TÉCNICO LISBOA

InfraRED: A Visual and Flow-based Designer of Smart IT Infrastructures

André Miguel Jesus Moura da Silva Moreira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. José Carlos Martins Delgado
Doctor Rui António dos Santos Cruz

Examination Committee

Chairperson: Prof. António Manuel Ferreira Rito da Silva
Supervisor: Prof. José Carlos Martins Delgado
Member of the Committee: Prof. Luís David Figueiredo Mascarenhas Moreira Pedrosa

October 2022

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Acknowledgments

I would like to thank my girlfriend for motivating me through the hardships and for giving me a comfortable space to rest in between work. I would like to thank my mom for her time and support and her dedication to my success. Thank you to all my friends and colleagues that helped me push this thesis to a greater quality. I would also like to acknowledge my dissertation supervisors Doctor Rui Cruz and Prof. José Delgado for their insight, support and sharing of knowledge.

Abstract

Information Technology (IT) management solutions and Infrastructure as Code (IaC) tools already allow the provisioning of networks and systems, but mostly in a non visual way, either by using web-based menus or Command Line Interface (CLI) instructions which need the user to be aware of the complexities of designing and maintaining those infrastructures and all the inherent minute details and configuration parameters. InfraRED is a proposal for software solution that can help users, either coders or non coders, to design and maintain visually, their own logical system topologies, to run on top of generic hardware, either locally or on cloud environments, while providing a friendly and accessible user interface, as InfraRED is based in the *Low Code* paradigm, which provides the user with a visual set of tools for systems' creation, management and analysis, not requiring detailed knowledge about the complexity of the *intended* infrastructures. The visual based solution will work similarly to how Intent Based Networking (IBN) technologies work by using *intents* to create and define a system's topology based on an end goal, but where those *intents* will be visually represented by the design the user wishes to achieve. Maintaining the created topologies will also be made easier through InfraRED since the user will have a high-level overview of how the core elements interact with each other and how data moves from one to another.

Keywords

Low Code; Cloud Computing; Infrastructure as Code (IaC); Development and Operations (DevOps);

Resumo

As soluções para gestão de Tecnologias de Informação (TI) e as ferramentas de *IaC* já são capazes de provisionar redes e sistemas mas de forma não visual, na maioria usando páginas web com menus ou através de uma interface em linha de comandos (CLI) e instruções que requerem que o utilizador esteja familiarizado com a complexidade de concepção e de manutenção dessas infraestruturas, e de todos os detalhes minuciosos e parâmetros de configuração inerentes.InfraRED é uma proposta para uma solução de software que pode ajudar vários tipos de utilizadores, quer programadores ou não programadores, a desenhar e a manter visualmente e de forma simples as suas topologias lógicas de sistemas tanto localmente como em ambientes de Computação em Nuvem visto que o interface de utilizador do InfraRED é baseado no paradigma de *Low Code*, equipando-o com um conjunto de ferramentas para criar, manter e analisar as suas infraestruturas, sem ser necessário conhecimento detalhado das complexidades das mesmas. A solução InfraRED funciona de forma semelhante a tecnologias Intent Based Networking (IBN), onde são usadas *intenções* para criar e definir topologias construídas a partir de um objectivo final, a *intenção*. No caso do InfraRED, estas *intenções* são representadas visualmente pelo desenho da topologia que o utilizador pretende alcançar. A manutenção dos sistemas também ficará facilitada com o InfraRED porque os utilizadores poderão entender mais facilmente como os elementos da topologia interagem uns com os outros e como os dados fluem entre eles.

Palavras Chave

Low Code; Computação em Nuvem; *Infrastructure as Code (IaC)*; *Development and Operations (DevOps)*;

Contents

1 Introduction **1**

1.1 How to solve the Problem . . . 3

1.2 Organization of the Document . . . 4

2 State of the Art **5**

2.1 Background . . . 7

2.1.1 Low Code . . . 7

2.1.2 Cloud Infrastructure . . . 7

2.1.3 Internet of Things (IoT) . . . 8

2.1.4 Intent Based Networking (IBN) . . . 9

2.2 State of the Art . . . 9

2.2.1 Node-RED . . . 10

2.2.2 TOSCA . . . 11

2.2.3 Google Cloud Platform (GCP) and Amazon Web Services (AWS) . . . 13

2.2.4 Metal as a Service (MaaS) . . . 14

2.2.5 Juju Charms . . . 15

2.2.6 Sagitech Software Studio (S3) . . . 17

3 InfraRED Design **19**

3.1 Architecture . . . 21

3.1.1 Server . . . 21

3.1.2 Client . . . 21

3.2 Infrastructure . . . 23

3.2.1 Nodes . . . 23

3.2.2 Capabilities and Requirements . . . 24

3.2.3 Relationships . . . 24

4 Solution Implementation **25**

4.1 Technology Stack . . . 27

4.1.1 NodeJS and ExpressJS . . . 27

4.1.2 Server . . . 27
4.1.3 Client . . . 27
4.2 User Interface (UI) of the Client Side . . . 28
4.2.1 Node . . . 28
4.2.2 Canvas . . . 30
4.2.3 Status Bar . . . 30
4.2.4 Category Bar . . . 30
4.2.5 Resource Bar . . . 31
4.2.6 Menu Bar . . . 31
4.2.7 Node Modal . . . 32
4.2.8 Relationship Lines . . . 32
4.3 Logic at the Client Side . . . 33
4.3.1 Nodes and Relationships . . . 33
4.3.2 Node Database Memory . . . 34
4.3.3 Events . . . 34
4.3.4 Loading the Nodes . . . 35
4.3.5 Initiating the Deployment . . . 35
4.4 Logic at the Server Side . . . 36
4.4.1 Node Files . . . 36
4.4.2 Receiving a Deployment Request . . . 37
4.4.3 Asynchronous Behaviour for the Server . . . 39
4.5 Deployment Methods and Use Case . . . 40
4.5.1 Deployment Example . . . 40
4.5.2 Client Side Process . . . 41
4.5.3 System Initialization . . . 42
4.5.4 User Interaction . . . 43
4.6 Development . . . 46
5 Evaluation **47**
5.1 Installation and Setup . . . 49
5.2 Loading Nodes . . . 50
5.3 Node Type Availability . . . 50
5.4 Usability . . . 51
5.4.1 Dashboard Functionalities . . . 51
5.4.2 Adding Nodes to the Canvas . . . 51
5.4.3 Creating a Relation between Nodes . . . 53

6 Conclusion **55**
6.1 Limitations . . . 57
6.2 Improvements . . . 58
6.2.1 Nodes . . . 58
6.2.2 Resource Bar Saved Pattern Nodes . . . 58
6.2.3 Tutorial and Descriptions . . . 58
6.2.4 Adding new Nodes . . . 59
6.2.5 Node Search . . . 59
Bibliography **61**

List of Figures

2.1 Cloud Service Providing 8
2.2 Node-RED Dashboard. 10
2.3 Node-RED Relation 11
2.4 TOSCA Service Template Example 12
2.5 Relationship between TOSCA Nodes 12
2.6 Node Artifacts 13
2.7 Public Cloud Platforms Dasboards 14
2.8 MaaS Compatibility 15
2.9 Juju Dashboard 16
2.10 Juju Relation 16

3.1 InfraRED Canvas Layout 22
3.2 Node Structure 23

4.1 InfraRED Dashboard 29
4.2 Node Example 29
4.3 Categories Example 31
4.4 Node Modal Box 32
4.5 A Relationships (blue line) between Nodes 33
4.6 Deployment Sequence per Level 40
4.7 Client File Structure 41
4.8 Adding a Node to the Canvas 43
4.9 Establishing Relationships between Nodes 44
4.10 Interacting with Node Properties 44
4.11 Saving Node properties 44
4.12 Deleting a Node from the Canvas 45
4.13 Deleting a Pattern Node 45

4.14 Saving a Pattern Node 45
4.15 Deploying a design 46

5.1 InfraRED Dashboard 52
5.2 Juju Dashboard 52
5.3 Node-RED Dashboard 53
5.4 InfraRED Relation 53
5.5 Node-RED Relation 54
5.6 Juju Charms creating a connection 54

Listings

4.1 Connectable and Node classes at client/nodes.js 33
4.2 Event example from client/events.js 34
4.3 GET call for /listNodes at client/loader.js 35
4.4 POST call for /deploy at client/deployer.js 35
4.5 POST call for /save at client/deployer.js 36
4.6 Deployment entry point at server/deployer.js 37
4.7 Node clean method before deploying at server/deployer.js 37
4.8 Creating the level lists at server/deployer.js 37
4.9 Creating the Node Instances at server/deployer.js 38
4.10 Deploying the Nodes at server/deployer.js 39
4.11 Searching for Node files at server/registry.js 42
4.12 Builing Runtime list of Nodes at server/registry.js 42

Acronyms

API	Application Program Interface
AWS	Amazon Web Services
CAPEX	Capital Expenditure
CLI	Command Line Interface
CPE	Customer-Premises Equipment
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DevOps	Development and Operations
GCP	Google's Cloud Platform
GUI	Graphical User Interface
HTML	HyperText Markup Language
IaaS	Infrastructure as a Service
IaC	Infrastructure as Code
IBN	Intent Based Networking
IDE	Integrated Development Environment
IoT	Internet of Things
IT	Information Technology
JaaS	Juju as a Service
JSON	JavaScript Object Notation
MaaS	Metal as a Service
OASIS	Organization for the Advancement of Structured Information Systems
OS	Operating System
PaaS	Platform as a Service

RAM	Random Access Memory
REST	Representational State Transfer
SaaS	Software as a Service
SASS	Syntactically Awesome Style Sheets
SVG	Scalable Vector Graphics
TOSCA	Topology and Orchestration Specification for Cloud Applications
UI	User Interface
URL	Uniform Resource Locator
YAML	YAML Ain't Markup Language

1

Introduction

Contents

1.1 How to solve the Problem 3
1.2 Organization of the Document 4

Nowadays, creating and managing Information Technology (IT) infrastructures poses many challenges. The growth in number of connected devices implies the need for easier scalability, and for a more dynamic method to re-design interconnected systems and logical topologies in a cloud or on premises.

Current IT management solutions and Infrastructure as Code (IaC) tools already allow the provisioning of networks and systems, as well as the deployment of software, mostly in a non visual way, in terms of topology visualization, either by using web-based menus or Command Line Interface (CLI) together with configuration files, which are only valuable when the user is aware of the configuration complexities of those infrastructures.

Additionally on the Internet of Things (IoT) market, different brands offer software and devices that use different languages and protocols from each other to describe the intended infrastructure components or to exchange data between their dedicated devices which, come at the expense of interoperability if all the devices are not from the same brand [1].

Abstraction in programming languages allows for a more general audience to easily make and create software solutions not locked-in to specific devices or systems vendors, which is important when the proliferation of cloud environments, IoT and respective software applications have been increasing in number, added to the fact that most networks, systems, services and functions are being virtualized, making better use of available resources either on premises or on cloud environments. On top of this the *Low Code* paradigm has been the main option when it comes to allowing as many users as possible to work in any coding field [2].

Development and Operations (DevOps) methodologies and IaC practices already enable a more systematic approach to the life-cycle of IT infrastructures through continuous development, deployment, testing and version control. However, while IaC allows for a "program-like" description of the infrastructure, it still does not allow for a simple method of defining the requirements, the metrics and the desired interactions between the elements of the intended system.

1.1 How to solve the Problem

So, as motivation, the objective of the InfraRED project is to abstract IT infrastructures creation, but in a *Low Code* visual way, so that the users can visually compose their desired infrastructure at the moment of design, easily defining the requirements, the metrics and the interactions, and subsequently trigger its Provisioning, Configuration and Deployment in an automated way.

With InfraRED, the design the user makes on the Graphical User Interface (GUI) will serve not only as the method for creation of the system's topology but also the way to observe and manage its state after deployment.

The user will work with abstracted versions of various common network, compute and software/application elements. InfraRED is tasked with presenting the user with all the necessary virtual elements, from Customer-Premises Equipments (CPEs) to Network elements, Compute elements and connectivity, in order to build the desired infrastructure, and also with abstracted functions from adequate Application Program Interfaces (APIs) to interact with external software, hardware or cloud environments.

It is therefore possible to derive a few goals for InfraRED:

Goal 1) Abstract computation, network and software elements.

Goal 2) Provide easy access to those abstractions as “nodes”.

Goal 3) Allow customization of the characteristics and features of the “nodes” that the user wishes to deploy.

Goal 4) Allow for the definition of the connections between those “nodes” to represent how data or state transactions should be performed.

By having these goals fulfilled, InfraRED will be capable of sustaining itself. The nodes that fuel the system will be created by users that are more tech savvy and then added to InfraRED’s public library. Other users, with varying technological capabilities, will then have access to the library of nodes to use and construct their desired design which should be as simple as possible.

1.2 Organization of the Document

The remainder of this document is organized as follows:

Chapter 2 goes over present technologies that are core for the development of InfraRED and also for those that aim to do something similar to what InfraRED aims to achieve, also shining some light on how they fare in comparison.

Chapter 3 goes over the design choices made to bring InfraRED to life.

Chapter 4 details the various algorithms and systems in place to run InfraRED.

Then, on chapter 5, an evaluation of the system will be performed that will show some qualities and disadvantages of the way InfraRED was designed.

Finally Chapter 6 summarizes InfraRED’s achievements and reflects on the future of this solution, as well as proposes some improvements that allow InfraRED to go beyond its current proof-of-concept state, describing features that were not implemented due to time constraints.

2

State of the Art

Contents

2.1 Background 7
2.2 State of the Art 9

This chapter introduces the concepts, the technology and the academic work that guided InfraRED in its conceptualization and implementation.

The development of the InfraRED solution took inspiration from some previous works, that guided its development method. Exploring those academic work about cloud, *Low Code* and IoT, allowed InfraRED's design choices to evolve in a concise way.

2.1 Background

It is important to lay out some concepts that are a baseline for the technology developed in InfraRED, these will provide the necessary context to understand the ideas and challenges tackled in this work.

2.1.1 Low Code

Low Code is a coding paradigm that focuses on giving more people—especially ones without expert level coding skills—the ability to construct solutions for their projects by means of a visual tool that involves minimal code writing in common high-level languages such as Python, Java or C. With this coding model, programmers or users normally have a visual drag-and-drop technology of element blocks representing the intended logic, and code is automatically built by generating it from the interpretation of a visual design created by the user [3].

In *Low Code* the focus is mostly on information flow, on most implementations, since the user creates directional connections between elements, for example, in a connection from component A to component B the user *intends* to disclose that the information can flow from one to the other. On those elements, the information or data can be then modified before passing it to the next element. In this sense, every component is tasked with processing data in a specific way. This functionality can be achieved by providing the user with predefined processing elements (some with even complex logic or functionality) or by giving space for high level language writing in order to more precisely cater to the users' needs [2].

With its visual flow based focus, the *Low Code* paradigm helps developers understand different pieces of code by simply observing the interactions between the element blocks and understanding the data flow. In this paradigm the code is auto generated based on the design the user created, combining the *deployment* and *testing* phases into one single process [2].

2.1.2 Cloud Infrastructure

A Cloud infrastructure is defined as the hardware and software components that compose a system accessible over the Internet. These elements can typically be servers, storage and networking [4].

One of the advantageous aspects of the cloud concepts is that it offers processing and storage power at large but then, on top of said power, the users can apply logical connections to derive desired functionalities. Cloud companies can also provide costumers access to plans and/or services where they sell said power on a pay-per-use basis.

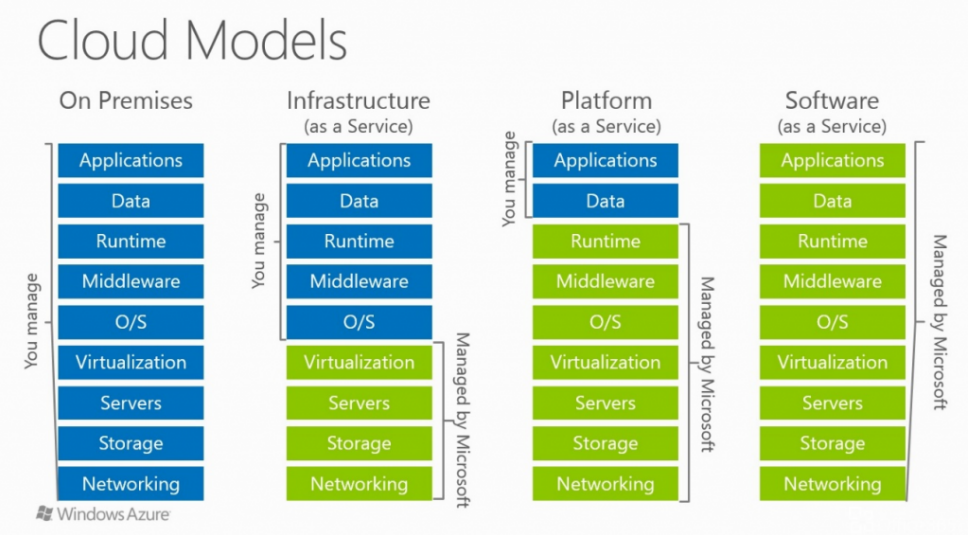


Figure 2.1: Cloud Service Providing. Source: [5]

There are a few business models for cloud providers, with the most common being Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS), offering, respectively, an increasing number of more specialized and pre-built services, as illustrated in Figure 2.1.

IaaS focuses on giving users access to networking and computing resources on demand, cancelling the need for costumers to buy hardware themselves.

PaaS normally provides the customer with access to a framework in which they can develop applications, removing the precise access to compute resources that IaaS gives but still giving control to the costumer on what applications and what functionalities said applications will provide.

For SaaS the cloud vendors handle all the details and all of the back end elements, while the costumer simply purchases access to applications offered as-a-service [4].

2.1.3 Internet of Things (IoT)

The concept of IoT comes from connecting simple physical devices, such as sensors, controllers or actuators, to the Internet, so that these can communicate and exchange information to achieve a combined goal on a complex system, which can range from a Smart Home to an industrial manufacturing line [6].

Despite the chip shortage happening around years 2020/2021 [7], IoT is a fast growing sector of technology but its growth is lacking when it comes to uniformization or standardization of communication

protocols and computing/management centers because most brands use a vertical model, by which they build their sensors and systems to only function under their dedicated software and hardware [8], often not allowing interoperability with other brands' devices.

This situation comes as a customer problem, as there is lack of flexibility when it comes to building an IoT network since the devices are only able to communicate to their brand's controller. When this communication is possible, it is not done in an uniform way, as the user has to interact with many different interfaces to connect each device, and sometimes the full functionality of each device is not available through a more flexible connection [6].

Al-Fuqaha et al. [6] mention the expected exponential growth of IoT devices in all sectors, despite the predictions made in 2015 being now extremely incorrect—and the growth not as powerful as predicted—the importance of IoT devices in all of the world can still be seen.

The study shows that the biggest percentage of devices are located in the manufacturing and health sectors, as those sectors require a great deal of network management as there are many equal devices, all communicating. With InfraRED it is possible to provide visual management to these types of systems and even add extra functionalities.

Observing recent values in comparison to the 2015 predictions for the number of active IoT devices, many factors came into play for decreasing the amount of devices actually connected to the Internet in present day, and just in 2020 the chip shortage caused by the SARS-CoV-2 pandemic put a dent on the growth of IoT manufacturing. Another part of the prediction that failed was the idea that all of these devices would have an easy time being connected to the Internet which has not been yet made possible due to all the network and protocol differences [7].

2.1.4 Intent Based Networking (IBN)

IBN is used to achieve network administration and creation without having to think about its components (and their, sometimes complex, details) individually. The IBN purpose is therefore to be able to define a network and its functionalities by using the user's desired end configuration for the network. With this automated behavior comes error validation, and so the IBN software is responsible for making sure the *intents* are viable and possible to be translated to deployments within the physical network, and if so, then launch the provisioning of the desired topology [9].

2.2 State of the Art

To understand how InfraRED can fit into the current cloud technology environment it is important to go over the main ”competitors” and similar technologies to InfraRED and observe what ideas are the foundation for it.

2.2.1 Node-RED

Node-RED [10] is the main inspiration of InfraRED and is the technology that started InfraRED with the idea of using nodes for representing interconnected elements. Node-RED is therefore a node-based *Low Code* programming tool that provides a large library of node types to interact with various types of systems. New nodes in the library are created by users with higher technological skills, since they need to be able to read documentation and have programming skills to code. These users then share the created node a the public library.

Node-RED's main elements, as already mentioned, are the nodes, which can be hardware devices, APIs or online services. These nodes can be wired together and through those connections they share messages with each other creating flows.

To add a new node to the public library, a savvy programmer needs to code the node. If the new node needs to interact with an already present node, the programmer has to read the node's page to learn about what messages it shares during execution.

The fundamental idea of Node-RED is that it is flow-based, this means there are only three types of nodes in any flow: input, output and transformation (intermediate) nodes.

The Node-RED's dashboard is shown on Figure 2.2. In that figure it is possible to observe that it contains a left vertical bar where all the node types reside and from where the user can drag them to the flow drawing area.

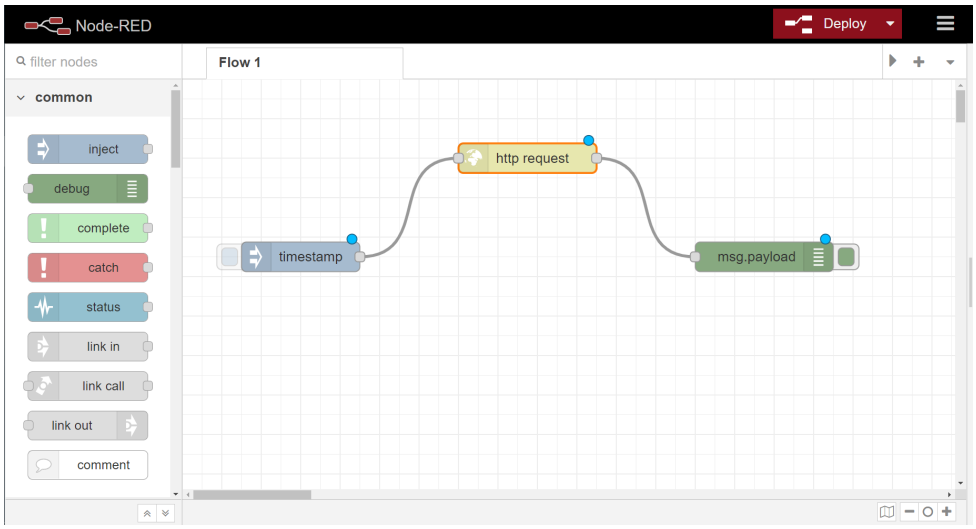


Figure 2.2: Node-RED Dashboard. Source: [10]

Input nodes are the ones that start the flow.This can happen with some sort of trigger from an external system to Node-RED, for example, doing a HyperText Markup Language (HTML) GET request or getting a signal from an IoT sensor. The input node will then pass messages to nodes along the flow, and then, intermediate nodes can alter the messages sent along the flow or serve as a conditional for

the progression of the flow based on said messages.

At the end of the flow is where the output nodes are placed. These are responsible for interacting with systems outside of Node-RED. Following the same example, this can be sending the response to a HTML request back to the server or activating an action on an IoT device.

In the example illustrated in Figure 2.2, the timestamp node is an input type node, this node sends a text message towards the `http response` node which transforms the message by placing the input into a query for a Uniform Resource Locator (URL) that was previously set up. This node will then pass the response to the output node which simply emits the message.

To create the connections between the nodes there is a small square that represents a flow input if placed on the left and a flow output if placed on the right. Figure 2.3 illustrates this, showing a completed relation between the *timestamp* node as the output and the `function` node as the input and a connection being created between the `catch:all` node and the `function` node.

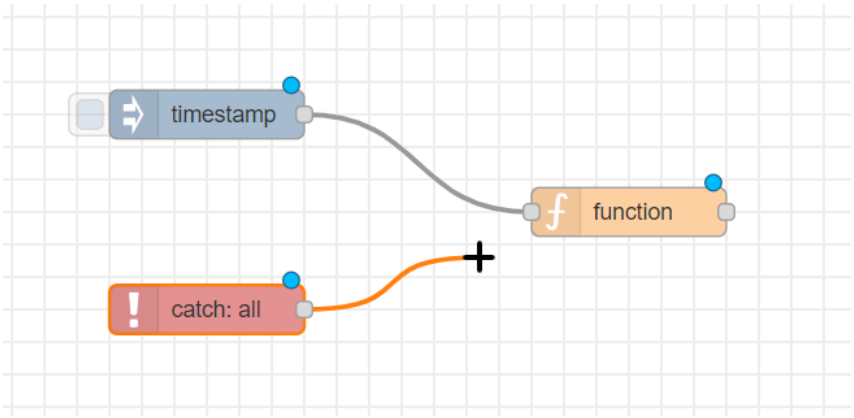


Figure 2.3: Node-RED Relation. Source: [10]

2.2.2 TOSCA

The Topology and Orchestration Specification for Cloud Applications (TOSCA) [11] is a standard developed by the Organization for the Advancement of Structured Information Systems (OASIS), for creating and maintaining cloud services. TOSCA's goal is to have a cloud service description that is independent of how the cloud environment is implemented.

TOSCA describes a service at the topology level by defining its components and the relationships between them. To aid in the orchestration and for service maintenance, the description also states the plans for the management of the service life cycle, such as creation and modification of the service. A TOSCA template [11] is called a service template (as in Figure 2.4) that includes a topology description and also the plans for orchestrating and managing that topology.

A topology template includes only two types of elements, Nodes and Relationships. On these,

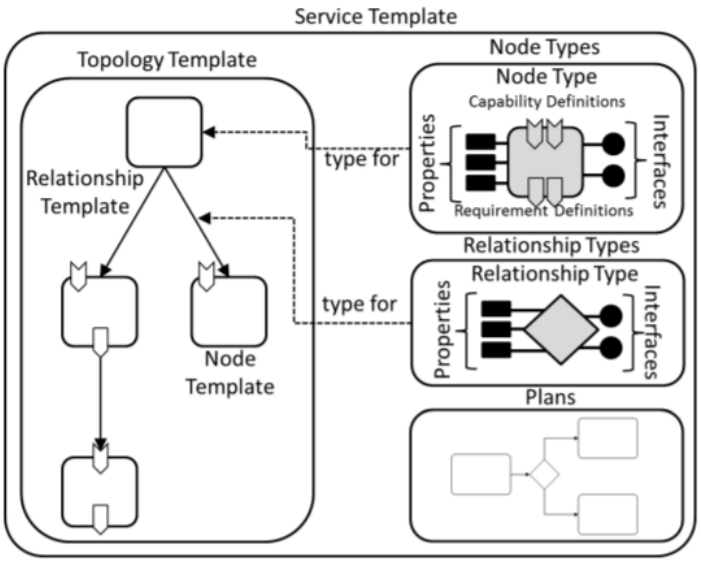


Figure 2.4: TOSCA Service Template Example. Source: [11]

TOSCA assumes a number of pre-existent types so as to create consistency throughout different users, as these would be simple and common elements like a *Compute*, *Network* or *Database* Node. As the language progresses, the community can define other base types that will then help future users with more types to choose from, instead of having to define a type as a more complex topology composed of these starter base types.

The strong point of a TOSCA template file is on providing a description for how the Nodes interact with each other, as Nodes have two main descriptors, as illustrated also in Figure 2.4.

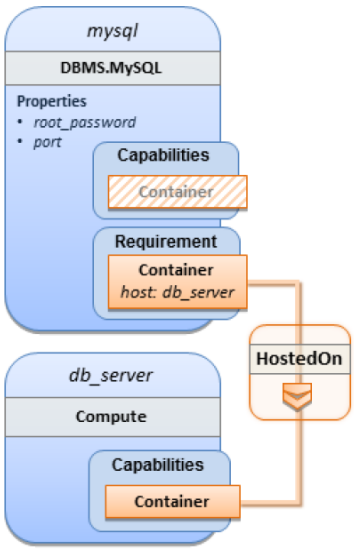


Figure 2.5: Relationship between TOSCA Nodes. Source: [11]

When it comes to Relationships between each other, these are the Capability and Requirement. Capability informs on what type of resources are offered by the Node, and Requirement gives the required Capabilities to be connected to the Node, necessary to its function.

Relationships between Nodes are crucial to the instantiating process because the topology defines which elements must be created first. If for example, as shown in Figure 2.5, the `mysql` database is hosted on a server, the server must be created first, and only after can the `mysql` database be created.

In Figure 2.5 the Node `mysql` has a requirement for a Container and the Node *db_server* has a capability of Container, which means this Node can provide for a Container requirement through a `HostedOn` relationship. These relationships will come from observing the connections made by the users in their design or created by default when, for example, the users describe a database in their system canvas, as exemplified by Figure 2.5.

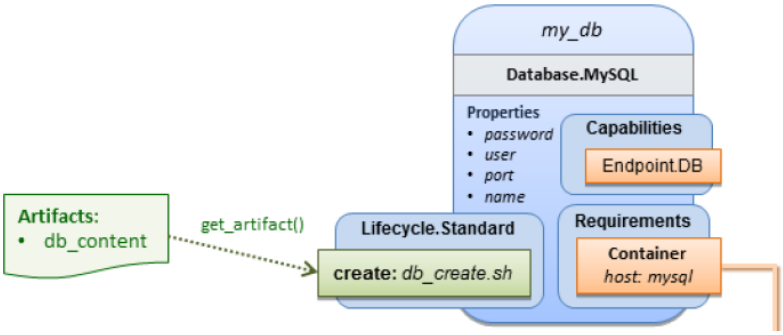


Figure 2.6: Node Artifacts. Source: [11]

As depicted in Figure 2.6, Nodes can also include files that take part in the Nodes' lifecycle, such as service deployment, implementation or runtime, and so they are called artifacts. If the user has to declare any external file to be used by a Node in the system, then this functionality will be used, making the file available inside said Node.

Nodes have requirements of values for initialization, for example when creating a compute node by dragging it into the canvas, the user gets a pop up menu in which the user needs to select a few mandatory properties, such as the machine's image (Ubuntu, CentOS, etc.), its computing power (the amount of virtual Central Processing Unit (CPU)s, storage and Random Access Memory (RAM) size) and to which network(s) it is attached.

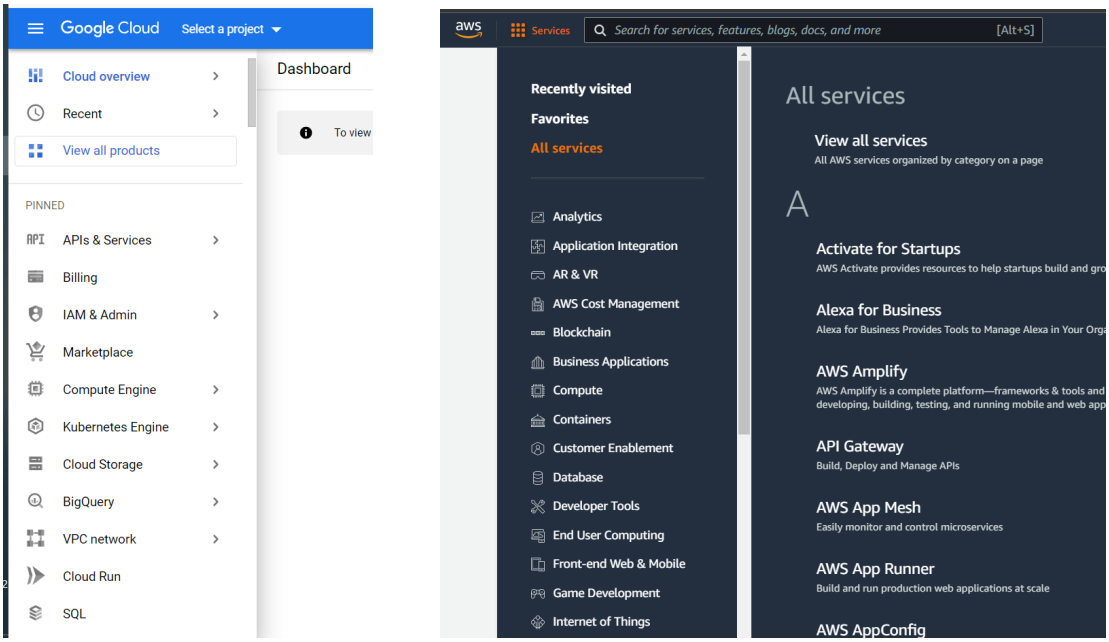
TOSCA's representation also allows nodes and connections to have their own properties.

2.2.3 Google's Cloud Platform (GCP) and Amazon Web Services (AWS)

A few companies offer solutions for creating cloud services (introduced in Section 2.1.2) in which the users have control of what infrastructure they wish to create and modify but no control over "where",

physically, the infrastructure is deployed. The major players in the market that sell these IaaS and PaaS models are Google, with its Google's Cloud Platform (GCP), and Amazon with its Amazon Web Services (AWS).

Both providers offer packages that enable to create different types of services, such as databases, block storage or computation, onto the cloud, typically in a web-based dashboard style menu. Both allow the users to find the services they want from a menu list as exemplified in Figure 2.7(a) (GCP) and Figure 2.7(b) (AWS).



(a) GCP Dashborad **(b)** AWS Dashboard

Figure 2.7: Public Cloud Platforms Dasboards

2.2.4 Metal as a Service (MaaS)

Metal as a Service (MaaS) [12] is a recent new service model that makes possible to deploy any type of Operating System (OS) on any type of bare-metal server rack as shown in Figure 2.8.

MaaS essentially expands on the idea of common cloud provider models, as typically those cloud providers offer services based on the already described three types of control, i.e., IaaS, PaaS and SaaS (as depicted in Figure 2.1), where in each level the user gives up on control over a part of the system for the sake of commodity, also allowing to reduce on Capital Expenditure (CAPEX), since the user does not need to purchase and set up the infrastructure. This leaves the on-premises solution option as the more "archaic" option for users, as these would have to buy all the servers, and networking devices, set

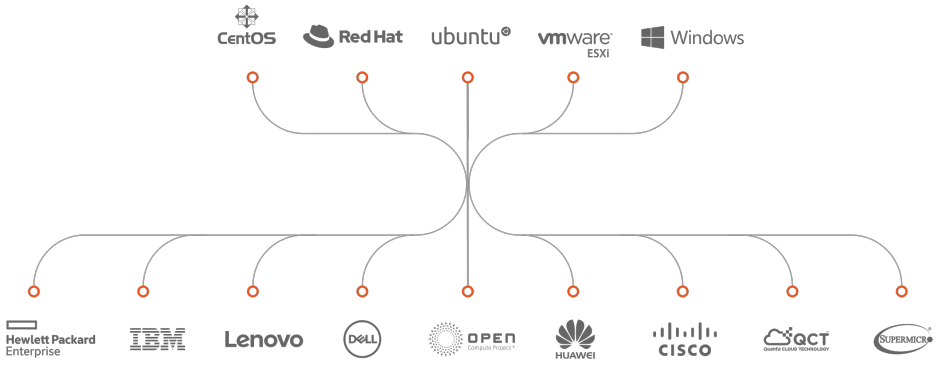


Figure 2.8: MaaS Compatibility. Source: [12]

up the needed electrical power, the racks, cabling, network connectivity, etc., and install the desired OS and software in each system.

However with MaaS that can be done more easily and with better organization while still allowing the user to have full control over their infrastructure, the servers, the storage and the networking between elements. Essentially MaaS aims to give the same flexibility and power of a cloud solution but by making use of on premises user equipment, the bare-metal system, to deploy an environment similar to the cloud which then makes it possible to deploy virtual machines and other pieces of software.

MaaS is often used for hybrid solutions, i.e., solutions mixing on-premises systems with public cloud environments. MaaS is also often used in conjunction with Juju, a cloud foundation solution, as described in the following Section 2.2.5.

2.2.5 Juju Charms

Juju Charms or Juju as a Service (JaaS) [13] is a Charmed Operator Framework, it is used to deploy cloud infrastructures and applications on bare-metal systems, being also responsible for many aspects of a service's lifecycle, its installation, upgrading and maintenance.

A Charm is responsible for managing a specific service's lifecycle. Charms can be coded by other users and added onto a public library, the CharmHub. Charms are written in Python and extend a CharmBase and a Status library which is needed in order to function as a correctly coded Charm.

Once a Charm is created, users can add it onto their Juju system and, on deployment, Juju handles the creation of the service defined in the Charm and manages any requirements of the service, creating them too. This relation between Charms is made possible because Charms can require or provide a service by exposing a logical connection capable of managing the relation between two Charms. Those relations can be exposing a data stream, setting up a network connection or any type of information exchange.

Juju Charms is mostly used to deploy already pre-built systems for a specific functionality, for exam-

Figure 2.9: Juju Dashboard. Source: [13]

ple, to build a complete OpenStack[1] system it is possible simply by downloading a deployment YAML Ain't Markup Language (YAML) file, called a bundle, that contains all the Charms necessary for the full OpenStack system. An example of such bundle is shown in Figure 2.9 for a Kubernetes[2] system.

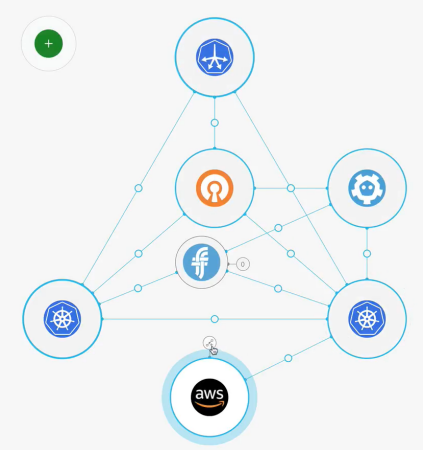


Figure 2.10: Juju Relation. Source: [13]

It is also possible to add Charms into a canvas and build a system from scratch with many Charms, adding the Charms as needed and making connections between them via a visual interface. This feature is illustrated in Figure 2.10, in which, to make the connections, the user must first click the node and then a pop up window will appear with options related to interactions with said node, where, one of them, called 'Build Relation', allows the user to build a connection between this node and a different one.

[1] https://www.openstack.org/
[2] https://kubernetes.io/

2.2.6 Sagitech Software Studio (S3)

Arora et al. [14] make emphasis on the standardization of rules in a *Low Code* system, a clear separation between the code design and the code generation, helping the system to migrate to new technologies, thus making upgrading the system easier and more accessible. For the user this can mean that the code generation is made more effective since the separations allows for generated code to be more modular, reducing the amount of duplicated code created per deployment.

3

InfraRED Design

Contents

3.1 Architecture 21
3.2 Infrastructure 23

This chapter will go through the design choices that were made on the project, establishing an abstract view of how InfraRED is laid out. This design should stand as a way to achieve the foundations of what InfraRED plans to achieve even if not all is presently achieved in code. To design InfraRED, inspiration was taken from Section 2.2.2 so as to establish a structure that is compatible with cloud infrastructure topologies since that is the goal of both the TOSCA project and InfraRED. TOSCA project will be mostly represented on a front facing and user interaction level, on the way a topology is laid out constituted of Nodes and Relationships which are concepts also used in InfraRED.

3.1 Architecture

To construct a system similar to tools like Node-RED [10] and Juju Charms [13], it is necessary to lay out a common structure in many applications, that expose an API, and that involves having a server and client separation. With that goal in mind, InfraRED's structure is a combination of three separations: the representation of infrastructure, which uses most of TOSCA's design, will be defined as Nodes, the Server side and the Client side.

The Server is the brains of the system, and this is the piece that controls all functionality of InfraRED, also holding information about all the Nodes, which are the individual components that make InfraRED.

The Client side is the front facing view of the system, where the user can manipulate the Nodes to create designs and then actions the Server to deploy them.

3.1.1 Server

The Server handles all interactions with the Nodes, as such it is tasked with deploying them, terminating them and saving them as a group, which is called a Pattern Node, as explained in Section 3.2.1.

To have Nodes available at the Server there will be files that the Server will load and initialize. The Server is then responsible for sending them to the Client for the user to see. The Server communicates with the Client via a Representational State Transfer (REST) API that it exposes.

To preserve Nodes across deployments and instances of InfraRED, the Server is responsible for saving Pattern Nodes making it possible for the user to have some persistence in InfraRED. This also aims to provide easier usability when the user wishes to deploy the same system across different times or in different locations.

3.1.2 Client

The Client side is structured to have a User Interface (UI) handling side and a functionality side. The UI side has the layout illustrated in Figure 3.1. The design canvas of InfraRED is the only frontal part

of the solution that the user will interact with. The current layout was chosen to be made similar to how Integrated Development Environments (IDEs) are structured, placing the resources and menus on the side and top of the screen, having the main interaction with the system happening at the center and placing status updates at the bottom.

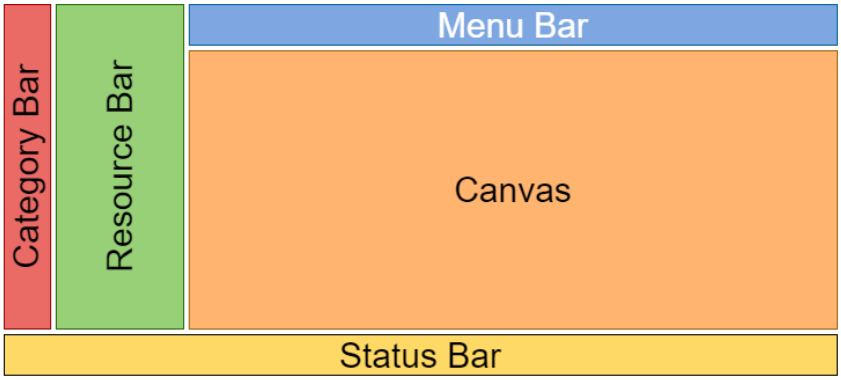


Figure 3.1: InfraRED Canvas Layout

In Figure 3.1 the green area, the Resource Bar, is where the user has access to multiple types of Nodes. These are shown in their normal form, grouped into categories that can be toggled via the red area, the Category Bar.

The users can then drag the Nodes onto the orange area, which is called the Canvas. In order to compose the desired topology, users make use of these Nodes that came from the green area, i.e., the Resource Bar.

The blue area, the Menu Bar, has buttons that allow the user to interact with the design being built, and the most important actions are to deploy the design in the Canvas or to save a Pattern Node.

Upon clicking on a Node that is present on the canvas, as introduced in Section 3.2.1, a menu will open to allow the user to interact with the properties of the clicked Node.

At the bottom, the Status Bar represented in yellow, will show status details about the current Canvas, such as information about resource usage and Node counts in the Canvas, so that through this bar, the user can observe if the system is functioning correctly and according to predicted values of resource usage.

The functionality side is responsible for handling the connections' logic and saving information about the current design being made in the canvas area. InfraRED holds information about all the Nodes that were loaded by the Server at the Client side.

3.2 Infrastructure

With InfraRED the aim is to design and build an IT infrastructure on top of a cloud or on local environment. To do so the system gives the user access to many Nodes which are representations of infrastructure, services or applications to be deployed on said environments.

3.2.1 Nodes

Nodes in InfraRED, which represent the infrastructure the user wishes to create, are a composition of three ideas, which are: the Node itself, its Connectables and the Relationships between them. This composition is detailed in TOSCA's idea of a topology, exemplified by Figure 2.4, and in InfraRED a simplified overview was derived to include the most necessary components that allow a cloud system to function.

For visual representation, Nodes will have a box like format and are structured to have their type shown at the top, making it possible to distinguish different Nodes, and will also expose the Connectables under the Type, with Capabilities and Requirements having different colors, as illustrated in Figure 3.2.

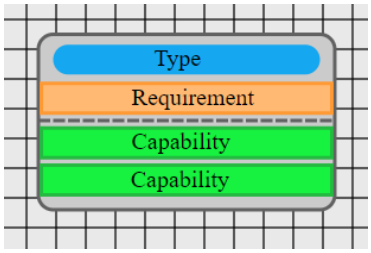


Figure 3.2: Node Structure

The main gist and goal of InfraRED comes from abstracting common cloud and local systems into Nodes, which are created via files of a standardized format that are then loaded by the Server, which is capable of using them as functional pieces. Each individual file is responsible for exposing a service, which has individual functionalities and can connect itself to other services to acquire necessary exterior functionalities. Nodes have their own properties that can be altered during design creation to change the Node specific properties that affect and alter their behaviour during and after deployment. To exemplify, there could exist a compute type Node with a property for the number of CPU cores it possesses or for the location where it should send its logs.

The Nodes are stored next to the Server, explained in section 3.1.1, as files that get loaded when the server starts and then makes them available to the user via the Client on the layout location for Nodes to be placed. At the location, Nodes will also be organized by categories.

Additionally, during user interaction, Nodes can be combined together for the sake of practicality as Pattern Nodes. This removes the need to individually set up groups of Nodes that can be repeated for

different scenarios, and so, only needing to manage a singular Node that encompasses multiple Node types and their Relationships, making it simple to create duplicates of complex structures.

3.2.2 Capabilities and Requirements

Nodes have Capabilities and Requirements, called Connectables. Exclusively, these are the pieces of a Node that allow it to connect to other Nodes. This connection, however, can only go from a Capability to a Requirement or from a Requirement to a Capability in order to be considered valid, since these connections represent a hierarchy of dependency where a Node with a Requirement must await a Node with a Capability if there is a connection between them. Connectables also have their own properties that allow users to customize the connection that can be built between them.

A Capability is the ability of a Node to offer a piece of functionality. This can be, for example, a database for holding tables or a processing power for a certain task.

A Requirement is the capacity of a Node to consume a piece of functionality that, if active, is a necessary requisite for the Node. Conversely, a tables Node requires a Node with a database Capability and a task Node requires a Node with processing power Capability.

3.2.3 Relationships

Relationships are what connects Nodes together. These Relationships do not hold or transpose any data from one Node to another. The Relationships are built in between the Connectables of the Nodes. The user can create these Relationships by joining Connectables of different Nodes together.

Visually, these Relationships will be represented by lines going from one Node to another and the lines will have their endpoints at the respective Connectables.

Their function is to show the system, the hierarchy of the Nodes that are currently in play, allowing to understand the order of deployment for the Nodes in a design, as will be explained in detail in Section 4.5.

4

Solution Implementation

Contents

4.1 Technology Stack 27
4.2 UI of the Client Side 28
4.3 Logic at the Client Side 33
4.4 Logic at the Server Side 36
4.5 Deployment Methods and Use Case 40
4.6 Development 46

This chapter describes how the design is implemented for a proof-of-concept of InfraRED, and the coded algorithms in place, making reference to what technologies are being used.

Any idea or concrete functionality that is not mentioned here should be perceived as a feature or element that is currently not yet programmed in the system, much of which will be discussed at the end, in Section 6.2.

4.1 Technology Stack

The entirety of InfraRED proof-of-concept will make use of the JavaScript[1] language. This choice of language comes from wanting to maintain similarity between how the Server and the Nodes function. JavaScript is the standard as a scripting language for web pages and since NodeJS[2] was the choice for server functionality, maintaining the client fully scripted with JavaScript felt like the most natural choice.

4.1.1 NodeJS and ExpressJS

The server will use NodeJS, which is a staple for developing server side applications with JavaScript. The idea behind this choice falls on NodeJS being an asynchronous event-driven JavaScript runtime, and because of those characteristics it would then be possible to remove wait times between processes that do not interact with each other, case being InfraRED's design of having many simultaneous interactions with different services' APIs.

ExpressJS[3] is also used to serve InfraRED's entry point and to handle requests coming from the client side. Since there is not a need for a complex server side, ExpressJS was chosen for its efficiency and simplicity of implementation.

4.1.2 Server

The Server exposes a REST API, and so, the entry points to the Server's functionality are properly exposed to the Client and data is transferred uniformly in JavaScript Object Notation (JSON) format between Server and Client. This implementation should allow for a different Client side to be developed as long as it conforms to the data transfer structure in which Nodes are represented.

4.1.3 Client

The Client side implements a combination of standard HTML, Cascading Style Sheets (CSS) and JavaScript in order to create and display a web page structure to the users that allows to interact with

[1] https://www.javascript.com/
[2] https://nodejs.org/
[3] https://expressjs.com/

InfraRED by means of visual buttons, input boxes and drag-and-drop actions.

For the drag-and-drop functionality, InfraRED makes use of JavaScript's JQuery[4] library. For CSS, Syntactically Awesome Style Sheets (SASS) are used so it is possible to have a better control over the site CSS with variables and file structures. On the Canvas, SVG.js 3.0[5] is used to enable any type of Scalable Vector Graphics (SVG) drawing.

4.2 UI of the Client Side

This section will go over how each individual piece of the UI functions and the layout choices made in order to achieve a tool that is easy to use. These pieces are materialized by individual files that are very simple and only control the creation and destruction of UI elements, as well as how drag-and-drop is handled, allowing for finer detail when it comes to upgrading InfraRED. This separation permits a modular approach making it possible to make alterations to any layout element without completely affecting the whole UI system.

The UI side makes use of both HTML elements and SVG elements. This decision was made since a complete HTML based web page would not be capable of handling many of the drawing decisions that were considered in order to correctly represent the Canvas design. To tackle said decisions the SVG.js 3.0 tool was used in order to implement SVG elements into InfraRED, mainly on the Canvas area.

The remainder of the layout is also composed with HTML and CSS, and only the Canvas makes use of SVG. For this to happen it was necessary to implement a translation from HTML to SVG that happens when a Node comes from the Resource Bar and is dropped into the Canvas.

A zoomed in example of the layout with which the user interacts is illustrated in Figure 4.1, showing that the layout follows the structure that was proposed at Section 3.1.2. The following sections discuss the implementations of the various layout parts.

4.2.1 Node

A Node is constructed in a way to show its Type name at the top, or a custom name if the user sets one via the modal box, as will be explained in Section 4.2.7. Following the Type, the Connectables are displayed, with Capabilities having a background green color and the Requirements having a background orange color. These Connectables also display their type name. To exemplify, in Figure 4.2, it can be observed a Node with the type `kalivm` shown in blue, one Requirement that is called `storage` and two Capabilities that are called `os` and `synced_folder`.

[4] https://jquery.com/
[5] https://svgjs.dev/docs/3.0/

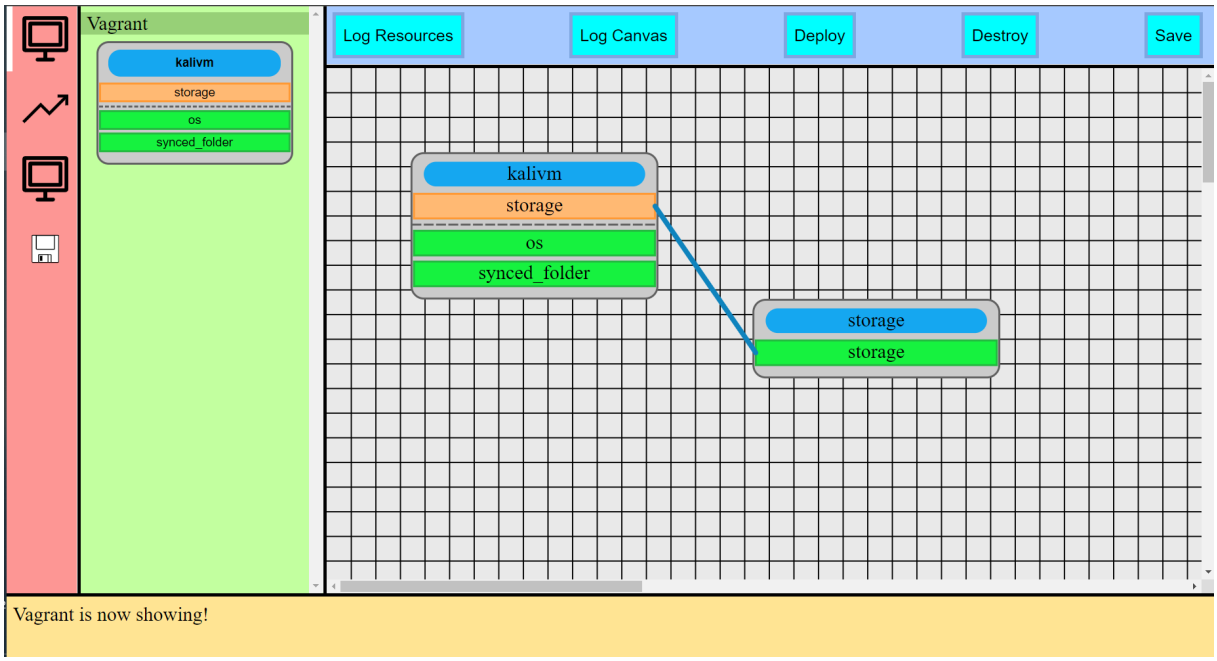


Figure 4.1: InfraRED Dashboard

The box like format allows InfraRED to provide a representation that gives the necessary information without cluttering the screen with extra information that is not essential for making the design process comprehensible.

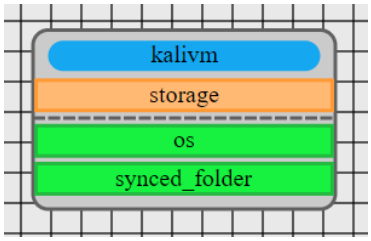


Figure 4.2: Node Example

To move the Nodes, both from the Resource Bar to the Canvas and in the Canvas itself, the user simply needs to hold the left click mouse button (or equivalent for a touch type screen) and the cursor will change from a selecting cursor to a grabbing cursor. When this happens, the border of the Node changes from grey to blue to signify that the Node is being held. Then the user just has to lift the left click mouse button (or equivalent) and the Node will stop moving. If the Node is coming from the Resource Bar then the user must drop the Node when fully inside the Canvas area, as otherwise the Node will not be placed. This functionality satisfies the drag-and-drop functionality that was proposed and is one of the foundations of any *Low Code* visual tool.

4.2.2 Canvas

To draw the Canvas the choice was made to give the user a spacious area, which is achieved in InfraRED through the use of a scroll-able environment. The implemented default size of a 2000 by 2000 pixels Canvas area for the proof-of-concept solution was chosen arbitrarily (for simplicity) but it can be easily changed to other values, similarly to what a tool like Node-RED [10] uses.

Since the user window will most likely be unable to show the entirety of the design in the Canvas, there are vertical and horizontal scroll bars to control the section of the Canvas that is being shown. Dragging a Node to an edge of the current section of the Canvas also allows the user to move that Node to the adjacent section of the Canvas without the need to touch the scroll bars.

The Canvas element further reacts to a Node being hovered above a valid dropping area. This is, for example, the case when the Node that the user is holding is fully inside the Canvas element, and accordingly its color is changed to visually inform the user that the Node can be dropped there and consequently be added into the current design.

4.2.3 Status Bar

The Status Bar is capable of showing the user text messages related to various types of information, such as statuses, warnings, errors, etc.

The implementation of the Status Bar for the proof-of-concept is minimal (for simplicity), being just capable of reacting to system changes, such as deployments and Pattern Node saves, or displaying text information about the current Nodes in the Canvas.

Changes of status are triggered by various events, as will be described in Section 4.3.3, and so, other logic pieces of the solution are the ones responsible for emitting such events. The Status Bar is completely reactive, meaning that it only waits for said events to then display the corresponding messages.

4.2.4 Category Bar

The Category Bar is built during the loading of the Nodes, and from there InfraRED is able to extract two pieces of information: the category name and its icon. InfraRED will then build for each category an icon based selector. This list in the Category Bar has scrolling functionality in case the number of categories does not fit the screen height.

By default, InfraRED shows as selected the first category of Nodes that was added to the system. The user can then click on any other category icon to view Nodes from that other category and stop seeing Nodes from the previous one. To allow the user to understand what category is selected a white line is shown on the left for that current category, as illustrated in Figure 4.3. When the user is hovering

each category icon, the button that is presently being hovered darkens to show that it is possible to click that category to change for it, as also illustrated in Figure 4.3.

After loading every category, InfraRED will create a special category exclusively for Pattern Nodes. This is the case of the last category shown in Figure 4.3, clearly informing that the *Vagrant* category can be seen as selected and the respective Resource Bar group is being displayed.

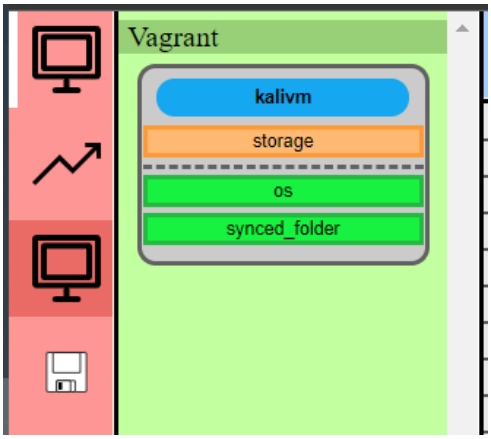


Figure 4.3: Categories Example

4.2.5 Resource Bar

The Resource Bar contains all the Node groups that were loaded into InfraRED but only shows one group at a time based on category selection. The Nodes that are shown in the Resource Bar can be dragged to the Canvas to build the desired design. If the number of Nodes in any category exceeds the screen size the Resource Bar also possesses a scroll bar for the user to view all the Nodes.

When the user drags a Node from the Resource Bar, the Node is not removed from the Bar but instead a copy is created, which shows the user that multiple copies of the same Node can be used at the same time and that it is not limited by number.

4.2.6 Menu Bar

The Menu Bar is the layout location that contains buttons that allow the user to interact with the design being made.

For the proof-of-concept solution only three buttons were created for functionalities that the menu is responsible for. These buttons are for “saving”, “deploying” and “destroying” what is being designed.

All the buttons in the Menu Bar emit events, as will be detailed in Section 4.3.3, that are then handled by the corresponding logic section. This way, the Menu Bar has no algorithmic logic and is only responsible for emitting events when a button click happens in any of the buttons that it contains.

4.2.7 Node Modal

Figure 4.4: Node Modal Box

By double clicking on a Node, a modal box opens, as illustrated in Figure 4.4, which allows the user to modify the properties of the Node or delete the Node from the Canvas. To exit the modal box the user simply has to click outside the modal box, noting that this discards any changes that were made, if not saved. To actually save any changes made to the properties of the Node the user has to click the (green) save button.

Changes that are made on this modal box are then reflected on the Client's memory so there is persistence of data for the user session, which will be explained in detail in Section 4.3.2.

4.2.8 Relationship Lines

A Relationship line is a (blue) line between Connectables created by the user, as illustrated in Figure 4.5.

To create a connection the user clicks any Connectable present in the Canvas. That Connectable then turns into a red color (that signifies the selection is being made) and an arrow is drawn from there. To complete the connection the user clicks another Connectable that is of the same type and of the opposing mode, i.e., if it is a Capability it can only connect to a Requirement and vice-versa.

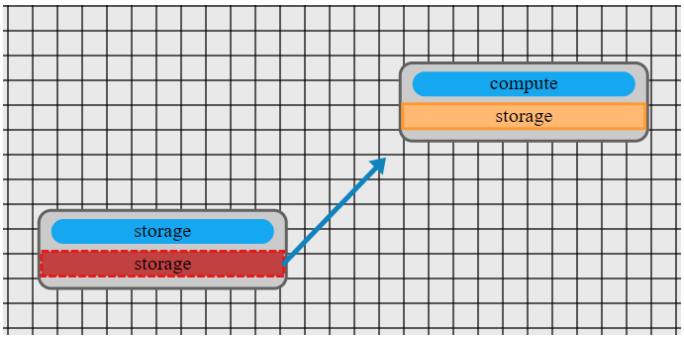


Figure 4.5: A Relationships (blue line) between Nodes

4.3 Logic at the Client Side

Logic wise, InfraRED's Client side has files to handle interactions with and between Nodes and sending information to the Server. The Client is also responsible for holding, in local memory, information about the Nodes that were loaded at the Server after it finishes its initialization.

4.3.1 Nodes and Relationships

For handling the main logical elements of InfraRED, there are three JavaScript classes: Node, Connectable and Relationship. These classes are capable of generating their respective HTML and SVG elements onto the Canvas and hold the information that is sent to the Server for deployment and for saving a Pattern Node (as can be observed in the code snippet of Listing 4.1).

Listing 4.1: Connectable and Node classes at client/nodes.js

```
class Connectable {
    constructor(mode, type, nodeID) {
        //select between requirement and capability connectable
        if (infraRED.validator.validateNodeMode(mode)) this.mode = mode;
        //type of the connectable
        this.type = type;
        //data for this connectable
        this.properties = {};
        //nodeID on the canvas of the parent node to this connectable
        this.nodeID = nodeID;
    }
    getDiv() {...}
    getSVG() {...}
    getPropertiesModalSection(propertyList) {...}
    updateSVG() {...}
    print() {...}
}
class Node {
    constructor(type) {
        this.resourceID = null;
        this.canvasID = null;
        this.type = type;
        //pattern functionality
        this.isPattern = false;
        this.patternMemory = null;
        this.nodeSVG = null;
        this.properties = {};
```

```
28          this.capabilities = {};
29          this.requirements = {};
30          this.relationships = [];
31      }
32      makePatternNode(patternComposition) {...}
33      addCapability(capabilityType, properties) {...}
34      addRequirement(requirementType, properties) {...}
35      addRelationship(relationship) {...}
36      getPropertiesModal() {...}
37      getDiv() {...}
38      getSVG() {...}
39      updateSVG() {...}
40      print() {...}
41  }
```

4.3.2 Node Database Memory

The Client holds in memory lists that contain information about the Nodes with which the user can interact. The first list in memory corresponds to a Node Resource list that gets built from the information about the Nodes coming from the Server and represents visual information about all the Nodes loaded in InfraRED. The second list corresponds to the Node Canvas list that stores the Nodes that get placed into the Canvas.

The Node Resource list stores the Nodes and distinguishes them with a sequential number identifier as every Node placed there will be unique. The Node Canvas list however will hold the same type of Node multiple times so the identifier chosen is a random number between 0 and a constant configured to be a maximum number of Nodes supported by the current system.

4.3.3 Events

The **events** module is used to map synchronous function callers to string constants. This means that in the Client side it is possible to add one or more function calls under a single event. This is a more human readable method than calling the functions from many different logic or UI files.

Lines 1 and 2 of Listing 4.2 show how to append a JavaScript function to a certain `event_name`.

Afterwards it is possible to invoke those functions by emitting said `event_name` which will make a call to all the functions under that name with the arguments given.

Listing 4.2: Event example from client/events.js

```
1 infraRED.events.on('event_name', functionName);
2 infraRED.events.on('event_name', anotherFunctionName);
3 infraRED.events.emit('event_name', functionArguments, functionArguments, ...);
```

4.3.4 Loading the Nodes

The **loader** module at the Client is tasked with making a GET call to the `/listNodes` endpoint, as illustrated in the code snippet of Listing 4.3, which contains no arguments. The Server responds with a list of objects representative of all the Nodes it loaded so that the Client can populate its local memory. From this, both the Category and the Resource Bar can populate their lists with the results from the loader which must happen before the user can interact with InfraRED.

Listing 4.3: GET call for /listNodes at client/loader.js

```
function getNodesFromServerRegistry() {
    let types;
    $.ajax({
        url: '/listNodes',
        dataType: 'json',
        async: false,
        success: function(data) {
            types = data;
        }
    });
    if (typeof(types) !== 'object') {
        throw "Couldn't fetch node list.";
    }
    return types;
}
```

4.3.5 Initiating the Deployment

The **deployer** module at the Client handles making a POST call to the `/deploy` endpoint, as illustrated in the code snippet of Listing 4.4, with POST data about all the Nodes that are currently present in the Canvas. This piece of data must be modified because in order to send data through JSON the object cannot contain circular references and those are present due to the SVG.js 3.0 library, so these references are removed.

Listing 4.4: POST call for /deploy at client/deployer.js

```
function deployNodes() {
    $.ajax({
        method: 'POST', url: '/deploy',
        contentType: 'application/json', dataType: 'json',
        async: false,
        data: JSON.stringify({
            'nodes': cleanUpCanvasList(false),
        }),
        success: function(data) {
            console.log(data);
        }
    });
}
```

The **deployer** module is also responsible for saving Pattern Nodes since the previously mentioned problem is also present when saving. For the saving functionality, the module makes a POST call to the `/save` endpoint, as can be observed in the code snippet of Listing 4.5, by sending a Pattern name, that

is acquired by prompting the user to insert a name in a text box, and the current Nodes present in the Canvas.

Listing 4.5: POST call for /save at client/deployer.js

```
function saveNodes() {
    try {
        let patternName = prompt('Please enter pattern name:', 'Default Pattern
            Name');
        if (patternName == null) throw new Error('Pattern has empty name.');
        let patternComposition = cleanUpCanvasList(true);
        $.ajax({
            method: 'POST', url: '/save',
            contentType: 'application/json', dataType: 'json',
            async: false,
            data: JSON.stringify({
                'name': patternName,
                'nodes': patternComposition,
            }),
            success: function(orderedNodes) {
                buildPatternNode(patternName, orderedNodes);
            }
        });
    } catch(error) {
        console.error(error);
    }
}
```

4.4 Logic at the Server Side

After receiving the API calls from the Client, the Server side starts its functionalities, which can be of deploying a topology of Nodes, saving Pattern Nodes or destroying a topology. These calls function asynchronously, as explained in Section 4.4.3, to allow for the actions to be more efficient, and to also remove the need to wait for Nodes' actions individually and sequentially. Each Node can activate its processes simultaneously as long as there are no special requirements for that deployment.

4.4.1 Node Files

The Nodes' files must conform to a specific structure and set of rules in order to be accepted and used by InfraRED's system.

The data properties that a Node must have are: category (name and icon), properties (any amount) and respective Capabilities and Requirements, each with its own properties.

An instance of a Node needs to expose a `deploy()` and a `clean()` method. The `deploy()` method is responsible for initializing the instance and the `clean()` method is responsible for destroying the instance after it has been initialized.

A Node file exposes a `create()` method from where the Server can get an instance of the Node and a `load()` method, which is responsible for any pre-deployment environment set up that the Nodes of this type requires.

The implemented Server initialization is capable of of detecting the lack of certain elements in the Node file, but it is not capable of, nor does InfraRED aim to, tackle malicious construction of Node files in order to exploit a language or InfraRED behaviour. In this sense, the security aspect of the Node files is not a focused for the implementation, and so,in this phase it is considered out of scope.

4.4.2 Receiving a Deployment Request

The **deployer** module at the Server is responsible for handling all deployment related functionalities which includes ordering the Nodes by their levels, and then proceeding to deploy each Node.

Listing 4.6 shows the entry point for a deployment action which is constituted by four steps.

Listing 4.6: Deployment entry point at server/deployer.js

```
async function deployNodes(nodesToDeploy) {
    if (Object.keys(orderedNodeInstances).length !== 0)
        await cleanNodeInstances(orderedNodeInstances);
    let orderedNodesToDeploy = orderNodesByHierarchy(nodesToDeploy);
    orderedNodeInstances = createNodeInstances(orderedNodesToDeploy);
    await nodeDeploy(orderedNodeInstances);
}
```

Firstly if there is already a running deployment, the Server does a cleanup, as shown on Listing 4.7. This action will proceed to call the `clean()` method of each Node that was deployed and is in the active state, doing so in the inverse order of its deployment.

Listing 4.7: Node clean method before deploying at server/deployer.js

```
async function cleanNodeInstances(nodesToClean) {
    for (let level in nodesToClean) {
        let cleanupPromises = [];
        for (let node of nodesToClean[currentMaxLevel - level]) {
            if (node.constructor.name == 'Pattern') {
                cleanupPromises.push(cleanNodeInstances(node.orderedInstances));
            } else {
                cleanupPromises.push(node.clean());
            }
        }
        await Promise.allSettled(cleanupPromises);
    }
    nodesToClean = {};
}
```

After verifying that the previous deployment was cleaned the Server proceeds to call the function. As illustrated in the code snippet of Listing 4.8, the Server starts ordering the Nodes that the user wishes to deploy, which is achieved by verifying the Relationships between these Nodes, and then creates each level as a map list. This algorithm is detailed in Section 4.5. The maximum level of the deployment list is also established, which aids in the cleanup process.

Listing 4.8: Creating the level lists at server/deployer.js

```
function lookupRelationships(level, orderedNodes, nodesToDeploy) {
    let currentLevelList = [];
    for (let nodeIndex = 0; nodeIndex < nodesToDeploy.length; nodeIndex++) {
```

```
            let node = nodesToDeploy[nodeIndex];
            let nodeID = node.canvasID;
            let appendToCurrentLevel = true;
            let requirementsCount = 0;
            let requirementsFulfilledCount = 0;
            for (let relationship of node.relationships) {
                if (relationship.requirement.nodeID === nodeID) {
                    if (level === 0) {
                        appendToCurrentLevel = false;
                        break;
                    } else { //level 1+
                        requirementsCount++;
                        for (let levelIndex = 0; levelIndex < level; levelIndex++) {
                            for (let capabilityNode of orderedNodes[levelIndex]) {
                                if (relationship.capability.nodeID === capabilityNode
                                    .canvasID) {
                                    requirementsFulfilledCount++;
                                }
                            }
                        }
                    }
                }
            }
            if (appendToCurrentLevel && requirementsCount ===
                requirementsFulfilledCount) {
                currentLevelList.push(node);
                nodesToDeploy.splice(nodeIndex--, 1); //remove it
            }
        }
        return currentLevelList;
    }
```

With the level list built, the Server starts creating instances of each Node, as shown in the code snippet of Listing 4.9.

Listing 4.9: Creating the Node Instances at server/deployer.js

```
function createNodeInstances(nodesToDeploy) {
    let orderedInstances = {};
    for (let level in nodesToDeploy) {
        orderedInstances[level] = [];
        for (let node of Object.values(nodesToDeploy[level])) {
            if (node.isPattern) {
                orderedInstances[level].push(new Pattern(createNodeInstances(node
                    .patternMemory)));
            } else {
                let newNode = registry.getRuntimeList()[node.type].create();
                newNode.properties = node.properties;
                for (let capability in node.capabilities) {
                    newNode.capabilities[capability] = node.capabilities[
                        capability].properties;
                }
                for (let requirement in node.requirements) {
                    newNode.requirements[requirement] = node.requirements[
                        requirement].properties;
                }
                orderedInstances[level].push(newNode);
            }
        }
    }
    return orderedInstances;
}
```

With all the instances positioned at the appropriate levels, via the method described in the code snippet

of Listing 4.10, the Server goes through the level list and calls the `deploy()` method of each instance, while respecting the levels by waiting for a full level to be completely deployed before deploying the next, and iterating through this process until all levels are deployed.

Listing 4.10: Deploying the Nodes at server/deployer.js

```
async function nodeDeploy(nodesToDeploy) {
    for (let level in nodesToDeploy) {
        let currentLevelDeployPromises = [];
        for (let node of nodesToDeploy[level]) {
            if (node.constructor.name == 'Pattern') {
                currentLevelDeployPromises.push(nodeDeploy(node.orderedInstances)
                    );
            } else {
                currentLevelDeployPromises.push(node.deploy());
            }
        }

        await Promise.allSettled(currentLevelDeployPromises);
    }
}
```

While creating the Node instances and a Pattern Node is detected, the internal memory is parsed, and as it contains data about all its inside Nodes and Relationships, the function at Listing 4.9 calls itself recursively with that data as argument.

The created design can then be perceived as a Pattern Node too, which was decided in order to maintain consistency on how a design is deployed and how a Pattern Node inside the design is deployed, which, through this implementation, keeps these two situations equal.

When deploying the Nodes from a Pattern the function at Listing 4.10 is called recursively.

4.4.3 Asynchronous Behaviour for the Server

When it comes to providing asynchronous behaviour to InfraRED, the Server makes use of JavaScript's Promises.

A Promise is simply an object that checks the success or failure of an asynchronous task. In InfraRED it is used to call the `deploy()` and the `clean()` methods of the Nodes, and so, it is possible for each Node to launch these processes without having to sequentially wait for each other. This is done by using the `allSettled()` method from the Promises library, which awaits the completion of all tasks inside the list provided as an argument, as demonstrated in the code snippets of Listing 4.7 and Listing 4.10.

In the case of InfraRED, this list will contain, for example, all the deploy tasks of Nodes in a given level, so it is possible for the next level to only start when all the Nodes in the previous level finish successfully.

4.5 Deployment Methods and Use Case

This section will go over how a deployment is handled at the Server and also describes a possible and simple Use Case for InfraRED.

Logic-wise, the deployment is very simple, since the Nodes are dependent on one another if they contain Relationships between each other. A Node that contains any amount of Requirements must wait for the Node or Nodes with the respective Capabilities to be deployed and only then can this Node be deployed if all its Requirements are fulfilled by the previously deployed Nodes.

When the deployment process starts, Nodes that are in play get put on a deployment list. The Deployment is separated by levels, where each level is constituted by Nodes that have no deployment dependencies in Nodes of previous levels and may or may not have dependencies in Nodes of future levels. To create Level 1, InfraRED cycles the deployment list looking for Nodes that have no Requirements and therefore are not dependent on any other Nodes' deployment. Once all the Nodes are cycled, InfraRED will have Level 1 complete. Level 1 Nodes are then taken out of the deployment list and put on Level 1 list. For Level *N* InfraRED cycles the deployment list searching for the remaining Nodes. This time, InfraRED checks to see if the level lists less than Level *N* fulfil all the connections that the current Node that InfraRED is looking at needs. If they are fulfilled, that Node gets added to Level *N* and removed from the deployment list, and if not it stays in the deployment list to be added to a level after *N*.

4.5.1 Deployment Example

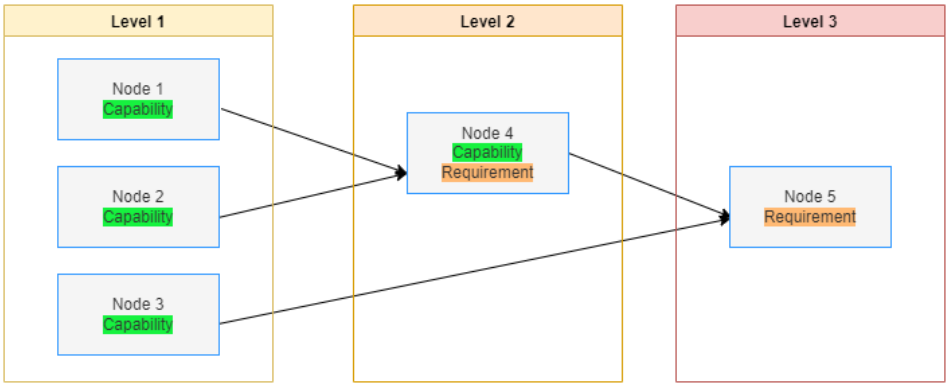


Figure 4.6: Deployment Sequence per Level

To better understand how a deployment is usually processed, lets go over a hypothetical deployment cycle, as illustrated in Figure 4.6, where Node 1, Node 2 and Node 3 have no active Requirements so they get put on Level 1. The remaining Nodes stay on the deployment list.

In that figure, Node 4 checks the Level 1 list and confirms that all Nodes it needs are already de-

ployed, so, it gets added to Level 2. Node 5, however, requires Node 4 to be deployed at Level 1 or lower, which is not, since InfraRED just added Node 4 to Level 2.

Afterwards, Node 5 stays on the deployment list. This time Level 3 is being created and now Node 5 gets fulfilled by Node 3 from Level 1 and Node 4 from Level 2, so then, it is possible for Node 5 to be placed on Level 3.

Since there are no more Nodes in the deployment list, the system has completed analysing and creating the level lists for the given design.

4.5.2 Client Side Process

The Client gets sent a singular JavaScript file called *infraRED.js* that is a bundle of many JavaScript files separated as UI and logic files.

Before bundling the JavaScript files together into one, the code is separated into its areas of functionality, as shown in Figure 4.7. This unbundled view allows to better showcase how the logical elements interact with each other inside the Client.

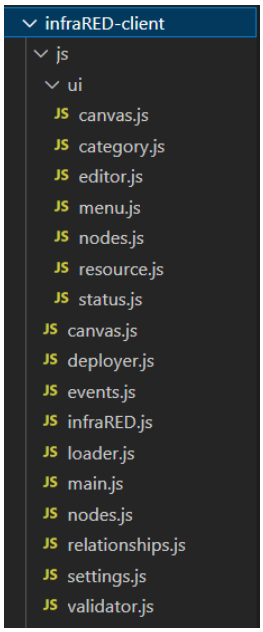


Figure 4.7: Client File Structure

The unbundled view serves as an effort of good coding practices relying on separating the logical components that are at work in the Client side so it is possible for modular upgrades to the pieces that make up InfraRED. These pieces are then meshed together via the use of the Events to allow communication between elements as long as the event names are propagated correctly through InfraRED.

The files are bundled together for distribution by using a tool called Grunt[6] to complete development,

[6] https://gruntjs.com/

so when the users is making use of InfraRED they only see one file sent by the Server.

4.5.3 System Initialization

To begin the initialization process, the Server searches for all the available Node files, by traversing a directory that contains all Nodes as illustrated in Listing 4.11, then validates the composition of the Node file to see if it is created correctly and subsequently invokes the `load()` method of each one.

Listing 4.11: Searching for Node files at server/registry.js

```
function buildNodesFullPathList() {
    nodesFullPathList = traverseDirForFiles(path.join(__dirname, '../nodes'));
    return nodesFullPathList;
}
function traverseDirForFiles(dir) {
    let fileList = {};
    fs.readdirSync(dir).forEach(file => {
        let fullPath = path.join(dir, file);
        if (fs.lstatSync(fullPath).isDirectory()) {
            fileList = Object.assign(traverseDirForFiles(fullPath), fileList);
        } else {
            fileList[file] = fullPath;
        }
    });
    return fileList;
}
```

All the successful loads get put into a runtime variable and the rejected ones get discarded, as shown in the code snippet of Listing 4.12.

The process of Node loading has another important piece of functionality which is the process of creating the front end representation of a Node. This is handled by the functions described in the code snippet of Listing 4.12 where InfraRED goes over checking the file for necessary information such as category name and Node name and if all of those are present it creates a variable that is added to a list with the only the necessary elements to show to the user.

Listing 4.12: Builing Runtime list of Nodes at server/registry.js

```
async function buildNodesRuntimeList() {
    buildNodesFullPathList();
    let loaderPromises = [];
    for (let nodeFile of Object.keys(nodesFullPathList)) {
        let nodeLoader = loadNode(nodeFile);
        loaderPromises.push(nodeLoader);
    }
    await Promise.allSettled(loaderPromises).then((results) => {
        results.forEach((result) => {
            if (result.status === 'rejected') {
                console.error(result);
            }
        });
    });
}
function loadNode(nodeFile) {
    return new Promise(async (resolve, reject) => {
        let nodeFileInfo = nodeFile.split('.');
        let nodeName = nodeFileInfo[0];
        try {
```

```
21              if (nodeFile.match(/^[A-Za-z]+\.js$/) === null) {
22                  throw new Error('...');
23              }
24              nodesRuntimeList[nodeName] = require(nodesFullPathList[nodeFile]);
25              if (!(nodesRuntimeList[nodeName].hasOwnProperty('create') &&
                    nodesRuntimeList[nodeName].hasOwnProperty('load'))) {
26                  throw new Error('...');
27              }
28              await nodesRuntimeList[nodeName].load();
29              buildResourceList(nodeName);
30              resolve(nodeName);
31          } catch (error) {
32              delete nodesRuntimeList[nodeName];
33              logger.log(`Failed to load node from ${nodeFile}.`);
34              reject({
35                  error: error.message, who: nodeFile,
36              });
37          }
38      });
39  }
```

The Server then sends properties of the Nodes it loaded to the Client for it to build the Node list in the Resource Bar and their respective categories in the Category Bar. During these two actions the user cannot interact with the Client side and experiences a loading time.

4.5.4 User Interaction

Once the Server and Client are fully booted up, the users can start interacting with Nodes from the Resource Bar and on the Canvas to build the design they wish to deploy. The user's possible actions can then be listed and summed up as follows:

Adding a Node to the Canvas: To add Nodes into play in order to compose a design, the user must drag them from the list of resources containing all the Nodes present in the system, and onto the Canvas as illustrated in Figure 4.8, the place where the user is designing their topology.

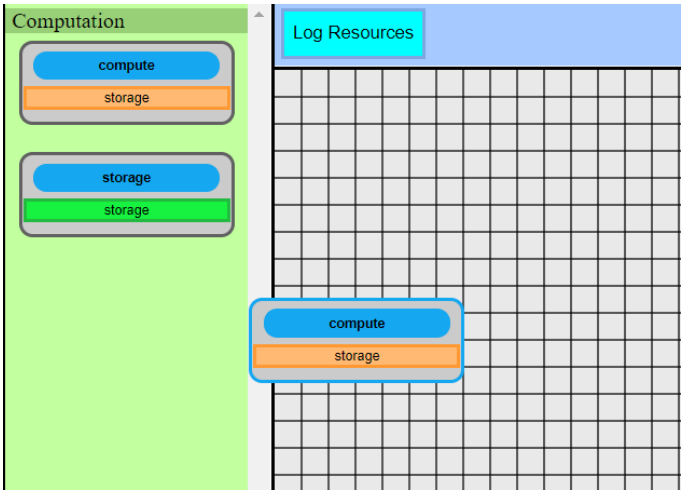


Figure 4.8: Adding a Node to the Canvas

Establishing Connections: Nodes have Requirements and Capabilities and if the user clicks on them they can connect them to other Nodes' Connectables establishing a Relationship between the Nodes (Figure 4.9).

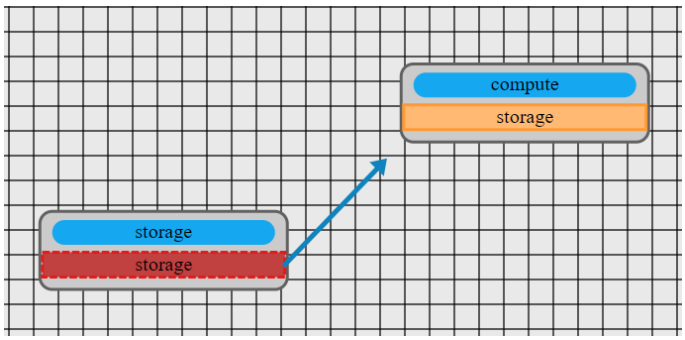


Figure 4.9: Establishing Relationships between Nodes

Interact with Node Properties: By double clicking on a Node, the user is met with a modal box (Figure 4.10) that exposes the Nodes' properties and shows additional information about the Node and contains buttons to interact with it (Figure 4.11).

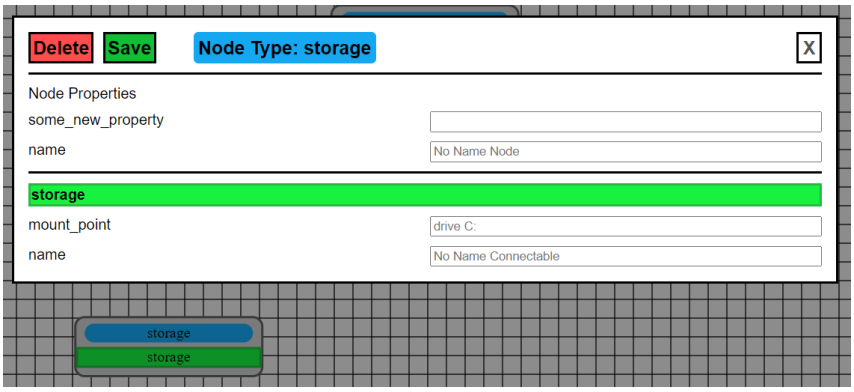


Figure 4.10: Interacting with Node Properties

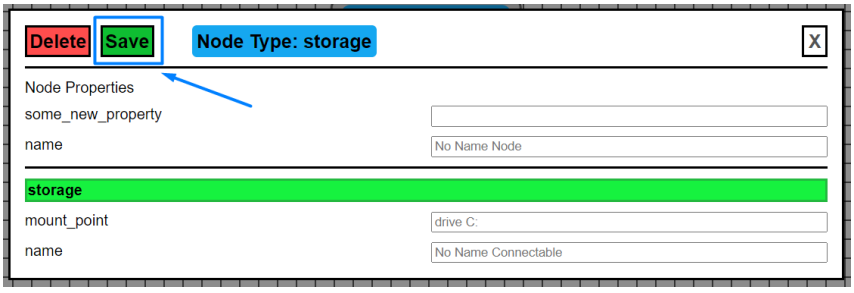


Figure 4.11: Saving Node properties

Delete Nodes: On the modal box of each Node there is a delete button, illustrated by Figure 4.12, at the top left that removes the Node from play. Additionally, for saved Pattern Nodes on the Resource

Bar, it is possible to delete Saved Patterns from the Resource Bar, in case the user does not wish to use them anymore (Figure 4.13).

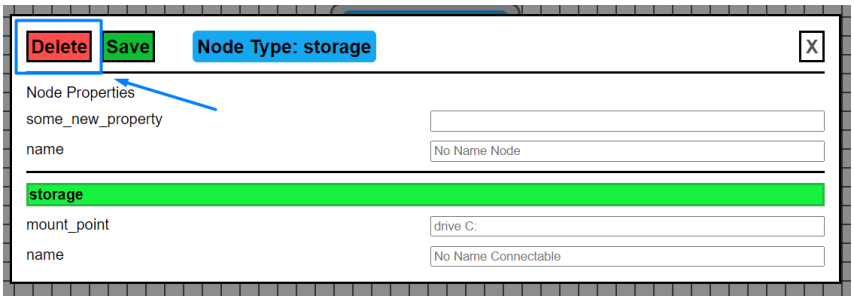


Figure 4.12: Deleting a Node from the Canvas

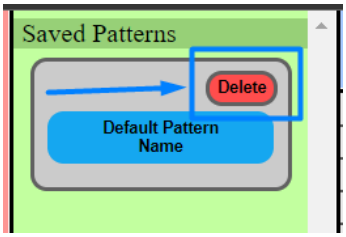


Figure 4.13: Deleting a Pattern Node

Save a Pattern Node: It is also possible to save the Nodes in play as a Saved Pattern Node, meaning that the design is condensed into a single Node that contains all the connections and properties the user made previously. Saved Pattern Nodes themselves are unable to contain other Saved Pattern Nodes, so, if a user tries to create a Saved Pattern it fails if there is a Saved Pattern Node present in the design (Figure 4.14).

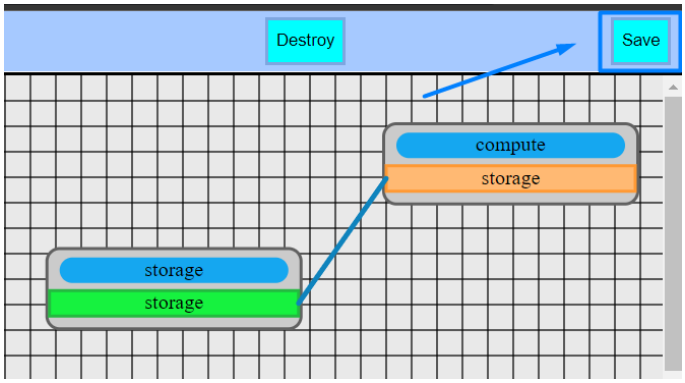


Figure 4.14: Saving a Pattern Node

Deploy: At any moment the user can press the deploy button on the Menu bar to start the deployment process. This sends data about the current Nodes in play (present in the Canvas) to the Server

from the Client. The Server has an algorithm to decide which Nodes get deployed first based on how they are connected to each other (Figure 4.15).

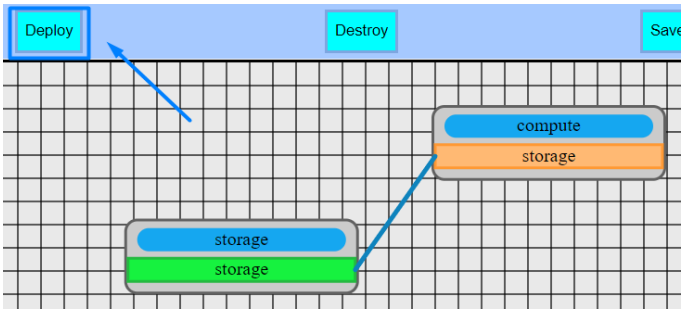


Figure 4.15: Deploying a design

4.6 Development

The code for this project is available on a public repository on Github, allowing readers to match the implementations described in this chapter. The repository is available at [15].

5

Evaluation

Contents

5.1 Installation and Setup 49
5.2 Loading Nodes 50
5.3 Node Type Availability 50
5.4 Usability 51

There are similarities between InfraRED, Juju Charms [13] and Node-RED [10] so in this chapter the three systems will be evaluated to see how they compare, in basic and common functionalities, to each other. The evaluation will not have a performance focus since the goal of InfraRED is for a system capable of combining *Low Code* and the TOSCA standard interpretation of cloud-based IT infrastructures design, deployment and orchestration.

Additionally, any Performance tests would not be significantly comparable, since Node-RED and Juju Charms differ in the scope of what type of elements they interact with. Node-RED interacts with IoT devices and some cloud systems but does not deploy them while Juju Charms is mostly tasked with deploying cloud infrastructure and installing software on said infrastructure, and so, a performance evaluation would not be able to showcase the differences between the three solutions, as only a little subset of functionalities would be comparable, and those functionalities are not among the important ones (related with IT deployment and orchestration), and so, the option was to showcase just a more visual comparison and usability based comparison.

5.1 Installation and Setup

This section explores how the setup differs in the three systems. With this, it should be possible to infer at what level a user with low levels of proficiency would be capable of installing and using these systems.

For InfraRED and Node-RED there is no user setup requirement after installation. Upon initialization the user is presented with a fully functioning system that is ready to be used at any moment.

For Juju Charms, however, it requires that the user previously sets up a cloud-based environment, either in a public cloud provider or in a local MaaS environment,. This is typically done using command line tools. Only then will the user have access to the visual dashboard to create the desired system on top of that cloud environment.

An end user with limited technical knowledge, is able in most cloud-based systems to just use the provided web page dashboard that controls the system, requiring therefore no installation actions or hardware setup from the user, but just requesting the installation of some pre-configured resources. This means that if some more knowledgeable user is responsible for setting up the system then the end users only need to know how to access the service dashboard to finalize the desired configurations of the software to run in the created instances.

However an end user could want to setup any of these systems in their own local hardware and then make use of the initialized system to finalize the desired configurations of the software by themselves. This poses a problem for a Juju installation if the user is not capable of setting up the environment to be used by Juju.

With either Node-RED or InfraRED, the user simply has to have these tools installed to then start

designing and creating the desired system.

5.2 Loading Nodes

Another important aspect of these software tools is how are new elements added to the system and how are they created, since it should be discussed what level of proficiency a user has to have to be able to add new elements onto the system.

In the case of the InfraRED proof-of-concept, it is not possible to add new Node types while the Server is already active and running. This is because the Nodes are only added at runtime during Server initialization and in no other moment InfraRED rechecks the Node files, unless InfraRED is restarted, at which moment everything resets and the files are scanned again to check if any new Node types were added to the system.

Juju and Node-RED both make use of a public library, a compilation of Nodes/Charms uploaded by various users, from where the users can choose and add new elements to their system at any time. This means all the elements that have been created are not necessarily loaded while the user is creating their infrastructure but the user can see all the elements that exist and add them to their system on a need to use basis.

5.3 Node Type Availability

Firstly, InfraRED aims to tackle all types of cloud activity and all types of devices that can be used on a network. Because of that, any type of Node should be possible to create and added to the system at any time. But as already stated in Section 5.2, this is a capability that would be available in a future enhancement of the solution, as for the proof-of-concept this capability is only possible at Server initialization.

Node-RED is flow based and its focus is on simple interactions between many devices or software, and so, while there are no elements that create, manage or deploy cloud infrastructures, there are elements that are active elements able to interact with already deployed cloud infrastructures in order to obtain information or act upon their physical devices.

Juju Charms, in the other hand, only handles the configuration of cloud infrastructures or the deployment of software inside a previously instantiated cloud environment. Therefore a user cannot achieve any type of logical functionality with Juju, besides, for example, accessing already provisioned instances, and that is only possible with the assumption of prepared server images that were previously created with a functionality in mind.

InfraRED, however, is capable not just of doing both action, i.e, configure and interact with devices on

an infrastructure, but additionally instantiate the desired devices, because users can create infrastructure Nodes and functionality Nodes and then, by accessing them, achieve the same level of functionality a user would get by using both Node-RED and Juju together, i.e., Juju to configure the cloud infrastructure and then Node-RED to interact with said infrastructure. The flow-based behaviour however is not present in the proof-of-concept of InfraRED since the connections made between elements are not meant to represent a direction of logical flow behaviour but a hierarchy for deployment order.

5.4 Usability

InfraRED, Node-RED and Juju, all offer a web-based front end interface that the users interact with in order to design their infrastructures.

5.4.1 Dashboard Functionalities

The observation of the dashboards of each solution aims to provide a sense of how the software structures themselves behave, when it comes to the front facing view that the user has. For this it is important that the user is shown a simplistic and intuitive view, since *Low Code* environments should be easy and natural to use, as their main premise is to be a drag-and-drop environment with as little menus as possible.

All three dashboards (Figure 5.1, Figure 5.2 and Figure 5.3) are very similar, in that they offer the elements to compose an infrastructure design on a bar to the side and the user can add any amount of items from there to a canvas. Connections can be made between these elements on all systems and the properties of the elements can be changed by selecting the elements and making the necessary modifications.

All dashboards therefore, follow the same structure, by positioning the usable elements on a list. The prrof-of-concept of InfraRED, however, does not contain a search bar, which is considered in Section 6.2.5 as future improvement.

Node-RED presents an improvement in this section since it offers a tutorial for any first time users. This creates fast proficiency for the users and reduces any doubts about how to interact with the system or how is the software structured in terms of UI.

5.4.2 Adding Nodes to the Canvas

On all solutions the user takes Nodes from a section on the UI and places them on the Canvas, the designing space.

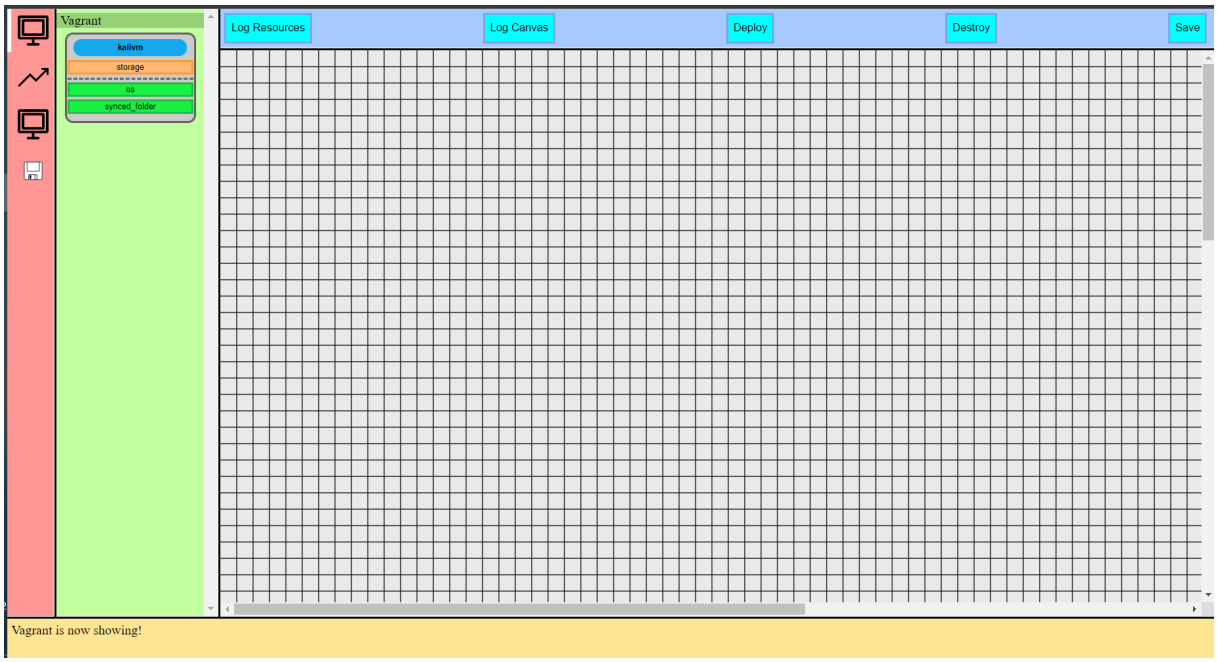


Figure 5.1: InfraRED Dashboard

Figure 5.2: Juju Dashboard. Source: [13]

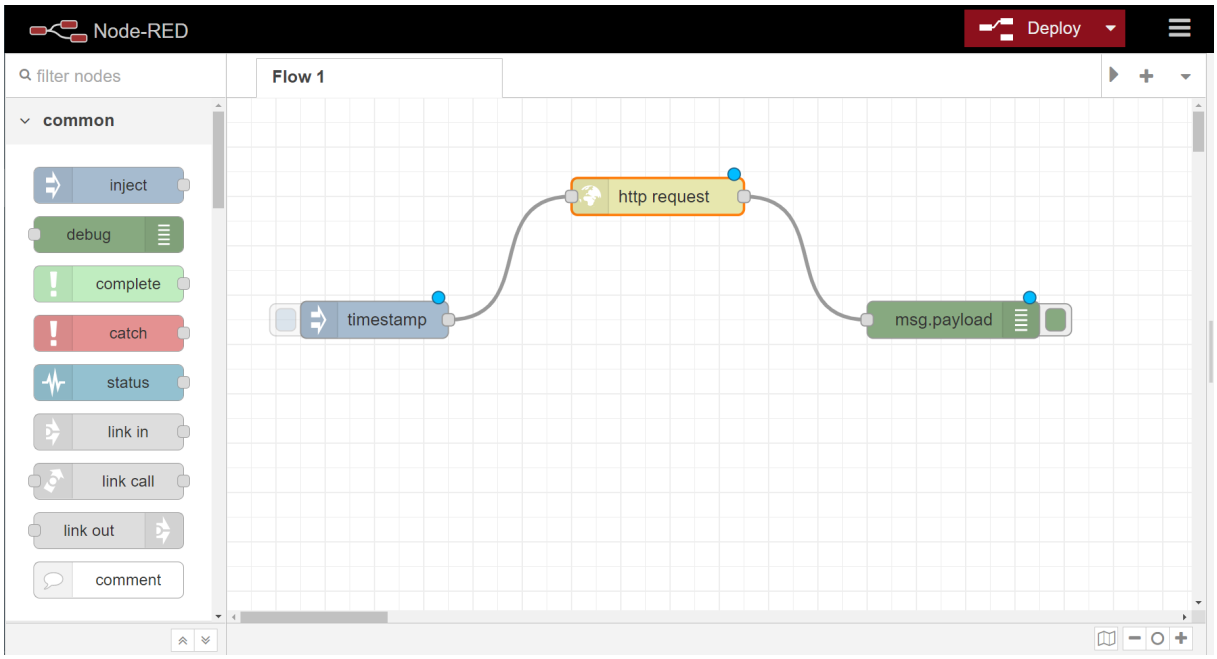


Figure 5.3: Node-RED Dashboard. Source: [10]

For InfraRED and Node-RED, the users have a drawer on the left of the UI from where they can drag Nodes into the design area, creating a copy of the grabbed Node.

In Juju, the user searches for the desired element and presses a button on a page for that element that adds it to the design area.

5.4.3 Creating a Relation between Nodes

All implementations follow the same concept of drawing a line between the *Nodes* to show that a connection between those elements is present.

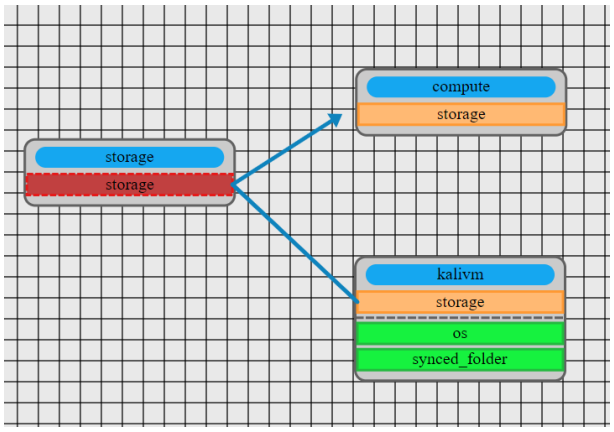


Figure 5.4: InfraRED Relation

To create this line InfraRED follows the idea of Node-RED by having a part of the element that can be clicked to start making a connection. On InfraRED, the user clicks the area of the Connectables and on Node-RED the user clicks a small square of the element.

Figure 5.4 shows how a connection is made in InfraRED. The `storage` Node has its `storage` capability selected, shown by the color red, because a relation is being drawn from it towards the `compute` Node that requires `storage`. This temporary Relationship line contains a arrow end to signal the user that it possible to connect the Connectable to another Node. A completed connection made between the `storage` Node and the `kalivm` Node is represented by a normal line (without arrow).

For Node-RED, the process and representation is similar. The elements' representation possesses small squares that serve as the connecting points between elements, as illustrated in Figure 5.5, and the line is more advanced as it curves based on the element position in the canvas, in order to not draw the line over the elements. That figure also shows a connection being drawn (line with a "+" sign).

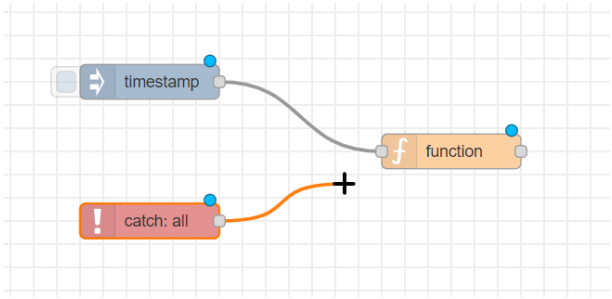


Figure 5.5: Node-RED Relation. Source: [10]

Finally for Juju, there is a small indicator at the top of the lastly moved element that allows the user to attach a connection to any other element that accepts its type, as illustrated in Figure 5.6. By clicking another element the user can then complete the connection.

Figure 5.6: Juju Charms creating a connection. Source: [13]

6

Conclusion

Contents

6.1 Limitations 57
6.2 Improvements 58

The research and development work performed in this thesis allowed to have a deeper understating about how an application could be designed to be in charge of deploying and managing cloud-based IT infrastructures. What was set up gave an overview of what systems and components need to be set in place, and so the developed InfraRED solution proof-of-concept implementation was an attempt to tackle the complexity of those challenges.

As it stands, it can be said that with InfraRED all the goals proposed in Section 1.1 were achieved. However, those goals were achieved at surface level in the proof-of-concept implementation since a full encompassing solution would require not just more time to develop, but also a multidisciplinary team of developers, in order to guarantee that it would be able to handle all the features, capabilities and the quality and quantity of deployments similar to what existing IaC tools or cloud providers service tools already do nowadays.

Another possible future would be to consider the InfraRED concept as an expansion or enhancement of an existing tool, such as Juju Charms, since its design aligns with theirs while potentially adding more types of Nodes and functionalities.

6.1 Limitations

While InfraRED's goal is to provide ease of deployment of cloud infrastructure and other devices, like the present tools, it needs a great backing of capable coders since the Nodes need to be created and managed by skilled individuals who have proficiency with a specific service and proficiency with the tool. So, while the tool is targeted at low skilled coders, to start up a library of Nodes and in turn make the tool usable, there needs to be a team of skilled coders or an incentive for other users to add Nodes to a public library, which is often tackled as an open source or community solution.

The proof-of-concept solution of InfraRED did not test the cloud capabilities of different technologies, but focused instead on implementing a foundation for what could be a solution capable of handling said cloud infrastructures or applications.

In this sense the proof-of-concept implementation of InfraRED addresses essentially the elements and the connections that compose a design, the methods of deployment and the structure of the software, as opposed to making sure that different services would be capable of communicating with each other in full, and the user being able to fully interact with the deployed system via InfraRED, as well as capable of high complexity error handling.

But with continued development it could achieve those goals, especially by tackling the improvements detailed in the following section. Those improvements can be seen as the next steps to take InfraRED to a higher level, from just a hypothetical tool to an actual capable and practical tool able to compete with similar tools, like Juju Charms and Node-RED.

6.2 Improvements

The InfraRED proof-of-concept implementation has a lot of room for expansion and improvements and the current design can be seen as a foundation for what InfraRED could possibly achieve in the future.

Some of the following ideas and options were left out of the proof-of-concept version due to time constraints that led to some functionalities being more prioritized than others and also because some can be considered out of scope for the current work or seen as commodities.

6.2.1 Nodes

Nodes can be improved because as they stand in the proof-of-concept, users must make sure they are creating the Nodes with the correct structure. However, incorrectly created Nodes are undoubtedly rejected in the system. What could be done is the creation of a base/template that any Node file would needs to comply, exposing human readable methods and parameters for setting up any type of Node.

For Nodes to be used they have to be loaded first into the Server, but on failure they are simply discarded. This can perfectly be changed to include a compliance check and a retry or a timeout method, to allow for potential failures to be corrected. This, however, requires that the loading mechanism could be able to rollback any data changes made up to the failure point.

6.2.2 Resource Bar Saved Pattern Nodes

Pattern Nodes are an important part of the ease of cloud environment creations. In the proof-of-concept solution the user can only save the design present in the canvas onto a new Node, in order to later use it and repeat, but after this fact the user can not modify that design in any way besides the inwards and outwards properties. Anything that is inside the design, the properties of Nodes that are not exterior and the relationships between Nodes, cannot be changed.

The needed functionality here is a system for saving and loading pattern Nodes. Present in the code is a database at the Server level that saves the patterns but it does not save information of the visual design, meaning that only the logical elements of a design are saved and it is not possible to load a saved pattern because visual information, such as positions and relationship lines are not saved.

6.2.3 Tutorial and Descriptions

A tutorial is a great necessity in any type of *Low Code* solution [2] because the possible actions a user can take in the system are not always clear. A guide needs to exist to lay the foundation of how InfraRED works: explain the drag-and-drop, how to change Node properties by opening the modal box and show how to create connections between Nodes.

For the user to understand what each Node does, Nodes could have a description explaining what the Node does and what each Connectable in the Node does This way, the user would not need to have any previous knowledge about the Nodes that are loaded into the Server. These descriptions should be placed inside the modal box that opens for each Node, with text or a link written by the creator of the Node.

6.2.4 Adding new Nodes

For the service to keep expanding, InfraRED must be capable of adding new Nodes into the system. It is possible to add new Nodes files to InfraRED but the Server requires a reboot to be able to add new Nodes to the runtime. What should be in place would be a functionality to allow users to submit new Node files at runtime and then proceed to request a complete update or to request this new Node to be added to the runtime library.

6.2.5 Node Search

The quantity of Nodes on a instance of InfraRED will keep growing and currently the only way to access them is by selecting the correct category and scrolling to the desired Node. To simplify this action, a search bar needs to be added to allow a user to pin point as fast as possible a Node to be used, without having to go through the categories or scrolling through the Node list of a category.

Bibliography

[1] N. Bizanis and F. A. Kuipers, “SDN and Virtualization Solutions for the Internet of Things: A Survey,” *IEEE Access*, 2016.

[2] A. Al Alamin, S. Malakar, Uddin, Gias, Afroz, Sadia, Bin Haider, Tameem, and Iqbal, Anindya, “An Empirical Study of Developer Discussions on Low-Code Software Development Challenges.” *arXiv: Software Engineering*, 2021.

[3] A. Sahay, A. Indamutsa, D. Di Ruscio, and A. Pierantonio, “Supporting the Understanding and Comparison of Low-Code Development Platforms,” in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2020, pp. 171–178.

[4] C. Fehling, F. Leymann, R. Mietzner, and W. Schupeck, “A Collection of Patterns for Cloud Types, Cloud Service Models, and Cloud-Based Application Architectures,” Institute of Architecture of Application Systems (IAAS), Daimler AG, Tech. Rep., May 2011.

[5] R. Amorim, “IaaS, PaaS e SaaS.. Qual a diferença? — Robson Amorim,” http://rsamorim.azurewebsites.net/2017/07/31/iaas-paas-e-saas-qual-a-diferenca/, Jul. 2017.

[6] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, “Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications,” *IEEE Communications Surveys Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.

[7] S. Sinha, “State of IoT 2021: Number of Connected IoT Devices Growing 9% to 12.3 Billion Globally, Cellular IoT Now Surpassing 2 Billion,” https://iot-analytics.com/number-connected-iot-devices/, Sep. 2021.

[8] Y. Li, X. Su, J. Riekki, T. Kanter, and R. Rahmani, “A SDN-based Architecture for Horizontal Internet of Things Services,” in *2016 IEEE International Conference on Communications (ICC)*, 2016, pp. 1–7.

[9] L. Pang, C. Yang, D. Chen, Y. Song, and M. Guizani, “A Survey on Intent-Driven Networks,” *IEEE Access*, 2020.

[10] “Node-RED,” https://nodered.org/.

[11] OASIS, “TOSCA Simple Profile in YAML Version 1.3,” OASIS Standard, 2020.

[12] Canonical, “Metal as a Service,” https://maas.io/, 2022.

[13] “JAAS - Juju as a Service — Juju,” https://jaas.ai/.

[14] R. Arora, N. Ghosh, and T. Mondal, “Sagitec Software Studio (S3) - A Low Code Application Development Platform,” in *2020 International Conference on Industry 4.0 Technology (I4Tech)*, 2020, pp. 13–17.

[15] A. Moreira, “Fixenet/infraRED,” https://github.com/Fixenet/infraRED, 2022.