	1591908873803	2360
1591908874380	1591908877085	2705
1591908877629	1591908880338	2709
1591908880878	1591908883937	3059
1591908884472	1591908887147	2675
1591908887651	1591908889096	1445
1591908889803	1591908892200	2397
1591908892693	1591908894158	1465
1591908894846	1591908897623	2777

Table A.4: Node-RED modified + Dockers (different host) results

Start	End	Delta
1591904836329	1591904840130	3801
1591904844918	1591904849155	4237
1591904850127	1591904854579	4452
1591904855324	1591904859754	4430
1591904860483	1591904864559	4076
1591904865164	1591904869180	4016
1591904869770	1591904873905	4135
1591904874557	1591904878706	4149
1591904879318	1591904882934	3616
1591904888813	1591904893230	4417

Table A.5: Physical + MQTT results

Start	End	Delta
1591878582050	1591878587990	5940
1591878589026	1591878593492	4466
1591878594105	1591878598640	4535
1591878599238	1591878603841	4603
1591878604570	1591878608765	4195
1591878609340	1591878614220	4880
1591878615030	1591878620187	5157
1591878620870	1591878625038	4168
1591878625718	1591878630967	5249
1591878631560	1591878635880	4320

Table A.6: Node-RED modified + MQTT + Physical + Firmware

Appendix B

Paper Submitted

This appendix includes the paper submitted to the 2020 Concurrency and Computation: Practice and Experience Journal.

RESEARCH ARTICLE

Visual Programming and Orchestration in the Internet-of-Things: A Systematic Literature Review

Margarida Silva¹ | João Pedro Dias^{*1,2} | André Restivo^{1,3} | Hugo Sereno Ferreira^{1,2}¹DEI, Faculty of Engineering, University of Porto, Porto, Portugal²INESC TEC, Porto, Portugal³LIACC, Porto, Portugal**Correspondence**

*João Pedro Dias, DEI - FEUP, Rua Dr. Roberto Frias, Porto, Portugal Email: jpmdias@fe.up.pt

Funding Information

This research was supported by the Portuguese Foundation for Science and Technology (FCT), Grant Number: SFRH/BD/144612/2019

Summary

Internet-of-Things (IoT) systems are considered one of the most notable examples of complex, large-scale systems. Some authors have proposed visual programming solutions to address part of this inherent complexity, but most of these systems depend on a centralized unit to carry out most – if not all – the orchestration between devices and system components, hindering the degree of scalability and distribution that can be attained. In this work, we carry out a systematic literature review of the current solutions that provide visual and decentralized orchestration to define and operate IoT systems. Our work reflects upon a total of 29 proposals that address these issues up to a certain degree. We provide an in-depth discussion of these works, and find out that only four of these solutions attempt to tackle this issue as a whole, though still leaving a set of open research challenges. We finally argue that this challenges, if addressed, could make IoT systems more fault-tolerant, with impact on their dependability, performance, and scalability.

KEYWORDS:

Internet-of-Things, Orchestration, Visual Programming, Decentralized Computation, Large-Scale Systems

1 | INTRODUCTION

The Internet-of-Things (IoT) is composed of a myriad of devices, having a wide range of capabilities that are connected to the Internet directly or indirectly, allowing to transfer, integrate, analyze and act according to data generated by themselves¹. IoT systems agglomerate both sensing and actuating devices at an unprecedented scale[†], with applications ranging from mission-critical systems to entertainment and commodity solutions³.

The widespread usage of IoT across application domains, designed and built by different and, even, competitive, manufacturers, led to a mostly uncontrollable and ever-growing heterogeneity of devices, differing in several aspects such as computational power, protocols, and architectures. Additionally, the large-scale and distributed (geographically and logically) nature of IoT makes them highly-complex systems. These factors lead to several issues in developing these systems, as well as guaranteeing their scalability, maintainability, security, and dependability⁴. These factors prompted the need for tools allowing users that have reduced technical knowledge to configure and adapt their systems to their needs (from manufacturing floor automation to *smart home* system customization), leading to the birth of several different *low-code* solutions, that try to reduce the inherent complexity of programming and configuring these systems⁵. Visual Programming Languages (VPL) and similar visual programming

[†] According to Siemens, 26 billion physical devices are part of the Internet as of 2020, and predictions are pointing at 75 billion in 2025².

approaches, either model-based or mashup-based⁶, allow the user to configure the system by using and arranging visual elements that and then translate them into code⁷. It provides the user with an intuitive and straightforward interface for coding at the possible cost of losing expressive power. Different languages have different focuses, such as education, video game development, 3D construction, system design, and, of course, Internet-of-Things development and configuration⁸. As an example of one of the latter, we have Node-RED[‡], which is one of the most used open-source visual programming tools⁹, providing both a visual editor and a runtime environment for Internet-of-Things systems.

Most mainstream visual programming solutions focused on Internet-of-Things, Node-RED included, have a centralized approach (which can be *on-premises* or cloud-based), where the main component executes most of the computation on data provided by edge (*i.e.*, sensors and actuators) and fog devices, and other third-party services. There are several consequences of this approach: (1) it introduces a single point of failure, (2) local data is being transferred across boundaries (*e.g.*, private, technological, and political) either without a need (privacy) or even in violation of legal constraints (*e.g.*, General Data Protection Regulation)^{10,11}, and (3) computation capabilities of the lower-tier — edge and fog — might be being wasted. There has been an increasing research effort put in *Fog Computing* and *Edge Computing*¹². Both concepts focus on using the resources available in lower-tier devices to improve the overall dependability, performance, and scalability. However, these efforts are still in their early stages, and manifestos such as the Local-First¹³ (*i.e.*, data and logic should reside locally, independent of third-party services faults and errors) and NoCloud¹⁴ (*i.e.*, on-device and local computation should be given priority over cloud service computation) have not yet received enough attention from researchers.

In this paper, we present a systematic review of the current literature on visual programming solutions for IoT with a particular focus on the ones which take into account orchestration of multiple system parts. Our initial search yielded a total of 2698 results across three different scientific databases. These results were selected, according to our defined *inclusion* and *exclusion* criteria, leaving us with 21 papers. We proceeded to enhance our selection by a mixture of snowballing, taking into account previous (mostly non-systematic) surveys and adding eight solutions not found during the search process. Our result provided a total of 28 solutions, presented across 22 papers. We proceeded to compare them regarding a selection of characteristics, including scope, architecture, scalability, and visual programming paradigms. We then carried out an in-depth analysis on the subset of the solutions that provided mechanisms for decentralized orchestration.

This remaining of this paper is structured as follows: Section 2 gives an overview of some key background concepts, Section 3 presents the methodology used in this research, Section 4 presents the results of the literature review, followed by an in-depth analysis of the current alternative for visual IoT decentralized orchestration in Section 5. An overview of the current issues and research challenges, along with some final remarks, is given in Section 6.

2 | BACKGROUND

2.1 | Internet-of-Things

Internet-of-Things paradigm is defined by the committee of the International Organization for Standardization and the International Electrotechnical Commission¹⁵ as:

"An infrastructure of interconnected objects, people, systems and information resources together with intelligent services to allow them to process information of the physical and the virtual world and react."

IoT systems are, mostly, networks of heterogeneous devices attempting to bridge the gap between people and their surroundings. According to Buuya¹⁶, the applications of IoT systems can be divided into four categories: (i) *Home* at the scale of a few individuals or domestic scenarios, (ii) *Enterprise* at the scale of a community or larger environments, (iii) *Utilities* at a national or regional scale and (iv) *Mobile*, which is spread across domains due to its large scale in connectivity and scale.

One might think that IoT only relates to machines and interactions between them. Most of the devices we use in our day-to-day — *e.g.*, mobile phones, security cameras, watches, coffee machines — are now computation capable of making moderately complex tasks and are continually generating and sending information. This relates to the *human-in-the-loop* concept, where humans and machines have a symbiotic relationship¹⁷.

[‡]Node-RED, <https://nodered.org/>

2.2 | IoT architectures

Internet-of-Things systems deal with big amounts of data from different sources and have to process it in an efficient and fast fashion. Typical IoT systems are composed of three tiers, which are:

Cloud Tier mostly composed of data centers and servers, normally running remotely. It is characterized by having high computation power and latency.

Fog Tier composed of gateways and devices that are normally between the cloud servers and the edge devices. This tier has less latency than the cloud, more heterogeneity, and, typically, is more geographically distributed.

Edge Tier composed of all the peripheral devices (*e.g.*, sensors, embedded systems, light sources and air conditioners). These devices have several limitations in computational capabilities, but have less latency.

Complementary to these tiers, we can also partition IoT systems into an Application Layer, a Network Layer, and a Perception Layer¹⁸. At first sight, these might seem compatible with the tiers mentioned above (in the same order); however, not all devices in each tier map to their respective layer. One example is a third-party service that gives readings. It can be contained in the Perceptive Layer, but it is not included in the Edge Tier.

New paradigms of computing appeared related to each of these tiers. The majority of IoT systems use a Cloud Computing architecture, taking advantage of centralized computing and storage. This approach has several benefits, such as increased computational capabilities and storage, as well as easier maintenance. However, it comes with several problems such as (1) high latency, and (2) high use of bandwidth, due to the need to send the data generated from the sensors to the centralized unit¹⁹. Systems that only use cloud computing face several challenges²⁰, especially real-time applications, which are sensitive to increased latency. With the increasing computation capabilities of edge devices and the requirement of reduced latency, two new paradigms appeared: Fog Computing and Edge Computing.

2.2.1 | Fog Computing

With the improvement of wireless technologies and the increasing computational power (and reducing costs) of lower-tier devices (*i.e.*, fog and edge), it became possible to improve the computational execution of IoT systems. By not depending so much on the cloud tier, communication and resource sharing between devices can occur with lower latency. The central coordinator (on-premises or cloud-based), which in Cloud Computing was responsible for all the computation, now serves as a scheduler and state manager of the communication between devices, occasionally providing necessary resources. This new paradigm, where fog and edge devices are leveraged as computational entities (and not only merely sensing, actuating and gateway devices in the network) is called Fog Computing, which aims to bring computing closer to the perception tier, bringing the computation nearer to the edge of the network²¹. It focuses on distributing data throughout the IoT system, from the cloud to the edge devices, making the system distributed.

According to Buuya¹, Fog Computing has several advantages: (1) reduction of network traffic by having edge devices filtering and analyzing the data generated and sending data to the cloud only if necessary, (2) reduced communication distance by having the devices communicate between them without using the cloud as a middleman, (3) low-latency by moving the processing closer to the data source rather than communicating all the data to the cloud for it to be processed, and (4) scalability by reducing the burden on the cloud, which could be a bottleneck for the system.

Despite all the advantages, Fog Computing has several requirements and difficulties. To make a successful and efficient distribution of computation and communication, it requires knowledge about the resources of the connected devices. The complexity is also more significant than Cloud Computing since it needs to work with heterogeneous devices with different capacities.

2.2.2 | Edge Computing

Edge Computing, also known as Mist Computing, is a distributed architecture that uses the devices' computational power to process the data they collect or generate. It takes advantage of the Edge tier, which contains the devices closer to the end-user, such as smartphones, TVs and sensors. The goal of this paradigm is to minimize the bandwidth and time response of IoT systems while leveraging the computational power of the devices in them. It reduces bandwidth usage by processing data instead of sending it to the cloud to be processed, which is also correlated to reduced latency since it does not wait for the server response. In addition to these advantages, and related to their cause, Edge Computing also prevents sensitive data from leaving the network, reducing data leakage and increasing security and privacy^{22,23}.

In this paradigm, each device serves both as a data producer and a data consumer. Since each device is constrained in terms of resources, this brings several challenges such as system reliability and energy constraints due to short battery life and overall security. Other issues consist of the lack of easy-to-use tools and frameworks to build cloud-edge systems, non-existent standards regarding the naming of edge devices and the lack of security edge devices have against outside threats such as hackers²⁴.

There is some confusion in the research community regarding the concepts of Fog and Edge computing. The publication from Iorga et al.²⁵ was used to inspire the definitions of these terms. Edge Computing focuses on executing applications in constrained devices, without worrying about storage or state preservation. On the other hand, Fog Computing is hierarchical and includes devices with more capabilities, capable of control activities, storage, and orchestration.

2.3 | Visual Programming Languages

Visual Programming, as defined by Shu²⁶, consists of using meaningful graphical representations in the process of programming. With this definition, we can consider Visual Programming Languages (VPLs) as a way of handling visual information and interaction with it, allowing the use of visual expressions for programming. According to Burnet and Baker²⁷, visual programming languages are constructed to *"improve the programmer's ability to express program logic and to understand how the program works"*.

There are several applications of visual programming languages in different areas, such as education, video game development, automation, multimedia, data warehousing, system management, and simulation, with this last area being the area with most use cases⁸.

Visual programming languages have several characteristics, such as a concrete process and depiction of the program, immediate visual feedback and require the knowledge of fewer programming concepts²⁷.

VPLs can be categorized by their visual paradigms and architecture²⁸:

Purely Visual Languages where the system is developed using only graphical elements and the subsequently debugging and execution is made in the same environment.

Hybrid text and visual systems where the programs are created using graphical elements, but their executions is translated into a text language.

Programming-by-example systems where a user uses graphical elements to teach the system.

Constraint-oriented systems where the user translates physical entities into virtual objects and applies constraints to them, in order to simulate their behaviour in reality.

Form-based systems which are based on the architecture and behaviour of spreadsheets.

The categories mentioned can be present in a single system, making them not mutually exclusive.

3 | SYSTEMATIC LITERATURE REVIEW METHODOLOGY

This work follows a Systematic Literature Review (SLR) methodology to gather information on the state of the art of visual programming applied to the Internet-of-Things paradigm, with a special emphasis on orchestration concerns. The goal of a systematic literature review is to synthesize evidence with emphasis on the quality of it²⁹.

During this SLR, a specific methodology was followed to reduce bias and produce the best results²⁹. We started by defining the research questions to be answered as well as choosing data sources to search for publications.

3.1 | Research Questions

To reveal the current practice, research and studies related to orchestration in the Internet-of-Things that leverage visual approaches, which enable us to find the current, and pending research challenges, we outline the following research questions (RQ):

- RQ1** *What relevant visual programming solutions applied to IoT orchestration exist?* Internet-of-Things is a paradigm with several years, and its integration with visual programming languages makes their development easier for the end-user. The tools that integrate these two paradigms are useful and reduce the overhead of programming or prototyping IoT systems.
- RQ2** *What is the tier and architecture of the tools found in RQ1?* IoT systems can belong to one or more of tiers — Cloud, Fog and Edge — as well as implement a centralized or decentralized architecture. A visual programming tool applied to IoT orchestration can be used to facilitate the development of systems that operate on these tiers. Each tier and type of architecture offers vantages and disadvantages, which are essential to understand the usages and characteristics of a system.
- RQ3** *What was the evolution of visual programming solutions applied to IoT orchestration over the years?* To understand the field of visual programming applied to IoT, more specifically, its orchestration, it is essential to perceive its evolution.

Answering these questions will provide insights that can be valuable for both practitioners — in terms of summarizing what the current practices on the usage of visual programming methodologies for IoT orchestration are — and researchers — showing current challenges and issues that can be further researched.

3.2 | Candidate Searching and Filtering

Our systematic literature review protocol followed the inclusion and exclusion criteria detailed in Table 1 and is outlined in Figure 1. To find the most relevant works for this study, three scientific databases were used, namely: *IEEE Xplore*, *ACM Digital Library* and *Scopus*. These electronic databases contain some of the most relevant digital literature for studies in the area of Computer Science, thus being considered reliable sources of information.

TABLE 1 Inclusion and exclusion criteria.

I/E	ID	Criterion
Exclusion	EC1	Not written in English.
	EC2	Presents just ideas, tutorials, integration experimentation, magazine publications, interviews or discussion papers.
	EC3	Presents a tool, framework or approach that does not support the orchestration of multiple devices.
	EC4	Has less than two (non-self) citations when more than five years old.
	EC5	Duplicated articles.
	EC6	Articles in a format other than camera-ready (PDF).
Inclusion	IC1	Must be on the topic of visual programming in Internet-of-Things.
	IC2	Contributions, challenges and limitations are presented and discussed in detail.
	IC3	Research findings include sufficient explanation on how the approach works.
	IC4	Publication year in the range between 2008 and 2019.
	IC5	Is a survey that focus visual programming in IoT or

We begun our search in these data sources using a query that captured the most probable keywords to appear in our target candidates, namely *visual programming*, *node-red*, *dataflow*, and *Internet-of-Things*. This led us to specify variants of the following query that are understood by the mentioned databases: ((vp1 OR visual programming OR visual-programming) OR (node-red OR node red OR nodered) OR (data-flow OR dataflow)) AND (IoT OR Internet-of-Things OR internet-of-things). This search was performed in October of 2019 and the number of results produced can be seen in the first step of Figure 1.

The evaluation process of the publications then followed eight steps with specific purposes:

1. **Automatic Search:** Run the query string in the different scientific databases and gather results;

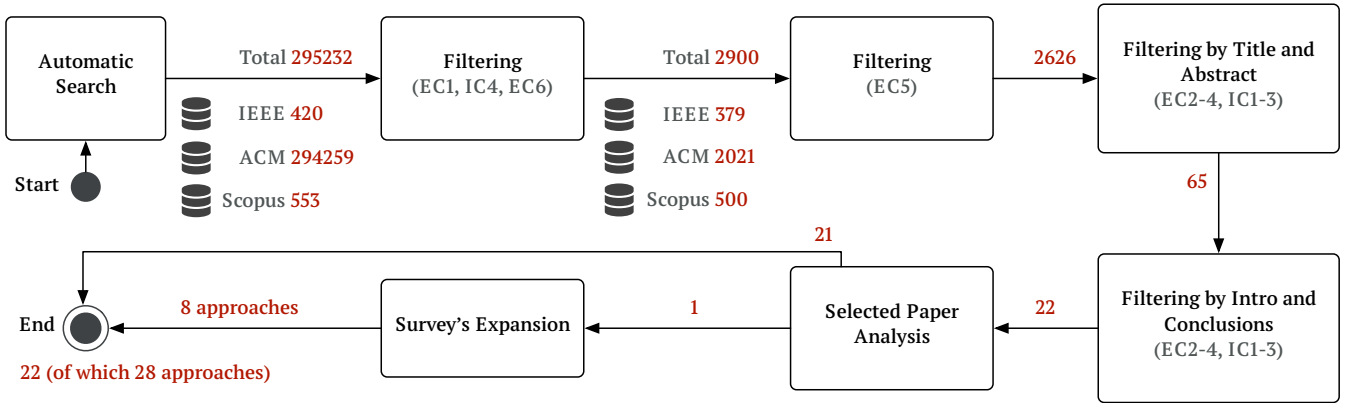


FIGURE 1 Pipeline overview of the SLR Protocol.

2. **Filtering (EC1, IC4, and EC6):** Publications are selected regarding its (1) language, being limited to the ones written in English language, (2) publication date, being limited to the ones published between 2008 and 2019, and (3) publication status, being selected only the ones that are published in their final versions (camera-ready PDF format);
3. **Filtering to remove duplicates (EC5):** The selected papers are filtered to remove duplicated entries;
4. **Filtering by Title and Abstract (EC2–EC4, and IC1–IC3):** Selected papers are revised by taking into account their *Title* and *Abstract*, by observing the (1) stage of the research, only selecting papers that present approaches with sufficient explanation, some experimental results and discussion on the paper contributions, challenges and limitations, (2) contextualization with recent literature, filtering papers that have less than two (non-self) citations when more than five years old, and (3) leverages the use of visual notations for orchestrating and operating multi-device systems.
5. **Filtering by Introduction and Conclusions (EC2–EC4, and IC1–IC3):** The same procedure of the previous point is followed but taking into consideration the *Introduction* and *Conclusion* sections of the papers;
6. **Selected Papers Analysis:** Selected papers are grouped, and surveys are separated; their content is analyzed in detail.
7. **Surveys Expansion:** For the survey papers found, the enumerated solutions are analyzed and filtered taking into account their scope and checking if they are not duplicates of the current selected papers.
8. **Wrapping:** Approaches and solutions gathered from the *Selected Papers Analysis* (individual papers) and from the *Survey Expansion* are presented and discussed.

The total number of publications was 2698, and, after the evaluation process, 22 publications were selected as can be seen in Figure 1. From those, one was a survey and the others presented approaches relevant to our research questions.

4 | LITERATURE REVIEW RESULTS

After analyzing the 22 publications, we organized them by categories; of these, one was a survey⁸, and the remaining 21 were papers describing papers that address our research questions. In that survey, the authors make an in-depth review of 13 visual programming languages in the field of IoT, comparing them using four attributes: (1) programming environment, (2) license, (3) project repository and (4) platform support. We used this survey to complement our research in Section 4.2.

4.1 | SLR Results

The selected 21 articles described approaches that use visual programming in the IoT context having orchestration considerations. One of the tools is described in two papers, which showcases its evolution. The 20 unique solutions are:

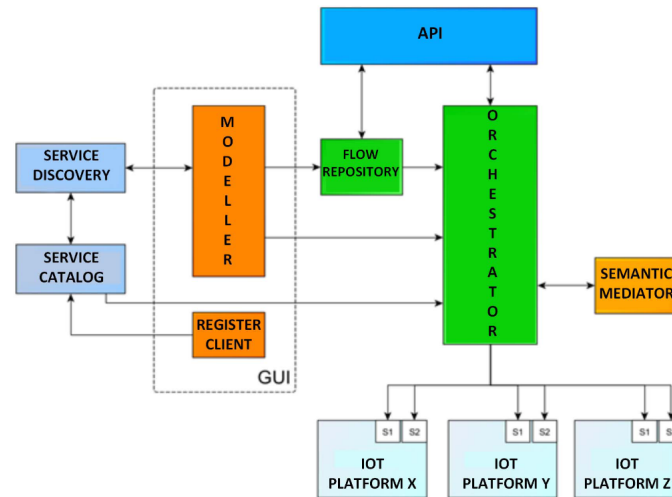


FIGURE 2 Belsa et al.³⁰ solution architecture. The Modeller is a Node-RED flow editor canvas where new flows can be created by connecting nodes that correspond to the available services (Service Catalog and Discovery) which are then stored in the Flow Repository. The Orchestrator is responsible for managing and running the specified flows as required (by running several instances of the Node-RED runtime) and converting Node-RED calls to the different IoT Platforms *native calls* (aided by the Semantic Mediator).

- **Belsa et al.**³⁰ present a solution for connecting devices from different IoT platforms, using Flow-Based Programming with Node-RED, depicted in Fig. 2. Its motivation is based on the limitation imposed by the IoT platform on communication between components and extensibility, which limits the possibility to interact with other platforms' services. To validate their solution, they implemented a use case in the domain of transportation and logistics, with a service that uses five different types of applications. The developed tool offers access to available services in a centralized visual framework, where end-users can use them to build more complex applications.
- **Ivy**³¹ proposes the next step toward visualization applied to IoT with a visual programming tool that allows its users to link devices, inject logic, and visualize real-time data flows using immersive virtual reality. It provides the end-users with an immersive virtual reality that allows them to visualize the data flow, access to debugging tools, and real-time deployment. Each programming construct called node - data flow architecture - has a distinct shape and colour, which facilitates the understanding of the system being built or debugger for the user. The experiences made to validate the prototype were positive, with the participants being receptive to Ivy and proposing new use cases.
- **Ghiani et al.**³² proposition is to build a collection of tools that allow non-developer users to customize their Web IoT applications using trigger-actions rules. The proposed solution provides a web-based tool that allows users to specify their trigger-action rules using *IFTTT*, as well as a context manager middleware that can adapt to the context and events of the devices and apply rules to the system. To validate the developed tool, an example home automation application that displays sensor values and directly controls appliances were built. The results were, for the most part, positive, and the issues found are related to usability and visual clues.
- **ViSiT**³³ uses the jigsaw puzzle metaphor³⁴ to allow its end-users to implement a system of connected IoT devices. It provides a web-based visual tool connected with a web-service that, given a jigsaw representation, generates an executable implementation. Their goal is achievable by adapting model transformations used by software developers into intuitive metaphors for non-developers to use. They validated the developed tool with a usability evaluation, which was overall positive, with a significant percentage considering the tool useful and providing real-life scenarios where they could implement it.
- **Valsamakis and Savidis**³⁵ propose a framework for Ambient Assisted Living (AAL) using IoT technologies, which allows for customized automation. It uses visual programming languages to facilitate their end-users - carers, family, friends, elderly - to build and modify automation. They built a visual programming framework that introduces smart

objects grouping in tagged environments and real-time smart-object registration through discovery cycles. It runs on typical smartphones and tablets and is built in Javascript, allowing it to run in browsers. Their future work focuses on integrating different visual programming paradigms to accomplish the requirements of the end-user fully.

- **WireMe**³⁶ is a solution for building, deploying, and monitoring IoT systems, built with non-developer end-users in mind but also extensible for advanced users to build over it. The developed solution makes use of Scratch, a visual programming interface, to provide its users with a customizable dashboard where they can monitor and control their IoT system as well as program automation tasks. It has a Main Control Unit responsible for communicating the device's status to the dashboard via MQTT, which is programmable using their visual interface and Lua programming language. Their tool was validated in an empirical study with students around 16 years old and engineering students without programming experience. The results were not positive, with some students not being able to create the required simple logic. Future work consists of improving programming blocks to become more intuitive.
- **VIPLE**³⁷, Visual IoT/Robotics Programming Language Environment, is a new visual programming language and environment. It provides an introduction to topics such as computing and engineering and tools for more technical domains like software integration and service-oriented computing. It focuses on complex concepts such as robot as a service (Raas) units and Internet of Intelligent Things (IoIT) while studying the programming issues of building systems classified as such. The developed tool has been tested and used in several universities since 2015 due to its large set of features and use cases.
- **Smart Block**³⁸ is a block-based visual programming language and visual programming environment applied to IoT systems, that allows non-developer users to build their systems quickly. Their solution is specific to the home automation domain, like Smart Things. The language was designed using IoTa calculus, used to generalize Event-Condition-Action rules for home automation. The environment was built using a client-side Javascript library called Blockly, which allows for the creation of visual block languages. Future work for this project consists of supporting device grouping and security by expanding custom blocks, as well as extending the tool for other domains besides home automation.
- **PWCT**³⁹ is a visual programming language applied to build IoT, Data Computing, and Cloud Computing systems. Its goal consists of reducing the cost of development of these types of systems by providing a comfortable and more productive development tool. The language was meant to compete with text-based languages such as Java and C/C++. It makes use of graphical elements to replace code. It has three main layers: (1) the VPL layer, composed of graphical elements, (2) the middleware layers, responsible for connecting the VPL layer to the system's view, which is the (3) System Layer, responsible for dealing with the source code generated by the first layer. The created solution received positive feedback from the community, with more than 70,000 downloads and 93% of user satisfaction.
- **DDF**⁴⁰ is a Distributed Dataflow (DDF) programming model for IoT systems, leveraging resources across the Fog and the Cloud. They implemented a DDF framework extending Node-RED, which, by design, is a centralized framework. Their motivation comes from the possibility to develop applications from the perspective of Fog Computing, leveraging these devices for efficiency and reduced latency, since there is a significant amount of resources such as edge devices and gateways in IoT systems. They evaluated their prototype using a small scale evaluation, which was positive. The results showed that their DDF framework provides an alternative for designing and developing distributed IoT systems, despite having some open issues such as not having a distributed discovery of devices and networks.
- **GIMLE**⁴¹, Graphical Installation Modelling Language for IoT Ecosystems, is a visual language that uses visual elements to model domain knowledge using significant ontological requirements. The goal of this language is to fill the gap of modeling requirements on the physical properties of IoT installations by proposing a new process for configuring industrial installations. It makes use of flow-based and domain-based visual programming to isolate the requirements' logical flow from their details. The developed tool supports reuse within the models, which is valuable due to the repetitive nature of industrial installations. However, it still needs to clarify its scope within the current practice and its use in production settings.
- **DDFlow**⁴² is a macro-programming abstraction that aims to provide efficient means to program high quality distributed apps for IoT. The authors point to a lack of solutions for complex IoT systems programming, causing developers to build their systems, which leads to a lack of portability/extensibility and results in a lot of similar systems that do the same

thing but are “different” because different programmers created them. Developers use Node-Red to specify the application functionalities, and DDFlow handles scalability and deployment. The authors describe DDFlow’s goal to allow developers to formulate complex applications without having to care about low-level network, hardware, and coordination details. This is done by having the DDFlow accompanying runtime dynamically scaling and mapping the resources, instead of the developer. DDFlow gives developers the possibility to inject custom code on nodes and has custom logic if the available nodes are not enough for some tasks.

- **Kefalakis et al.**⁴³ proposition consists of a visual environment that operates over the OpenIoT architecture and allows for the development of IoT applications with reduced programming effort. Modeling IoT services with the developed tool is made by specifying a graph that corresponds to an IoT application, which can be validated and have its code generated and performed over the OpenIoT middleware platform. It aims to fill the gap of tools that provide support for the development and deployment of integrated IoT applications. The approach taken presents several advantages: (1) it leverages standards-based semantic model for sensor and IoT context, making it easier to be widely adopted, (2) it is based on web-based technologies which open the possibilities of applications from developers and (3) it is open source.
- **Eterovic et al.**⁴⁴ propose an IoT visual domain-specific modeling language based on UML, with technical and non-technical users in mind. The authors defend that, with the evolving nature of IoT, the future end-user will be a non-technical person, with no programming knowledge. Attaining the issues that this can create in the future, it is crucial to create a visual language easy enough to be understood by non-technical people but expansible enough to represent complex systems. To evaluate the proposed solution, they invited 11 users of different levels of UML expertise to model a simple IoT system with the developed language. The System Usability Score was positive, as well as the Tasks Success Rate. Despite the positive score, some future actions would be the testing of the language with a more complex task as well as the integration of advanced UML notations.

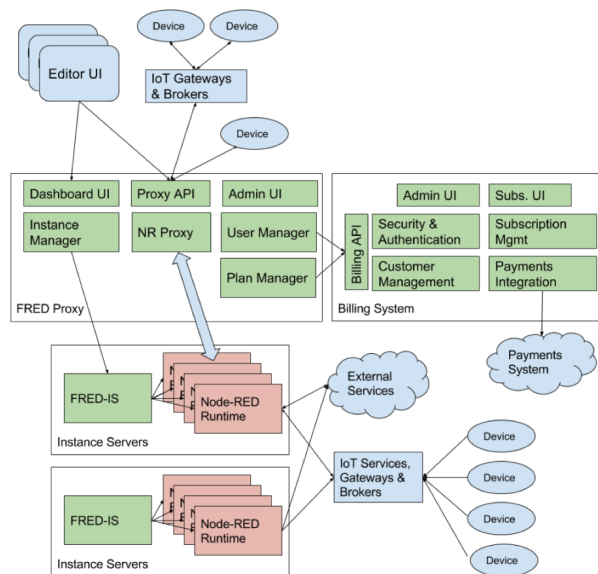


FIGURE 3 FRED⁴⁵ high-level architecture. FRED is based on Node-RED and addresses the limitation of running several flows in parallel (multiple runtimes) by orchestrating several instances of Node-RED (FRED-IS) using their FRED Proxy.

- **FRED**⁴⁵ is a frontend for Node-RED, a development tool that makes it possible to host multiple Node-RED runtimes (Fig. ??). It can be used to connect devices to services in the cloud, manage communication between devices, create new web app applications, APIs and event-integrated services. To provide all these features, FRED allows the running of flows for multiple users, in which all flows get fair access to resources such as CPU, memory, storage, as well as secure access to flow editors and the flow runtime. The authors concluded that FRED is useful for users learning about Node-RED and allows users to prototype cloud-hosted applications rapidly.

- **WoTFlow**⁴⁶ is proposed as a cloud-based platform that aims to provide an execution environment for multi-user cloud environments and individual devices. It aims to take advantage of data flow programming, which allows parts of the flow to be executed in parallel in different devices. Based on this, the tool will take advantage of the ability to split and partition the flows and distribute them by edge devices and the cloud. The state of the developed tool was in the early stages, with future expansions based on the use of optimization heuristics, automatic partitioning based on calculated constraints, security, and privacy.
- **Besari et al.**^{47,48} proposes an IoT-based GUI that aims to control sensors and actuators in an IoT system using an android application, in which the users use a visual programming language to configure and interact with the IoT system. The system was tested with a Pybot, a robot that is programmable like an IoT system, with sensors and actuators. After testing and evaluating the system, the authors came to a score of 72.917 (out of 100) for the Pybot software, which is considered “good”. The overall acceptability of the system was “ACCEPTABLE”, which led the authors to consider the application accepted by users.
- **CharIoT**⁴⁹ is a programming environment that promises its end-users a solution that unifies and supports the configuration of IoT environments. It provides three blocks of support: capturing higher-level events using virtual sensors, construction of automation rules with a visual overview of the current configuration and support for sharing configuration between end-users using a recommendation mechanism. Two types of virtual sensors were developed to capture higher-level events. The programmed virtual sensor provides more accessible and understandable abstractions (defining that a room is “cold” if the temperature is below 20°C). The demonstrated virtual sensors are more complex, requiring the user to provide a demonstration of the occurrence and lack of occurrence of the event (for example, the event of someone knocking on the door and the absence of someone knocking on the door). This last one requires the training of a Random Forest classifier. This programming environment is similar to IFTTT but goes one step further, with smarter event capturing and reusing of configurations, allowing the end-user to build faster and more robust IoT installations.
- **Desolda et al.**⁵⁰ proposition uses a tangible programming language that allows non-programmers to configure smart objects’ behaviour to create and customize smart environments. The main goal was to create, with the developed technology, a scenario of a smart museum. The authors defend that the synchronization of smart devices cannot limit the personalization of a smart environment, and it may require experts to build their narrative, much like a museum said. With this in mind, they introduced custom attributes to assign semantics to connected objects to empower and simplify the creation of event-condition-action rules. This is ongoing research focused on developing new technology with an interaction paradigm that supports the input of domain experts in the creation of smart environments. The fact that this technology uses expensive material (tabletop surface as a digital workspace) does not allow a regular user to use it, as stated in the introduction.
- **Eun et al.**⁵¹ proposes an End-User Development (EUD) tool that allows users to develop their applications. It uses the dataflow approach, which allows for a more generalized programming experience as well as the facility to build more complex programs with simple modules. The proposed tool has three main components: Service Template Authoring Tool, Service Template Repository, and Smartphone Application. The first one allows the end-user to build more complex methods using atomic templates (components with simple functionality, like opening a curtain if it receives a command). The Service Template Repository contains the proprietary atomic templates as well as ones built by the user. Lastly, the Smartphone Application runs and manages the applications built by the user, as well as their requirements and dependencies. The developed EUD tool was compared with *IFTTT* and Zapier; other tools focused on end-user development. *IFTTT* and the developed tool are similar, focusing on consumer development, IoT, and home environments, with Zapier focusing on business environments. Both Zapier and *IFTTT* use the Trigger-Action paradigm (TAP), which differs from the dataflow paradigm used in this paper’s tool.

4.2 | Complementary Results

The results of the Systematic Literature Review were disclosed in the previous section. However, some tools were found in a non-systematic survey⁸ that are not present in the selected papers. We consider that this divergence may result from tools that have no academic publications associated with them, thus not being present in the publication databases mentioned in Section 3.2. One famous example is *Node-RED*⁵². The results from the survey⁸ were analyzed, the described tools were assessed against the

evaluation process defined in Section 3 and characterized by the categories mentioned in Subsection 4.1. Using the methodology described, the results are:

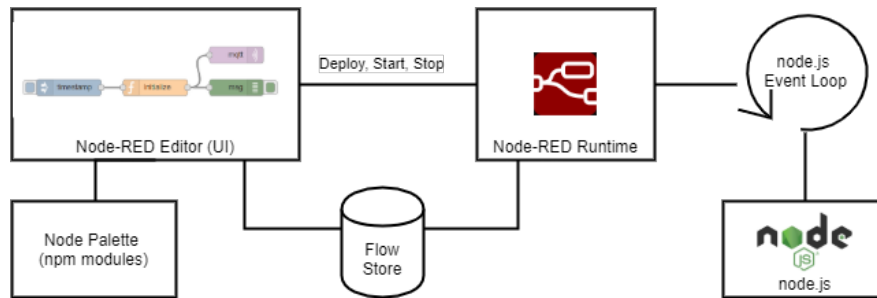


FIGURE 4 Node-RED⁵² high-level architecture, identifying its development interface, runtime and `node.js` dependency. The *flows* can be versioned and organized in projects and new modules (*i.e.*, *nodes*) can be added using the `node.js` dependency manager tool (*i.e.*, `npm`).

- **Node-RED**⁵² is a visual programming environment applied to the IoT paradigm. It makes use of flow-based development (connecting communication and computation *nodes* in *flows*), supporting a wide range of devices and APIs. It has two main modules: (1) a development interface which consists of a flow drawing canvas and a *node* palette, and (2) a runtime module that leverages the Node.JS event-loop to pass messages between the different *nodes*[?]. Due to being open-source and extendable, its large community contributes with features that enrich the tool, some of them talked about in Subsection 4.1 (*e.g.*, FRED⁴⁵ and DDF⁴⁰).
- **NETLab Toolkit**⁵³ is a visual environment that makes use of *drag-and-drop* actions to allow users to build IoT applications. It provides a web interface to connect sensors, actuators, and others for quick prototypes.
- **NooDL**⁵⁴ is a platform that provides a visual programming interface for prototyping applications. It allows for the creation of interfaces, using live data, and supporting several types of hardware. Although it is not specific to IoT, NooDL covers the programming of IoT systems. It makes use of MQTT broker agents for connecting devices and visual paradigms such as *nodes*, *connections*, and *hierarchies* to allow the user to build its system.
- **DGLux5**⁵⁵ for DSA is a *drag-and-drop* visual language and environment that allows its users to build applications tailored for Distributed Services Architecture (DSA) IoT middleware. It provides a dashboard for analyzing and controlling device data in real-time and builds the system only using visual elements.
- **AT&T Flow Designer**⁵⁶ is a visual tool incorporated in a cloud development environment, applied to the development of IoT systems. Its visual paradigm is similar to Node-RED, with the notion of *nodes* and *wires*. This tool provides an easy iteration and improvement of a product, as well as an easy deployment.
- **GraspIO**⁵⁷ is a Graphical Smart Program for Inputs and Outputs that contains a block *drag-and-drop* visual paradigm that allows its users to build applications for the *Cloudio* hardware. It offers a Cloud Service that connects and manages all *Cloudio* devices, making them available at the user's mobile device.
- **Wylidrin**⁵⁸ is a browser-based visual programming environment that allows the development of IoT systems of several devices, such as Raspberry Pi, Arduino, Intel Galileo, Intel Edison, and others. It provides a *drag-and-drop* environment, as well as support for text-based languages. A dashboard for visualizing the data collected is provided.
- **Zenodys**⁵⁹ provides a *drag-and-drop* interface to build application backends as well as user interfaces. Its computing engine can run in several types of devices, from the cloud to chips, devices, and distributed computers. Zenodys contains a visual debugger as well as support for text-based programming and code generation.

4.3 | Results Categorization

The mentioned frameworks and tools were divided into the following categories, according to several characteristics:

1. **Scope.** Some tools have specific use cases in mind. Therefore, knowledge of the scope of a tool is useful to assess if it solves a problem or fills a specific gap in the literature. Example values consist of *Smart Cities*, *Home Automation*, *Education*, *Industry* or *Several* if there is more than one.
2. **Architecture.** Visual programming tools applied to the Internet-of-Things can have a centralized or decentralized architecture, based on their use of Cloud, Fog or Edge Computing architecture. Possible values are *Centralized*, *Decentralized* and *Mixed*.
3. **License.** The license of software or tool is essential in terms of its usability. Normally, an open-source software reaches a bigger user base and allows them to expand and contribute to it. Possible values are the name of the tool license or N/A if it does not have one.
4. **Tier.** IoT systems, as explained in Section 2.2 is composed of three tiers - *Cloud*, *Fog* and *Edge*. A tool can interact in several of these tiers, which shapes the features it contains and how it is built.
5. **Scalability.** Defines how the tool or framework scales. It can be calculated based on metrics used to test the performance of the system. In this case, we considered scalability in terms of number and different types of devices supported. Possible values are *low*, *medium*, *high* or N/A, in case there is no sufficient information.
6. **Programming.** According to Downes and Boshernitsan²⁸ and also mentioned in Section 2.3, visual programming languages can be classified in five categories: (1) Purely Visual languages, (2) Hybrid text and visual systems, (3) Programming-by-example systems, (4) Constraint-oriented systems and (5) Form-based systems. These classifications are not mutually exclusive. It is important to know which type, so that might be possible to assess the type of experience the tool provides to the user and its architecture.
7. **Web-based.** Defines if the visual programming language and/or environment can be used in a browser. It is useful in terms of the usability of the tool.

The resulting categorization of the SLR results is depicted in Table 2. Some key take-ways are easily observable, namely: (1) most tools use a centralized architecture, (2) the hybrid visual-textual programming paradigm is predominant, and (3) most of the tools are web-based. The extended search findings and their categorization is presented in Table 3, following the same previously defined categories.

4.4 | Analysis and Discussion

The tools presented in this Systematic Literature Review passed the evaluation process defined in Section 3. Tools that only supported one device were left out, as well as tools that extended a VPL by applying it to IoT.

4.4.1 | Evolution Analysis

To understand the evolution of visual programming languages applied to IoT, the publication years of the tools found in Section 4, as well as the launch years of the survey tools of Section 4.2, were analyzed. Figure 5 contains the their evolution, where we can observe that there was a more substantial amount of work related to this topic in the years 2017 and 2018. The year 2019 still does not have conclusive data.

4.4.2 | Result Analysis

Scope Most of the tools found have several scopes, such as education, industry or home automation. From the 28 tools, six were specific to home automation, 4 to education, 3 to specific domains, and 1 for the industry; the remainder 14 had a wide range of use cases.

TABLE 2 Visual programming solutions applied to IoT and their characteristics. Small circles (●) mean *yes*, hyphens (-) means *no information available*, empty means *no* and asterisk (*) means more than one.

Tool	Scope	Architecture	License	Tier	Scalability	Programming	Web-based
Belsa et al. ³⁰	Several	Centralized	-	Cloud	High	Hybrid	●
Ivy ³¹	Several	Centralized	-	Cloud	Medium ⁷	Purely visual	
Ghiani et al. ³²	Home Automation	Centralized	-	Cloud	-	Form-based	●
ViSiT ³³	Several	Centralized	-	Cloud	High	Hybrids	●
Valsamakis and Savidis ³⁵	Ambient Assisted Living	Centralized	-	Cloud	-	Hybrid	●
WireMe ³⁶	Education, Home Automation	Centralized	-	Cloud	-	Hybrid	
VIPLE ³⁷	Education	Centralized	-	Cloud	-	Hybrid	
Smart Block ³⁸	Home Automation	Centralized	-	Cloud	-	Hybrid	●
PWCT ³⁹	Several	Centralized	GNU GPL v2.0	- ¹	High	Hybrid	
DDF ⁴⁰	-	Decentralized	Apache 2.0	Fog	High	Hybrid	●
GIMLE ⁴¹	Industry	Centralized	-	Cloud	High	Hybrid	●
DDFlow ⁴²	Security	Decentralized	-	Fog and Edge	-	Hybrid	●
Kefalakis et al. ⁴³	-	Centralized	LGPL V3.0 ³	Cloud	-	Hybrid	
Eterovic et al. ⁴⁴	Home Automation	- ⁴	-	-	-	Hybrid	-
FRED ⁴⁵	Several	Centralized	- ⁵	Cloud	High	Hybrid	●
WoTFlow ⁴⁶	-	Decentralized	-	Fog and Edge	-	Hybrid	●
Besari et al. ^{48 47}	Education	Centralized	-	Cloud	-	Hybrid	
CharIoT ⁴⁹	Home Automation	Centralized ⁶	-	Cloud and Edge ⁶	High ⁶	Form-based	●
Desolda et al. ⁵⁰	Smart Museums	-	-	-	-	Hybrid	
Eun et al. ⁵¹	Home Automation	Centralized	-	-	-	Form-based	●

¹ Used for several purposes, did not specify the tier it is located in regarding IoT.

² Since it uses Node-RED, this information was based on its architecture.

³ Under the same license of OpenIoT.

⁴ No information is given regarding the architecture of the environment created, only the VPL.

⁵ No information about the license is given, but further research discovered that it had paid plans and no source code available.

⁶ CharIoT uses the Giotto stack⁶⁰ from where we retrieved this information.

⁷ Certainty regarding this information is low.

TABLE 3 Characterization of the visual programming solutions applied to IoT from survey⁸. Small circles (●) mean *yes*, hyphens (-) means *no information available*, empty means *no* and asterisk (*) means more than one.

Tool	Scope	Architecture	License	Tier	Scalability	Programming	Web-based
Node-Red ⁵²	Several	Centralized	Apache 2.0	Cloud and Edge	High	Hybrid	●
NETLab Toolkit ⁵³	-	-	GNU GPL	Edge ²	-	Hybrid	●
NooDL ⁵⁴	Several	-	NooDL End User License ¹	Cloud ²	-	Hybrid	
DGLux5 ⁵⁴	Several	-	DGLux Engineering License	Cloud and Fog ²	High ²	Purely visual	
AT&T Flow Designer ⁵⁶	Several	-	GNU GPL3	Cloud ²	High ²	Hybrid	●
GrasPIO ⁵⁷	Education	-	BSD	Cloud ²	-	Purely visual	
Wylidrin ⁵⁸	Several	-	GNU GPL3	All ²	-	Hybrid	●
Zenodys ⁵⁹	Several	-	GNU GPL3	Cloud ²	High ²	Hybrid	●

¹ Available at <https://www.noodl.net/eula>

² Certainty regarding this information is low.

Architecture From the 28 tools found, 16 tools have a centralized architecture, three are decentralized, and the remaining nine do not present enough information to conclude on this topic.

License Most of the tools did not mention a license and the ones who did were in its majority open-source (*e.g.*, GNU GPL2, GNU GPL3, Apache 2.0 and LGPL3).

Scalability The majority of tools analyzed do not have scalability metrics analyzed, more specifically, the number of devices supported by them. The ones that do have high scalability, which seems to indicate that scalability is only analyzed when the tool has support for it.

Programming From the 28 analyzed tools, 22 employ a hybrid text and visual system visual programming paradigm, while 3 use a purely visual and the other three a form-based one.

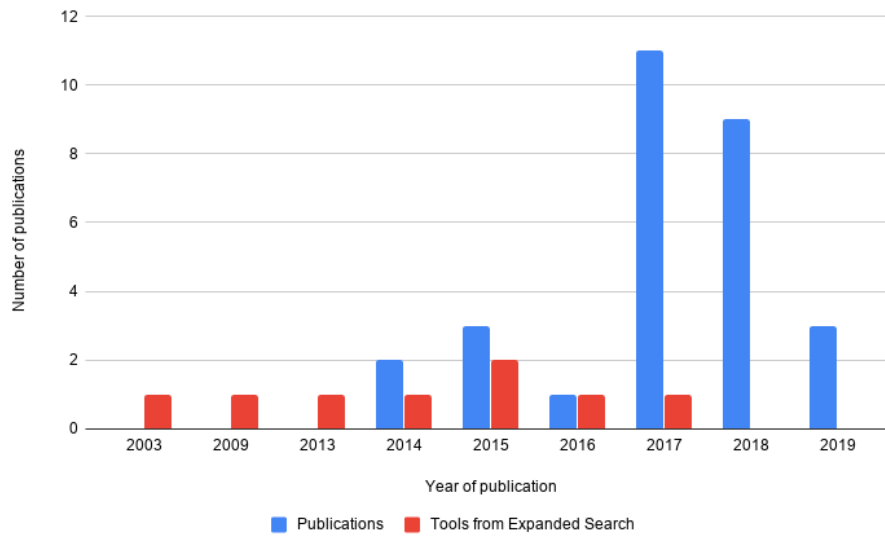


FIGURE 5 Publications and tools of VPL tools applied to IoT per year.

Web-based The majority of tools analyzed are web-based, being accessible with the use of a browser. Only one tool did not provide an environment, only a specification of a visual programming language.

4.4.3 | Research Questions

The research questions presented in Section 3.1 served as a way of directing the research of this Systematic Literature Review and obtain answers to relevant questions regarding the available tools that apply visual programming languages to the IoT domain. These answers are:

- RQ1** *What relevant visual programming solutions applied to IoT orchestration exist?* From the analyzed tools in Section 4 and 4.2, we found 28 visual programming tools applied to IoT orchestration.
- RQ2** *What is the tier and architecture of the tools found in RQ1?* Tables 2 and 3 give an overview of the characteristics of all the tools found. In these tables and subsequent analysis in Section 4.4.2 it is concluded that the majority of the tools have a centralized architecture and work in the Cloud tier.
- RQ3** *What was the evolution of visual programming solutions applied to IoT orchestration over the years?* As it can be observed in Section 4.4.1 and more specifically in Figure 5, there are visual programming tools applied to the orchestration of IoT since 2003, and in 2017 and 2018 there was a bigger number of publications with a focus on building these type of tools.

4.5 | Summary

In this Systematic Literature Review, 2698 publications were analyzed from IEEE, ACM and Scopus databases, resulting in 20 visual programming tools applied to the Internet-of-Things. A survey made on the visual programming solutions applied to IoT found during the research process resulted in 8 more tools, making a total of 28.

The results show that there is a significant number of tools that allow end-users to build IoT systems using visual programming in several different scopes. The majority of these tools have a centralized architecture and operate in the Cloud tier. Despite the considerable amount of tools, most of them do not have their source code accessible nor have a license. The results from the expanded search are more positive in this aspect, with the majority of them being open-source, such as Node-RED⁵², NETLab Toolkit⁵³ and others. However, this poses a problem since there is an evident lack of open source tools.

In summary, the majority of tools found do not possess a license, employ a centralized architecture, operate in the Cloud tier and use a hybrid text and visual programming system. Thus, it propels the possibility of future research on designing and building

a visual programming tool applied to IoT that is (1) open-source, (2) has a decentralized architecture and (3) also operates in the Fog and/or Edge tiers.

5 | VISUAL ORCHESTRATION IN IOT

Although the substantial amount of solutions found during our systematic literature (Section 4), only a small fraction of those aim to offer a truly decentralised solution to visual orchestration for Internet-of-Things systems. These solutions are now analysed in detail, followed by a comparison and discussion.

5.1 | DDF

The work made by in WoTFlow⁴⁶, DDF⁴⁰ and subsequent works^{61,62} consists of a system built on the Node-RED framework and focused on the use case of Smart Cities. Their goal is to make a tool more suitable for the development of fog-based applications that are dependent on the context of the edge devices where they operate.

In DDF⁴⁰, the authors started by extending Node-RED and implementing D-NR (Distributed Node-RED), which contains processes that can run across devices in local networks and servers in the Cloud. The application, called flow, is built in the visual programming environment, which is running in a development server. All the other devices running D-NR subscribe to an MQTT topic that contains the status of the flow. When a flow is deployed, all devices running D-NR are notified and subsequently analyse the given flow. Based on a set of constraints, they decide which nodes they may need to deploy locally and which sub-flow (parts of a flow) must be shared with other devices. Each device has a set of characteristics, from its computational resources such as bandwidth, available storage to its location. The developer can insert constraints into the flow, by specifying which device a sub-flow must be deployed in or the computational resources needed. Further, each device must be inserted manually into the system by a technician.

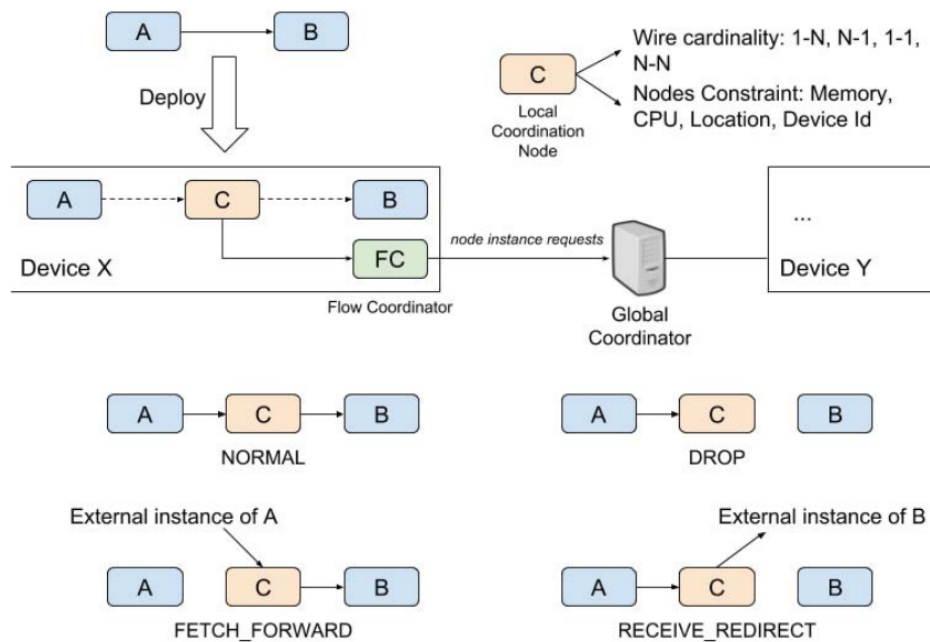


FIGURE 6 Coordination between nodes in D-NR⁶¹.

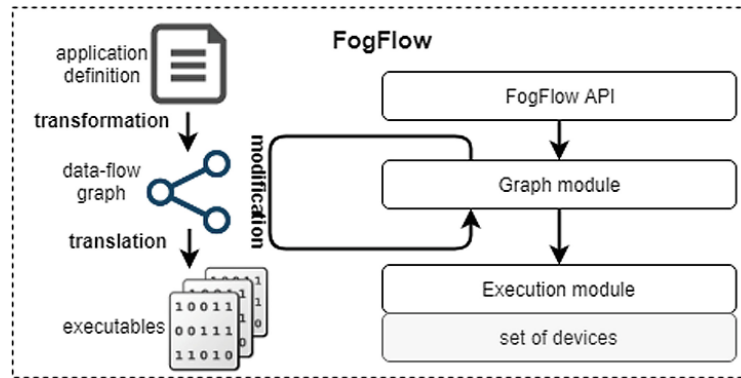


FIGURE 8 *FogFlow* architecture⁶⁴.

and heterogeneity, they abstract out the application definition from its architecture and rely on graph representation to provide an unambiguous, well-defined model of computations. The application definition should be infrastructure-independent and contain only data processing logic, and its execution should be possible on different sets of devices with different capabilities. Several algorithms for flow decomposition are mentioned^{65,66}, but none were specified in terms of results. Figure 8 represents the *FogFlow* architecture, which is composed by three modules: (1) the *FogFlow* API, which enables the creation of the application definition, (2) the Graph Module, responsible for processing and transforming the application definition into a data flow graph and finally the (3) Execution Model, which translates the graph and generates executables ready to be run on the assigned devices.

5.3 | *FogFlow*

There is another tool with the same name *FogFlow* but created by Cheng et al.⁶⁷. In the first publication related to this tool⁶⁸, the contributions made were the implementation of a standards-based programming model for Fog Computing and scalable context management. The first contribution consists in extending the dataflow programming model with hints to facilitate the development of fog applications. The scalable context management introduces a distributed approach, which allows overcoming the limits in a centralised context, achieving much better performance in terms of throughput, response time and scalability. The *FogFlow* framework focuses in a Smart City Platform use case, separated in three areas: (1) Service Management, typically hosted in the Cloud, (2) Data Processing, present in cloud and edge devices and (3) Context Management, which is separated in a device discovery unit hosted in the Cloud and IoT brokers scattered in Edge and Cloud.

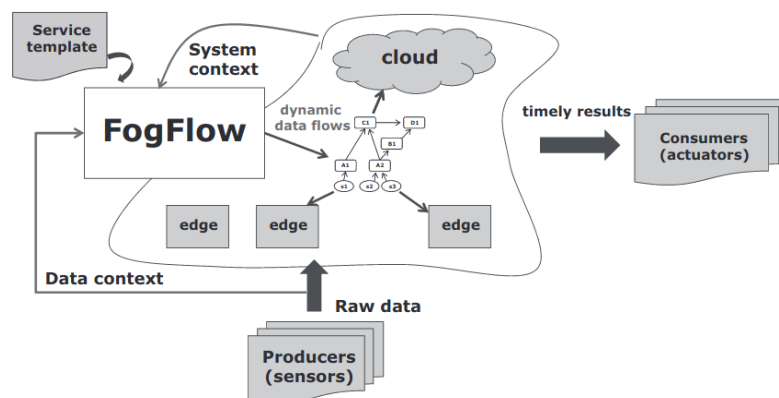


FIGURE 9 *FogFlow* high level model⁶⁹.

In more recent work⁶⁹, *FogFlow* was improved to deliver infrastructure providers with an environment that allows them to build decentralised IoT systems faster, with increased stability and scalability. The architecture can be seen in Figure 9, where dynamic data representing the IoT system flows that are orchestrated between sensors (Producers) and actuators (Consumers). The application is first designed using the *FogFlow* Task Designer, a hybrid text and visual programming environment, which results in an abstraction called Service Template. This abstraction contains specifics about the resources needed for each part of the system. Once the Service Template is submitted, the framework will determine how to instantiate it using the context data available. Each task is associated with an operator (a Docker image), and its assignment is based on (1) how many resources are available on each edge node, (2) the location of data sources, and (3) the prediction of workload. Edge nodes are autonomous since they can make their own decisions based on their local context, without relying on the central Cloud.

5.4 | DDFlow

DDFlow⁴², first mentioned in Section 4, presents another distributed approach by extending Node-RED with a system run-time that supports dynamic scaling and adaption of application deployments. The coordinator of the distributed system maintains the state and assigns tasks to available devices, minimizing end-to-end latency. Dataflow notions of *node* and *wire* are expanded, with a *node* in DDFlow representing an instantiation of a task that is deployed in a device, receiving inputs and generating outputs. *Nodes* can be constrained in their assignment by optional parameters, *Device*, and *Region*, inserted by the developer. A *wire* connects two or more nodes and can have three types: *Stream* (one-to-one), *Broadcast* (one-to-many) and *Unite* (many-to-one).

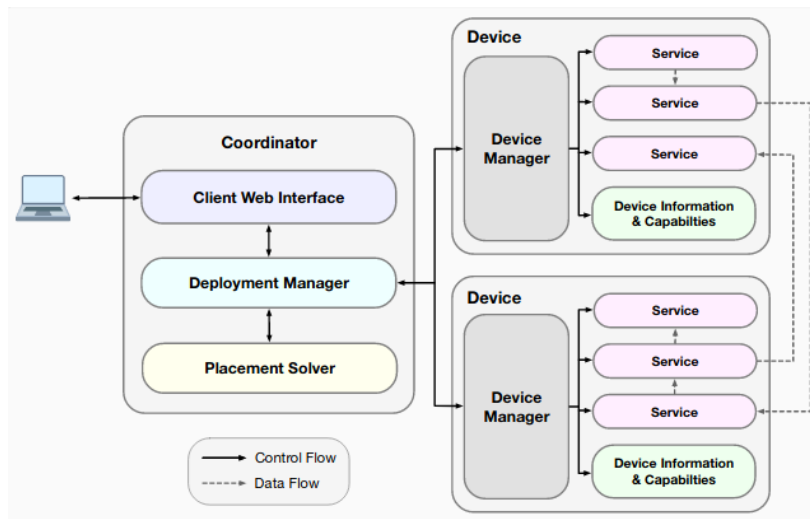


FIGURE 10 DDFlow architecture⁴².

In a DDFlow system, each device has a set of capabilities and a list of services that correspond to an implementation of a *Node* (Fig. 10). The devices communicate this information through their Device Manager or a proxy if it is a constrained device. The coordinator is a web server responsible for managing the DDFlow applications and is composed of three parts, which can be seen in Figure 10: (1) a visual programming environment where DDFlow applications are built, (2) a Deployment Manager that communicates with the Device Managers of the devices and (3) a Placement Solver, responsible for decomposing and assigning tasks to the available devices. When an application is deployed, a network topology graph and a task graph are constructed based on the real-time information retrieved from the devices. The coordinator proceeds with mapping tasks to devices by minimising the task graph's end-to-end latency of the longest path. Dynamic adaptation is supported by monitoring the system and adapting to changes. If changes in the network are detected, such as the failure or disconnection of a device, adjustments in the assignment of tasks are made. In addition to this, the coordinator can be replicated onto many devices to improve the reliability and fault-tolerance of the system.

In the evaluation made to DDFlow, the system can recover from network degradation or device overload, whereas in a centralised system, this would cause its total failure.

5.5 | Comparison and Discussion

The mentioned tools were characterised based on their mentions or support for the following features and characteristics:

1. **Leverage devices.** A decentralised architecture takes advantage of the computational power of the devices in the network, assigning them tasks. However, some tools can have limitations on the type of devices, making constraints or only focusing on the devices of the Fog tier and not Edge.
2. **Capabilities communication.** The devices need to communicate to the orchestrator their capabilities so that it can make an informed decision regarding the decomposition and assignment of tasks.
3. **Open-source.** The license of software or tool is essential in terms of its usability. Open-source allows access to the code, making it possible for its analysis, improvement, and reuse.
4. **Computation decomposition.** To implement a decentralised architecture, it is important to decompose the computation of the system into independent and logical tasks that can be assigned to devices. This is made using algorithms, which can be specified or mentioned.
5. **Run-time adaptation.** A system needs to adapt to run-time changes, such as non-availability of devices or even network failure. The system notices these events and can take action to circumvent the problems and keep functioning.

TABLE 4 Small circles (●) mean yes, hyphens (-) means *no information available*, empty means *no* and asterisk (*) means more than one.

Tool	Leverage devices	Capabilities communication	Open-source	Computation decomposition	Run-time adaptation
DDF ^{40,46,61,62}	Limited ¹	●	●	Limited ²	●
<i>FogFlow</i> & <i>uFlow</i> ^{63,64}	●	Limited ³		Limited ²	Limited ³
<i>FogFlow</i> ^{67,68,69}	●	-	●	Limited ²	●
DDFlow ⁴²	Limited ⁴	●		Limited ²	●

¹ Assumes that all devices run Node-RED, which limits the type of devices.

² Do not specify the algorithm used.

³ Communication between devices is made through the Cloud.

⁴ Assumes that all devices have a list of specific services they can provide.

From the analysis and the characteristics Table 4, we can conclude that the current research in decentralised architectures in visual programming tools applied to IoT is incomplete. All the tools leverage the devices in the network but in different ways. DDF⁴⁰ assumes that all devices run Node-RED, which limits the type of devices that can be leveraged since it needs to have minimum resources to run it. *FogFlow* and *uFlow*^{64,63} is the only tool that specifies how it truly leverages constrained devices, with the transformation of sub-flows into Lua code, with DDFlow⁴² assuming that all devices have a list of specific services they can provide, that should match the node assigned to them.

Regarding the method used to decompose and assign computations to the available devices, DDFlow describes the process with the use of the longest path algorithm focused on reducing end-to-end latency between devices. *FogFlow* and *uFlow*^{64,63} mention several algorithms that could be used, but do not specify which one was implemented. Both DDF⁴⁰ and *FogFlow*^{68,69} do not specify the algorithm used besides some constraints but are the only tools with their source code accessible and with an open-source license. All the tools claim to have support for run-time adaptation to changes in the system, such as device failures.

6 | CONCLUSION

Several solutions are available that provide a decentralized architecture in visual programming tools applied to the Internet-of-Things paradigm. However, some of these tools solve specific problems or make assumptions regarding the scale of the system and the constraints of the devices. We can highlight the following research challenges that remain to be addressed by the research community:

1. **Leveraging devices in the network:** since most tools use a centralized architecture, including Node-RED, they do not leverage the devices in the network. Fog Computing introduces a decentralized solution, one that can be applied to Node-RED by distributing the computational tasks across the edge devices.
2. **Communication of computational capabilities:** some of the current tools require the developer to manually introduce the resources of each device in the network, which is not a scalable solution. Others have a specific list of services, manually inserted that the devices can provide. Information about the computational capabilities of the devices in the network is vital for the successful distribution of computation across the devices.
3. **Detecting unavailability:** when a device fails or becomes unavailable, the system needs to realize and adapt automatically. The majority of current solutions do not possess this feature, which is vital if a system aims to adapt to changes in the environment dynamically.
4. **Code generation of sub-flows:** to truly leverage constrained devices, it is important to convert sub-flows or "tasks" into executable code. Devices that support simple firmware capable of executing code can be used to execute blocks of code, despite their limited capabilities.
5. **Provide self-adaption of the system:** devices can fail, as well as the connection between them or even the network. The system needs to discover and identify these changes and adapt to them at run-time in order to keep functioning.

Addressing these research challenges can improve the current state of the Internet-of-Things ecosystem, where systems are mostly centralized, leading to the proliferation of single point of failure (SPOF) (*e.g.*, dependency on the Cloud can render systems unusable if there is an Internet-connectivity problem). Further, high amounts of computational power are unused (*e.g.*, response times could be improved by using on-premises and already existent resources), and can address some pending security and privacy issues¹⁴.

We should also mention that although Partha's survey⁸ conclude that there exists some advantages of using visual programming languages, such as the ease of visualizing programming logic (which useful for rapid prototyping), and less burden on handling syntax error, as well as mentioning some negative aspects, such as the significant amount of time required for building simple IoT applications, we cannot find support or a rationale on how this observation results from the attributes they analysis (*e.g.*, licence and project repository).

References

1. Buyya R, Dastjerdi AV. *Internet of Things: Principles and Paradigms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. 1st ed. 2016.
2. Alam T. A Reliable Communication Framework and Its Use in Internet of Things (IoT). 2018; 3.
3. Chen S, Xu H, Liu D, Hu B, Wang H. A Vision of IoT: Applications, Challenges, and Opportunities With China Perspective. *IEEE Internet of Things Journal* 2014; 1(4): 349-359. doi: 10.1109/JIOT.2014.2337336
4. Zhang K, Han D, Feng H. Research on the complexity in internet of things. *IET Conference Publications* 2010; 2010(571 CP): 395-398. doi: 10.1049/cp.2010.0796
5. Burnett M, Kulesza T. End-User Development in Internet of Things: We the People. In *International Reports on Socio-Informatics (IRSI), Proceedings of the CHI 2015 - Workshop on End User Development in the Internet of Things Era* 2015; 12(2): 81-86.
6. Prehofer C, Chiarabini L. From IoT Mashups to Model-based IoT. *W3C Workshop on the Web of Things* 2013.
7. Chang SK. *Handbook of Software Engineering and Knowledge Engineering*. World Scientific Publishing Company . 2002
8. Ray PP. A Survey on Visual Programming Languages in Internet of Things. *Scientific Programming* 2017; 2017. doi: 10.1155/2017/1231430

9. Dias JP, Lima B, Faria JP, Restivo A, Ferreira HS. Visual Self-Healing Modelling for Reliable Internet-of-Things Systems. In: Springer; 2020: 27-36. To Appear.
10. Zhou Y, Zhang D, Xiong N. Post-cloud computing paradigms: A survey and comparison. *Tsinghua Science and Technology* 2017; 22(6): 714–732. doi: 10.23919/TST.2017.8195353
11. Bangui H, Rakrak S, Raghay S, Buhnova B. Moving to the edge-cloud-of-things: Recent advances and future research directions. *Electronics (Switzerland)* 2018; 7(11). doi: 10.3390/electronics7110309
12. Varshney P, Simmhan Y. Demystifying fog computing: Characterizing architectures, applications and abstractions. In: IEEE. ; 2017: 115–124.
13. Kleppmann M, Wiggins A, Hardenberg P, McGranaghan M. Local-first software: you own your data, in spite of the cloud. In: ; 2019: 154-178
14. Rawassizadeh R, Pierson T, Peterson R, Kotz D. NoCloud: Exploring Network Disconnection through On-Device Data Analysis. *IEEE Pervasive Computing* 2018; 17. doi: 10.1109/MPRV.2018.011591063
15. 1 IJ. Internet of things (iot) - preliminary report. *ISO,Tech. Rep.* 2014.
16. Gubbi J, Buyya R, Marusic S, Palaniswami M. Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions. *Future Generation Computer Systems* 2012; 29. doi: 10.1016/j.future.2013.01.010
17. Nunes DS, Zhang P, Sá Silva J. A Survey on Human-in-the-Loop Applications Towards an Internet of All. *IEEE Communications Surveys Tutorials* 2015; 17(2): 944-965. doi: 10.1109/COMST.2015.2398816
18. Miao Yun , Bu Yuxin . Research on the architecture and key technology of Internet of Things (IoT) applied on smart grid. In: ; 2010: 69-72
19. Linthicum DS. Connecting Fog and Cloud Computing. *IEEE Cloud Computing* 2017; 4(2): 18-20. doi: 10.1109/MCC.2017.37
20. Aazam M, Khan I, Alsaffar AA, Huh EN. Cloud of Things: Integrating Internet of Things and cloud computing and the issues involved. In: IEEE. ; 2014: 414–419.
21. Liu W, Nishio T, Shinkuma R, Takahashi T. Adaptive resource discovery in mobile cloud computing. *Computer Communications* 2014; 50: 119 - 129. Green Networkingdoi: <https://doi.org/10.1016/j.comcom.2014.02.006>
22. Martín Fernández C, Díaz Rodríguez M, Rubio Muñoz B. An Edge Computing Architecture in the Internet of Things. In: ; 2018: 99-102
23. Shi W, Pallis G, Xu Z. Edge Computing [Scanning the Issue]. *Proceedings of the IEEE* 2019; 107(8): 1474-1481. doi: 10.1109/JPROC.2019.2928287
24. Shi W, Dustdar S. The Promise of Edge Computing. *Computer* 2016; 49(5): 78-81. doi: 10.1109/MC.2016.145
25. Iorga M, Feldman L, Barton R, Martin MJ, Goren NS, Mahmoudi C. Fog computing conceptual model. tech. rep., 2018.
26. Shu NC. Visual Programming: Perspectives and Approaches. *IBM Syst. J.* 1999; 38(2–3): 199–221. doi: 10.1147/sj.382.0199
27. Burnett MM, Baker MJ, Bohus C, Carlson P, Yang S, Van Zee P. Scaling up visual programming languages. *Computer* 1995; 28(3): 45-54. doi: 10.1109/2.366157
28. Boshernitsan M, Downes M. Visual Programming Languages: A Survey. 1998.
29. Petersen K, Vakkalanka S, Kuzniarz L. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology* 2015; 64: 1 - 18. doi: <https://doi.org/10.1016/j.infsof.2015.03.007>

30. Belsa A, Sarabia-Jacome D, Palau CE, Esteve M. Flow-based programming interoperability solution for IoT platform applications. In: ; 2018: 304–309
31. Ens B, Anderson F, Grossman T, Annett M, Irani P, Fitzmaurice G. Ivy: Exploring spatially situated visual programming for authoring and understanding intelligent environments. In: ; 2017: 156–163.
32. Ghiani G, Manca M, Paterno F, Santoro C. Personalization of context-dependent applications through trigger-action rules. *ACM Transactions on Computer-Human Interaction* 2017; 24(2): 14:1—14:33. doi: 10.1145/3057861
33. Akiki PA, Bandara AK, Yu Y. Visual simple transformations: Empowering end-users to wire internet of things objects. *ACM Transactions on Computer-Human Interaction* 2017; 24(2): 10:1—10:43. doi: 10.1145/3057857
34. Humble J, Crabtree A, Hemmings T, et al. “Playing with the Bits” User-Configuration of Ubiquitous Domestic Environments. In: . 2864. ; 2003: 256-263
35. Valsamakis Y, Savidis A. Visual end-user programming of personalized AAL in the internet of things. In: . 10217 LNCS. ; 2017: 159–174
36. Pathirana D, Sonnadara S, Hettiarachchi M, Siriwardana H, Silva C. WireMe - IoT development platform for everyone. In: ; 2017: 93–98
37. De Luca G, Li Z, Mian S, Chen Y. Visual programming language environment for different IoT and robotics platforms in computer science education. *CAAI Transactions on Intelligence Technology* 2018; 3(2): 119–130. doi: 10.1049/trit.2018.0016
38. Bak N, Chang BM, Choi K. Smart Block: A Visual Programming Environment for SmartThings. In: . 2. ; 2018: 32–37
39. Fayed MS, Al-Qurishi M, Alamri A, Al-Daraiseh AA. PWCT: Visual language for IoT and cloud computing applications and systems. In: ; 2017
40. Giang NK, Blackstock M, Lea R, Leung VC. Developing IoT applications in the Fog: A Distributed Dataflow approach. In: ; 2015: 155–162
41. Tomlein M, Grønbaek K. A visual programming approach based on domain ontologies for configuring industrial IoT installations. In: ; 2017
42. Noor J, Tseng HY, Garcia L, Srivastava M. DDFlow: Visualized declarative programming for heterogeneous IoT networks. In: *IoTDI '19*. ACM; 2019; New York, NY, USA: 172–177
43. Kefalakis N, Soldatos J, Anagnostopoulos A, Dimitropoulos P. *A visual paradigm for IoT solutions development*. 9001 . 2015
44. Eterovic T, Kaljic E, Donko D, Salihbegovic A, Ribic S. An Internet of Things visual domain specific modeling language based on UML. In: ; 2015
45. Blackstock M, Lea R. FRED: A hosted data flow platform for the IoT. In: ; 2016
46. Blackstock M, Lea R. Toward a distributed data flow platform for the Web of Things (Distributed Node-RED). In: . 08-October. ; 2014: 34–39
47. Setiawan R, Anom Besari AR, Wibowo IK, Rizqullah MR, Agata D. Mobile visual programming apps for internet of things applications based on raspberry Pi 3 platform. In: ; 2019: 199–204
48. Besari ARA, Wobowo IK, Sukaridhoto S, Setiawan R, Rizqullah MR. Preliminary design of mobile visual programming apps for Internet of Things applications based on Raspberry Pi 3 platform. In: . 2017-Janua. ; 2017: 50–54
49. Tomlein M, Boovaraghavan S, Agarwal Y, Dey AK. CharIoT: An end-user programming environment for the IoT. In: ; 2017
50. Desolda G, Malizia A, Turchi T. A tangible-programming technology supporting end-user development of smart-environments. In: *AVI '18*. ACM; 2018; New York, NY, USA: 59:1—59:3

51. Eun S, Jung J, Yun YS, So SS, Heo J, Min H. An end user development platform based on dataflow approach for IoT devices. *Journal of Intelligent and Fuzzy Systems* 2018; 35(6): 6125–6131. doi: 10.3233/JIFS-169852
52. Foundation O. Node-RED. Available: <https://nodered.org/>; 2020. Last access 2020. [Online].
53. Toolkit N. NETLabTK: Tools for Tangible Design. Available: www.netlabtoolkit.org/; 2020. Last access 2020. [Online].
54. NooDL . NooDL. Available: <https://classic.getnoodl.com/>; 2020. Last access 2020. [Online].
55. DGLogik . DGLux5. Available: <http://dglogik.com/products/dglux-for-dsa>; 2020. Last access 2020. [Online].
56. AT&T . AT&T Flow Designer. Available: <https://flow.att.com/>; 2020. Last access 2020. [Online].
57. Ltd. GHP. GraspIO. Available: <https://www.grasp.io/>; 2020. Last access 2020. [Online].
58. Wyliodrin . Wyliodrin. Available: <https://wyliodrin.com/>; 2020. Last access 2020. [Online].
59. B.V. Z. Zenodys. Available: <https://www.zenodys.com/>; 2020. Last access 2020. [Online].
60. Agarwal Y, Dey AK. Toward Building a Safe, Secure, and Easy-to-Use Internet of Things Infrastructure.. *IEEE Computer* 2016; 49(4): 88–91.
61. Giang NK, Lea R, Blackstock M, Leung VCM. Fog at the Edge: Experiences Building an Edge Computing Platform. In: ; 2018: 9-16
62. Giang NK, Lea R, Leung VCM. Exogenous Coordination for Building Fog-Based Cyber Physical Social Computing and Networking Systems. *IEEE Access* 2018; 6: 31740-31749. doi: 10.1109/ACCESS.2018.2844336
63. Szydlo T, Brzoza-Woch R, Sendorek J, Windak M, Gniady C. Flow-Based Programming for IoT Leveraging Fog Computing. In: ; 2017: 74-79
64. Sendorek J, Szydlo T, Windak M, Brzoza-Woch R. *FogFlow - Computation Organization for Heterogeneous Fog Computing Environments*: 634-647; 2019
65. NAAS MI, Lemarchand L, Boukhobza J, Raipin P. A Graph Partitioning-Based Heuristic for Runtime IoT Data Placement Strategies in a Fog Infrastructure. In: SAC '18. Association for Computing Machinery; 2018; New York, NY, USA: 767–774
66. Gupta H, Vahid Dastjerdi A, Ghosh SK, Buyya R. iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments. *Software: Practice and Experience* 2017; 47(9): 1275-1296. doi: 10.1002/spe.2509
67. SmartFog . FogFlow. Available: <https://github.com/smartfog/fogflow>; 2020. Last access 2020. [Online].
68. Cheng B, Solmaz G, Cirillo F, Kovacs E, Terasawa K, Kitazawa A. FogFlow: Easy Programming of IoT Services Over Cloud and Edges for Smart Cities. *IEEE Internet of Things Journal* 2017; PP: 1-1. doi: 10.1109/JIOT.2017.2747214
69. Cheng B, Kovacs E, Kitazawa A, Terasawa K, Hada T, Takeuchi M. FogFlow: Orchestrating IoT services over cloud and edges. *NEC Technical Journal* 2018; 13: 48-53.



References

- [1] ISO/IEC JTC 1. Internet of things (iot) - preliminary report. *ISO, Tech. Rep.*, 2014.
- [2] Mohammad Aazam, Imran Khan, Aymen Abdullah Alsaffar, and Eui-Nam Huh. Cloud of things: Integrating internet of things and cloud computing and the issues involved. In *Proceedings of 2014 11th International Bhurban Conference on Applied Sciences & Technology (IBCAST) Islamabad, Pakistan, 14th-18th January, 2014*, pages 414–419. IEEE, 2014.
- [3] Yuvraj Agarwal and Anind K Dey. Toward building a safe, secure, and easy-to-use internet of things infrastructure. *IEEE Computer*, 49(4):88–91, 2016.
- [4] Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu. Visual simple transformations: Empowering end-users to wire internet of things objects. *ACM Transactions on Computer-Human Interaction*, 24(2):10:1—10:43, apr 2017.
- [5] S. A. Al-Qaseemi, H. A. Almulhim, M. F. Almulhim, and S. R. Chaudhry. Iot architecture challenges and issues: Lack of standardization. In *2016 Future Technologies Conference (FTC)*, pages 731–738, Dec 2016.
- [6] Tanweer Alam. A reliable communication framework and its use in internet of things (iot). 3, 05 2018.
- [7] Fahed Alkhabbas, Romina Spalazzese, and Paul Davidsson. Iot-based systems of systems. *Proceedings of the 2nd edition of Swedish Workshop on the Engineering of Systems of Systems (SWESOS 2016)*, 2016.
- [8] AT&T. AT&T Flow Designer. Available: <https://flow.att.com>, 2020. Last access 2020. [Online].
- [9] Nayeon Bak, Byeong Mo Chang, and Kwanghoon Choi. Smart Block: A Visual Programming Environment for SmartThings. In *Proceedings - International Computer Software and Applications Conference*, volume 2, pages 32–37, 2018.
- [10] Andreu Belsa, David Sarabia-Jacome, Carlos E. Palau, and Manuel Esteve. Flow-based programming interoperability solution for IoT platform applications. In *Proceedings - 2018 IEEE International Conference on Cloud Engineering, IC2E 2018*, pages 304–309, 2018.
- [11] Adnan Rachmat Anom Besari, Iwan Kurnianto Wobowo, Sritrasta Sukaridhoto, Ricky Setiawan, and Muh Rifqi Rizqullah. Preliminary design of mobile visual programming apps for Internet of Things applications based on Raspberry Pi 3 platform. In *Proceedings - International Electronics Symposium on Knowledge Creation and Intelligent Computing, IES-KCIC 2017*, volume 2017-Janua, pages 50–54, 2017.

- [12] Michael Blackstock and Rodger Lea. Toward a distributed data flow platform for the Web of Things (Distributed Node-RED). In *ACM International Conference Proceeding Series*, volume 08-October, pages 34–39, 2014.
- [13] Michael Blackstock and Rodger Lea. FRED: A hosted data flow platform for the IoT. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs, MOTA 2016*, 2016.
- [14] Marat Boshernitsan and Michael Downes. Visual programming languages: A survey. 08 1998.
- [15] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang, and P. Van Zee. Scaling up visual programming languages. *Computer*, 28(3):45–54, March 1995.
- [16] Brendan Burns and Craig Tracey. *Managing Kubernetes: operating Kubernetes clusters in the real world*. O’Reilly Media, 2018.
- [17] Rajkumar Buyya and Amir Vahid Dastjerdi. *Internet of Things: Principles and Paradigms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2016.
- [18] Zenodys B.V. Zenodys. Available: <https://www.zenodys.com/>, 2020. Last access 2020. [Online].
- [19] S K Chang. *Handbook of Software Engineering and Knowledge Engineering*. World Scientific Publishing Company, 2002.
- [20] S. Chen, H. Xu, D. Liu, B. Hu, and H. Wang. A vision of iot: Applications, challenges, and opportunities with china perspective. *IEEE Internet of Things Journal*, 1(4):349–359, Aug 2014.
- [21] B. Cheng, E. Kovacs, A. Kitazawa, K. Terasawa, T. Hada, and M. Takeuchi. Fogflow: Orchestrating iot services over cloud and edges. *NEC Technical Journal*, 13:48–53, 11 2018.
- [22] Bin Cheng, Gurkan Solmaz, Flavio Cirillo, Ernő Kovacs, Kazuyuki Terasawa, and Atsushi Kitazawa. Fogflow: Easy programming of iot services over cloud and edges for smart cities. *IEEE Internet of Things Journal*, PP:1–1, 08 2017.
- [23] Gennaro De Luca, Zhongtao Li, Sami Mian, and Yinong Chen. Visual programming language environment for different IoT and robotics platforms in computer science education. *CAAI Transactions on Intelligence Technology*, 3(2):119–130, 2018.
- [24] Giuseppe Desolda, Alessio Malizia, and Tommaso Turchi. A tangible-programming technology supporting end-user development of smart-environments. In *Proceedings of the Workshop on Advanced Visual Interfaces AVI, AVI ’18*, pages 59:1—59:3, New York, NY, USA, 2018. ACM.
- [25] DGLogik. DGLux5. Available: <http://dglogik.com/products/dglux-for-dsa>, 2020. Last access 2020. [Online].
- [26] Barrett Ens, Fraser Anderson, Tovi Grossman, Michelle Annett, Pourang Irani, and George Fitzmaurice. Ivy: Exploring spatially situated visual programming for authoring and understanding intelligent environments. In *Proceedings - Graphics Interface*, pages 156–163, 2017.

- [27] Espressif Systems. Esp8266 technical reference manual. Technical report, Espressif Systems, Shanghai, China, 2019.
- [28] Espressif Systems. Esp32 technical reference manual. Technical report, Espressif Systems, Shanghai, China, 2020.
- [29] Teo Eterovic, Enio Kaljic, Dzenana Donko, Adnan Salihbegovic, and Samir Ribic. An Internet of Things visual domain specific modeling language based on UML. In *2015 25th International Conference on Information, Communication and Automation Technologies, ICAT 2015 - Proceedings*, 2015.
- [30] Seongbae Eun, Jinman Jung, Young Sun Yun, Sun Sup So, Junyoung Heo, and Hong Min. An end user development platform based on dataflow approach for IoT devices. *Journal of Intelligent and Fuzzy Systems*, 35(6):6125–6131, 2018.
- [31] Mahmoud S. Fayed, Muhammad Al-Qurishi, Atif Alamri, and Ahmad A. Al-Daraiseh. PWCT: Visual language for IoT and cloud computing applications and systems. In *ACM International Conference Proceeding Series*, 2017.
- [32] OpenJS Foundation. Node-RED. Available: <https://nodered.org/>, 2020. Last access 2020. [Online].
- [33] Giuseppe Ghiani, Marco Manca, Fabio Paterno, and Carmen Santoro. Personalization of context-dependent applications through trigger-action rules. *ACM Transactions on Computer-Human Interaction*, 24(2):14:1—14:33, apr 2017.
- [34] N. K. Giang, R. Lea, M. Blackstock, and V. C. M. Leung. Fog at the edge: Experiences building an edge computing platform. In *2018 IEEE International Conference on Edge Computing (EDGE)*, pages 9–16, July 2018.
- [35] N. K. Giang, R. Lea, and V. C. M. Leung. Exogenous coordination for building fog-based cyber physical social computing and networking systems. *IEEE Access*, 6:31740–31749, 2018.
- [36] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C.M. Leung. Developing IoT applications in the Fog: A Distributed Dataflow approach. In *Proceedings - 2015 5th International Conference on the Internet of Things, IoT 2015*, pages 155–162, 2015.
- [37] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29, 07 2012.
- [38] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K. Ghosh, and Rajkumar Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.
- [39] Jan Humble, Andy Crabtree, Terry Hemmings, Karl-Petter Åkesson, Boriana Koleva, Tom Rodden, and Pär Hansson. “playing with the bits” user-configuration of ubiquitous domestic environments. volume 2864, pages 256–263, 10 2003.
- [40] Michaela Iorga, Larry Feldman, Robert Barton, Michael J Martin, Nedim S Goren, and Charif Mahmoudi. Fog computing conceptual model. Technical report, 2018.

- [41] Nikos Kefalakis, John Soldatos, Achilleas Anagnostopoulos, and Panagiotis Dimitropoulos. *A visual paradigm for IoT solutions development*, volume 9001. 2015.
- [42] Martin Kleppmann, Adam Wiggins, Peter Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. pages 154–178, 10 2019.
- [43] D. S. Linthicum. Connecting fog and cloud computing. *IEEE Cloud Computing*, 4(2):18–20, March 2017.
- [44] Wei Liu, Takayuki Nishio, Ryoichi Shinkuma, and Tatsuro Takahashi. Adaptive resource discovery in mobile cloud computing. *Computer Communications*, 50:119 – 129, 2014. Green Networking.
- [45] Grasp IO Innovations Pvt. Ltd. GraspIO. Available: <https://www.grasp.io/>, 2020. Last access 2020. [Online].
- [46] C. Martín Fernández, M. Díaz Rodríguez, and B. Rubio Muñoz. An edge computing architecture in the internet of things. In *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pages 99–102, May 2018.
- [47] Miao Yun and Bu Yuxin. Research on the architecture and key technology of internet of things (iot) applied on smart grid. In *2010 International Conference on Advances in Energy Engineering*, pages 69–72, June 2010.
- [48] Mohammed Islam NAAS, Laurent Lemarchand, Jalil Boukhobza, and Philippe Raipin. A graph partitioning-based heuristic for runtime iot data placement strategies in a fog infrastructure. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC '18*, page 767–774, New York, NY, USA, 2018. Association for Computing Machinery.
- [49] NoodL. NoodL. Available: <https://classic.getnoodl.com/>, 2020. Last access 2020. [Online].
- [50] Joseph Noor, Hsiao Yun Tseng, Luis Garcia, and Mani Srivastava. DDFlow: Visualized declarative programming for heterogeneous IoT networks. In *IoTDI 2019 - Proceedings of the 2019 Internet of Things Design and Implementation, IoTDI '19*, pages 172–177, New York, NY, USA, 2019. ACM.
- [51] D. S. Nunes, P. Zhang, and J. Sá Silva. A survey on human-in-the-loop applications towards an internet of all. *IEEE Communications Surveys Tutorials*, 17(2):944–965, Secondquarter 2015.
- [52] D. Pathirana, S. Sonnadara, M. Hettiarachchi, H. Siriwardana, and C. Silva. WireMe - IoT development platform for everyone. In *3rd International Moratuwa Engineering Research Conference, MERCon 2017*, pages 93–98, 2017.
- [53] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1 – 18, 2015.
- [54] Antonio Ramadas, Gil Domingues, Joao Pedro Dias, Ademar Aguiar, and Hugo Sereno Ferreira. Patterns for things that fail. In *Proceedings of the 24th Conference on Pattern Languages of Programs, PLoP '17*, USA, 2017. The Hillside Group.

- [55] Reza Rawassizadeh, Timothy Pierson, Ronald Peterson, and David Kotz. Nocloud: Exploring network disconnection through on-device data analysis. *IEEE Pervasive Computing*, 17, 03 2018.
- [56] Partha Pratim Ray. A Survey on Visual Programming Languages in Internet of Things. *Scientific Programming*, 2017, 2017.
- [57] James Scott and Rick Kazman. Realizing and Refining Architectural Tactics : Availability. Technical Report August, Software Engineering Institute, 2009.
- [58] Joanna Sendorek, Tomasz Szydlo, Mateusz Windak, and Robert Brzoza-Woch. *FogFlow - Computation Organization for Heterogeneous Fog Computing Environments*, pages 634–647. 06 2019.
- [59] Ricky Setiawan, Adnan Rachmat Anom Besari, Iwan Kurnianto Wibowo, Muh Rifqi Rizqullah, and Dias Agata. Mobile visual programming apps for internet of things applications based on raspberry Pi 3 platform. In *International Electronics Symposium on Knowledge Creation and Intelligent Computing, IES-KCIC 2018 - Proceedings*, pages 199–204, oct 2019.
- [60] W. Shi and S. Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, May 2016.
- [61] W. Shi, G. Pallis, and Z. Xu. Edge computing [scanning the issue]. *Proceedings of the IEEE*, 107(8):1474–1481, Aug 2019.
- [62] N. C. Shu. Visual programming: Perspectives and approaches. *IBM Syst. J.*, 38(2–3):199–221, June 1999.
- [63] SmartFog. FogFlow. Available: <https://github.com/smartfog/fogflow>, 2020. Last access 2020. [Online].
- [64] Dipa Soni and Ashwin Makwana. A survey on mqtt: a protocol of internet of things (iot). In *International Conference On Telecommunication, Power Analysis And Computing Techniques (ICTPACT-2017)*, 2017.
- [65] T. Szydlo, R. Brzoza-Woch, J. Sendorek, M. Windak, and C. Gniady. Flow-based programming for iot leveraging fog computing. In *2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 74–79, June 2017.
- [66] Matúš Tomlein, Sudershan Boovaraghavan, Yuvraj Agarwal, and Anind K. Dey. CharIoT: An end-user programming environment for the IoT. In *ACM International Conference Proceeding Series*, 2017.
- [67] Matúš Tomlein and Kaj Grønæk. A visual programming approach based on domain ontologies for configuring industrial IoT installations. In *ACM International Conference Proceeding Series*, 2017.
- [68] NETLab Toolkit. NETLabTK: Tools for Tangible Design. Available: www.netlabtoolkit.org/, 2020. Last access 2020. [Online].
- [69] Yannis Valsamakis and Anthony Savidis. Visual end-user programming of personalized AAL in the internet of things. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10217 LNCS, pages 159–174, 2017.

- [70] Wylodrin. Wylodrin. Available: <https://wylodrin.com/>, 2020. Last access 2020. [Online].