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

André Miguel Jesus Moura da Silva Moreira

Abstract—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. An InfraRED solution will make possible for programmers and non programmers to provide system topologies for their businesses since the complexities of network and systems design will be abstracted visually. 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.

Index Terms—Low Code; Cloud Computing; Infrastructure as Code (IaC); Development and Operations (DevOps).

1 Introduction

The objective of InfraRED is to allow many people of different backgrounds to deploy and manage their own infrastructure with ease. There's a necessity for easy deployment of cloud resources and *Low Code* is a way to have everyone capable of doing so.

To do this we are combining complex technologies with easy front facing ones, the abstraction of complex tasks will allow anyone to do what before could take them a lot of time, both learning and configuring, but with InfraRED all of that is done before for the commodity of the user and they simply have a non complex way of drawing their designs.

1.1 Contextualization

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.

To combat this, 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, added to the fact that most network, systems, services and functions are being virtualized instead of existing as hardware on premises. 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 [1].

The user will work with abstracted versions of various common network, compute and software/application elements. InfraRED is tasked with presenting the user with the necessary virtual Customer-Premises Equipment (CPE) 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.

1.2 How to solve the Problem

Based on the contextualization it is 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.

Having these goals fulfilled InfraRED will be capable of sustaining itself, nodes are created by users that are more tech savvy and are then added to InfraRED. Other users, with varying technological capabilities, then have access to a library of nodes to use and construct their desired design which should be as simple as possible.

1.3 Background

The development of the InfraRED solution took inspiration from some previous works, that guided its devel-

• *André Miguel Jesus Moura da Silva Moreira, nr. 87630, E-mail: andre.moreira@tecnico.ulisboa.pt, Instituto Superior Técnico, Universidade de Lisboa.*

Manuscript received October 31, 201822

opment method. Exploring those academic work about cloud, *Low Code* and Internet of Things (IoT), allowed InfraRED's design choices to evolve in a concise way.

1.3.1 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 [2].

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 [1].

1.3.2 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 [3]. 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.

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.

1.3.3 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 [4].

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 [4].

1.3.4 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 [5].

1.4 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.

1.4.1 Node-RED

Node-RED [6] 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.

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.

1.4.2 TOSCA

The Topology and Orchestration Specification for Cloud Applications (TOSCA) [7] 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 [7] is called a service template (as in Figure 1) that includes a

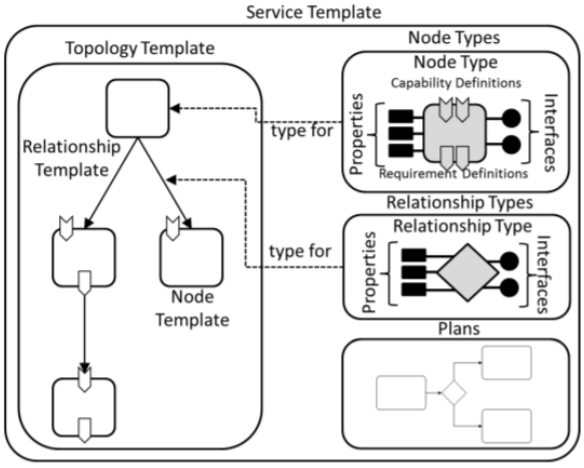

Figure 1. TOSCA Service Template Example. Source: [7]

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, 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..

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

A few companies offer solutions for creating cloud services (introduced in Section 1.3.2) 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).

1.4.4 Metal as a Service (MaaS)

Metal as a Service (MaaS) [8] 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.

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

MaaS is also often used in conjunction with Juju, a cloud foundation solution, as described in the following Section 1.4.5.

1.4.5 Juju Charms

Juju Charms or **JaaS!** (**JaaS!**) [9] 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.

1.4.6 Sagitech Software Studio (S3)

Arora et al. [10] 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.

2 Design

To design InfraRED, inspiration was taken from section 1.4.2 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.

2.1 Architecture

InfraRED's structure is a combination of three sections, the representation of infrastructure which we define as Nodes, the Server side which handles the deployment and the Client side which the user interacts with to create and design.

The Server is the brains of the operation, this is the piece that controls all functionality of InfraRED and holds 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 deploy them.

2.1.1 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 Node Pattern explained in section 2.2.1.

To have Nodes available at the Server there will be files that it will load and boot up, 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 Node Patterns so that it is 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.

2.1.2 Client

The Client side is structured to have a User Interface (UI) handling side and a functionality side. The UI side has the layout shown in Figure 2, the design canvas of InfraRED is the only frontal piece of software that the user will interact with. The current layout was chosen to be made similar to how 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.

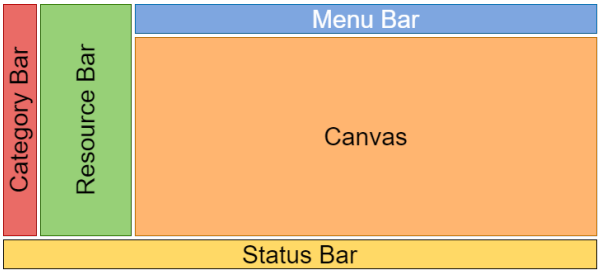

Figure 2. Canvas Layout

In Figure 2 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, to compose the desired topology by making use of these Nodes that came from the green area.

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

Upon clicking on a Node, introduced in section 2.2.1, that is present on the canvas 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.

2.2 Infrastructures

With InfraRED the aim is to build infrastructure on top of a cloud or local environment. To do so the system gives the user access to many Nodes which are a representation of infrastructure, services or pieces of software to be deployed on said environments.

2.2.1 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 1, 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 be structured to have their type shown at the top so it is possible to distinguish different Nodes and will expose the Connectables under the type, with Capabilities and Requirements having different colors, as shown by Figure 3.

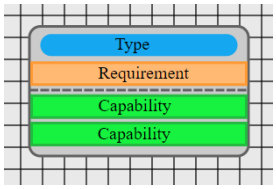

Figure 3. Node Structure

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 functionality. Nodes have their own properties that can be altered during design creation to change Node specific properties that affect and alter their behaviour during and after deployment. To exemplify, there could exist a computation Node with a property for the number of Central Processing Unit (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 2.1.1, as files that get loaded when the server starts and then made available to the user via the Client on the layout location for Nodes to be placed at which will also be organized by categories.

Additionally during user interaction, Nodes can be combined together for the sake of practicality as Node Patterns, this removes the need to individually set up groups of Nodes that are repeating and only needing to manage a singular Node that encompasses multiple Nodes and their Relationships, making it simple to create duplicates of complex structures.

2.2.2 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 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 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 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.

2.2.3 Relationships

Relationships are what connects Nodes together, these do not hold or transpose any data from one Node to another. These 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 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 it to understand the order of deployment for the Nodes in a design, explained in detail in section 4.

3 Implementation

This chapter will go through how the design is implemented and what algorithms are in place, making reference to what technologies are being used.

3.1 Technology Stack

The entirety of InfraRED 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. ExpressJS[3] will be used to serve InfraRED's entry point and handle requests coming from the client, since there is not a need for a complex server side.

3.1.1 Server

The Server exposes a REST API, this way 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.

3.1.2 Client

The Client side implements a standard HyperText Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript combination to create and display a web page to the user that allows them to interact with InfraRED with buttons and drag-and-drop.

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

3.2 UI of the Client Side

This section will go over how each individual piece of the UI functions and what layout choices were made to achieve a tool that is easy to use.

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 made to correctly represent the design in the Canvas. 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 rest of the layout is 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.

1. https://www.javascript.com/
2. https://nodejs.org/
3. https://expressjs.com/
4. https://jquery.com/
5. https://svgjs.dev/docs/3.0/

A zoomed in example of the layout that the user interacts with can be seen on Figure 4, this layout follows the structure that was proposed at section 2.1.2. On the sections below, the implementations of the various layout parts will be discussed.

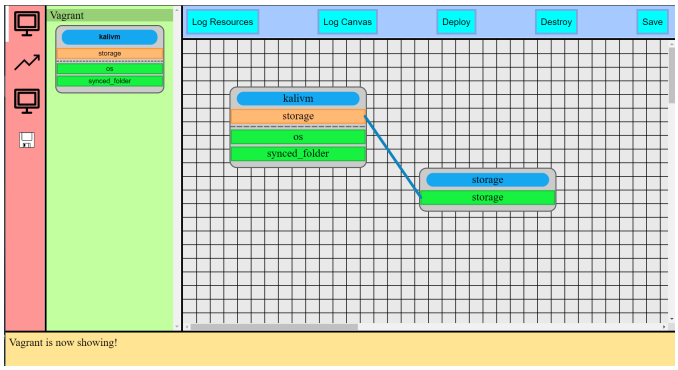

Figure 4. InfraRED Dashboard

3.2.1 Node

A Node is constructed to show its type at the top or a custom name if the user sets up one via the modal box explained in section 3.2.7. Then the Connectables are displayed, with Capabilities having a green color and the Requirements having a orange color. These Connectables also display their type name, as proposed in Figure 3

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.

To move the Nodes, both from the Resource Bar to the Canvas and to move them in the Canvas the user simply has to hold the left click button and the cursor will change from a selecting cursor to a grabbing cursor and the border of the Node change from grey to blue to signify the Node is being held, then the user just has to lift the left click and the Node will stop moving, if the Node is coming from the Resource Bar then the user must drop the Node in the Canvas or else 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* tool.

3.2.2 Canvas

To draw the Canvas the choice was made to give the user a spacious area that they could use, which is achieved in InfraRED through the use of a scroll able environment. The implemented size of a 2000 by 2000 pixels Canvas area was chosen arbitrarily and it can be easily changed but the value is similar to what a tool like Node-RED [6] uses.

3.2.3 Status Bar

The Status Bar is capable of showing the user text messages in order to do so. Current implementation of the Status Bar has minimal uses, being capable of reacting to system changes like deployments and Pattern Node saves or displaying text information about the current Nodes in the Canvas.

3.2.4 Category Bar

The Category Bar is built during the loading of Nodes, and from there InfraRED is able to extract two pieces of information: the category name and its icon. It then builds a icon based selector for the user then the user can click on any other category icon to then view Nodes from that category and stop seeing Nodes from the previous one.

After loading every category, InfraRED will create a special category exclusively for Pattern Nodes, the last category shown at Figure 5. From the figure, the *Vagrant* category can be seen selected and the respective Resource Bar group is being shown.

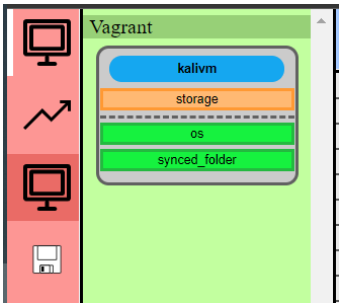

Figure 5. Categories Example

3.2.5 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 here 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 will have a scroll bar for the user to view all 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 it is not limited by number.

3.2.6 Menu Bar

The Menu Bar is the layout location that contains buttons that allow the user to interact with the design they are making.

Currently there are three buttons for functionality that the menu is responsible for, these are saving, deploying and destroying.

All the buttons in the Menu Bar emit an event, section 3.3.3, that is 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.

3.2.7 Node Modal

By double clicking on the Node, a modal box opens 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. 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 is explored in detail in section 3.3.2.

3.2.8 Relationship Lines

A Relationship line is a blue line between Connectables connected by the user. To create a connection the user clicks any connectable that is in the canvas, that connectable then turns into a red color that signifies the selection and an arrow is drawn from there. To complete the connection the user clicks another connectable that is of the same type and is the opposing mode, that is a Capability can only connect to a Requirement and vice versa.

3.3 Logic at the Client Side

Logic wise, InfraRED's Client side has files to handle interaction 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.

3.3.1 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 gets sent to the Server for deployment and for saving a Pattern Node.

3.3.2 Node Database Memory

The Client holds in memory two lists that contain information about the Nodes that the user can interact with, the first list in memory is 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 other list is the Node Canvas list that stores the Nodes that get placed into the Canvas.

3.3.3 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. With this module it is possible to append a JavaScript function to a certain `event_name`, afterwards it is possible to invoke these functions by emitting said `event_name` which will make a call to all the functions under that name with the arguments given after.

3.3.4 Loading the Nodes

The **loader** module at the Client is tasked with making a GET call to the `/listNodes` endpoint, 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.

3.3.5 Initiating the Deployment

The **deployer** module at the Client handles making a POST call to the `/deploy` endpoint 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.

The 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 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.

3.4 Logic at the Server Side

After receiving the API calls from the Client, the Server side starts its functionalities, which are deploying a topology of Nodes, saving Pattern Nodes and destroying a topology. These calls function asynchronously, explained at section 3.4.4, to allow for the actions to be more efficient, allowing us to remove the need to wait for Nodes' actions individually and sequentially, each Node can activate its processes simultaneously as long as there are no requirements for that deployment.

3.4.1 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 require.

3.4.2 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. The entry point for a deployment action is constituted by four steps.

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

After verifying that the previous deployment was cleaned the Server starts ordering the Nodes that the user wishes to deploy, which is achieved by verifying the Relationships between these Nodes and creates each level as a map list. The max level of the deployment list is also established, which aids in the cleanup process. With the level list built, the Server starts creating instances of each Node.

With all the instances positioned at the appropriate levels, 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 iterates through this process until all levels are deployed.

3.4.3 Deploying Pattern Nodes

While creating the Node instances and a Pattern Node is detected, its internal memory is parsed, which will contain data about the Nodes and Relationships that it has inside. The process then recursively parses from that data calling the `deploy()` based on the level lists inside the Pattern Node.

Note that the created design can 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 with this implementation keeps these two situations equal.

3.4.4 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 so it is possible for each Node to launch these processes without having to sequentially wait for each other. We do this by using the `allSettled()` method from the Promises library which awaits the completion of all tasks inside the list provided as an argument. 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 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. At the end of the algorithm a set of lists in the format of Figure 6 is built.

4.1 Deployment Example

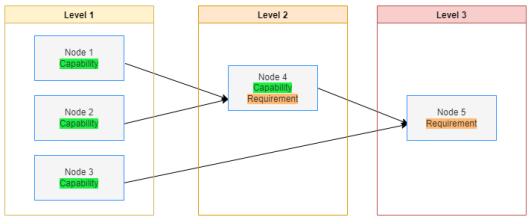

Figure 6. Deployment Sequence per Level

4.2 System Initialization

To begin the initialization process, the Server searches for all the available Node files, by traversing a directory that contains all Nodes, then validates the composition of the Node file to see if it is created correctly and subsequently invokes the `load()` method of each one. All the successful loads get put into a runtime variable and the rejected ones get discarded.

The process of Node loading has another important piece of functionality which is the process of creating the front end representation of a Node.

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.

4.3 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 7, the place where the user is designing their topology.

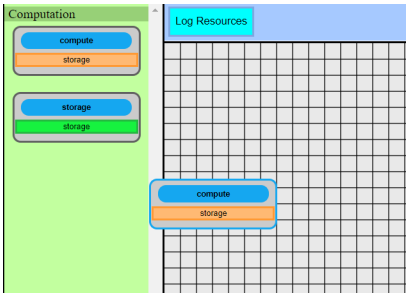

Figure 7. Adding a Node to the Canvas

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 8).

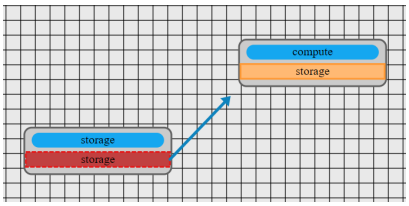

Figure 8. Establishing Relationships between Nodes

Interact with Node Properties: By double clicking on a Node, the user is met with a modal box (Figure 9) that exposes the Nodes' properties and shows additional information about the Node and contains buttons to interact with it (Figure 10).
Delete Nodes: On the modal box of each Node there is a delete button, illustrated by Figure 11, 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 12).

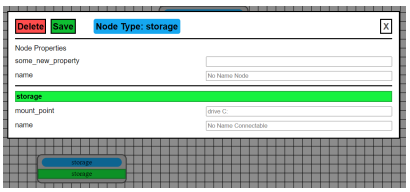

Figure 9. Interacting with Node Properties

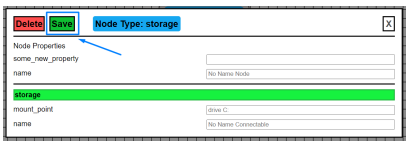

Figure 10. Saving Node properties

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 13).
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 14).

5 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 [11].

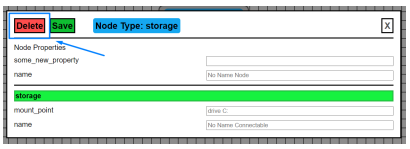

Figure 11. Deleting a Node from the Canvas

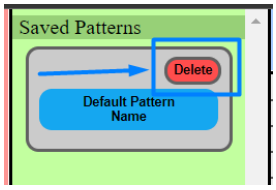

Figure 12. Deleting a Pattern Node

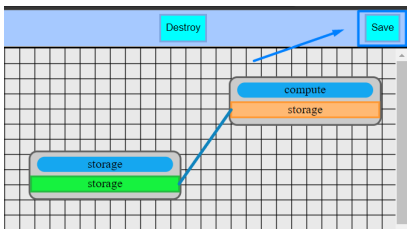

Figure 13. Saving a Pattern Node

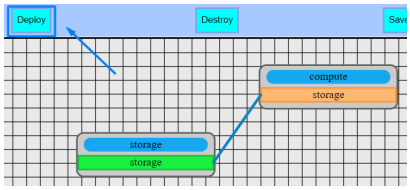

Figure 14. Deploying a design

6 Evaluation

There are many similarities between InfraRED, Juju Charms [9] and Node-RED [6] so the three systems will be evaluated to see how they compare to each other. The evaluation will not have a performance focus since the goal for InfraRED is to propose a system capable of combining *Low Code* and the TOSCA standard interpretation of cloud infrastructure and orchestration.

For InfraRED and Node-RED there is no user setup requirement after installation, upon boot up the user is presented with a fully functioning system that is ready to use at any moment. However, Juju requires that the user sets up an environment to deploy the system via the command line, either by adding a cloud provider or a MaaS environment.

As a complete end user, on all systems you can simply be redirected to the web page that controls the system and that would require no installation or set up from the user, this means that some other user is responsible for setting up the main system.

All three dashboards 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 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.

7 Conclusion

Current InfraRED allowed us to understand more about how an application could be in charge of deploying cloud infrastructure and what was set up gave an overview of what systems need to be in place and our implementation tried to show how to tackle those challenges.

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. 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.

As it stands, it can be said that InfraRED achieved all its goals, however only at surface level since InfraRED as it is can not be considered a full encompassing solution. InfraRED hopes to be an expansion on the currently present Juju Charms since our design aligns with theirs while potentially adding more types of Nodes and functionality.

Acknowledgments

I would like to thank my girlfriend for motivating me through this work and my mom for her dedication to my success. I would also like to acknowledge my dissertation supervisors Prof. Rui Cruz and Prof. José Delgado for their insight, support and sharing of knowledge.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] "Node-RED," https://nodered.org/.

[7] OASIS, "TOSCA Simple Profile in YAML Version 1.3," OASIS Standard, 2020.

[8] Canonical, "Metal as a Service," https://maas.io/, 2022.

[9] "JAAS - Juju as a Service — Juju," https://jaas.ai/.

[10] 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.

[11] A. Moreira, "Fixenet/infraRED," https://github.com/Fixenet/infraRED, 2022.