Easing Construction of Smart Agriculture Applications Using Low Code Development Tools

Isaac Nyabisa Oteyo[1,2(✉)], Angel Luis Scull Pupo[1], Jesse Zaman[1], Stephen Kimani[2], Wolfgang De Meuter[1], and Elisa Gonzalez Boix[1]

[1] Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium
{isaac.nyabisa.oteyo,angel.luis.scull.pupo,jesse.zaman,
wolfgang.de.meuter,elisa.gonzalez.boix}@vub.be
[2] School of Computing and Information Technology, Jomo Kenyatta University of Agriculture and Technology, Nairobi, Kenya
skimani@scit.jkuat.ac.ke

Abstract. Smart agriculture applications are a promising path to the future of modern farming. Building smart agriculture applications is a complex undertaking that requires considering different factors, such as the technology that can be used to implement the applications. These factors require advanced skills in software construction, such as handling the distributed setting for smart agriculture applications. As such, implementing smart agriculture applications requires engaging experienced developers with the skills to tackle the issues mentioned above. Low code development tools have risen that domain experts (e.g., agricultural extension workers that give advice to farmers) outside software engineering can use to construct software applications. The low code development tools provide visual programming environments that developers can use intuitively to construct applications. However, the existing low code development tools do not offer support for low infrastructure networking that sensors can use to communicate directly to mobile devices (e.g., smartphones and tablets), computation at the edge, and offline accessibility capabilities at the edge that are crucial for smart agriculture applications. In this paper, we present DisCoPar-K, a low code development tool that supports the properties mentioned above for implementing smart agriculture applications. We show how DisCoPar-K can improve the development of smart agriculture applications by implementing smart agriculture use cases on it.

Keywords: mobile applications · visual programming · smart agriculture · Internet of Things · cloud computing · edge computing

1 Introduction

Smart agriculture (SA) is a modern farming approach that is increasingly being exploited to improve processes, such as monitoring environmental conditions [5]. As a discipline, SA encompasses a set of technologies such as the Internet of Things, cloud computing,

S. Longfei and P. Bodhi (Eds.): MobiQuitous 2022, LNICST 492, pp. 21–43, 2023.
https://doi.org/10.1007/978-3-031-34776-4_2

and mobile applications that are integrated into smart agriculture applications (SAAs). Collectively, the SAAs can be applied to sensing environmental conditions, such as soil moisture and temperature [30]. The soil moisture and temperature conditions are critical parameters in the growth and development of crops right from the seeding and sprouting stage to maturity [35]. In fact, the sprouting of seeds determines the population of crops that reach maturity and, hence, the overall yield. As such, for agricultural extension workers and farmers to achieve optimal yields, it is important to keep track of the soil moisture and temperature conditions at the seeding and sprouting stage. By definition, agricultural extension workers advise and assist farmers to implement creative technologies geared toward improving yields [20]. As mentioned before, keeping track of soil moisture and temperature conditions can be done using sensors and SAAs. In this case, the sensors can be programmed to collect data and send it to the farmer's mobile phone in a timely fashion for decision-making. However, farm restrictions such as the physical space that must be covered to track the soil moisture and temperature conditions impose a constraint that forces the SAAs to be designed and implemented in a distributed setting. Designing and implementing SAAs in a distributed setting requires technological factors to be considered such as handling distribution and micro-controller programming. The technological factors require a combination of skills to implement the different parts of SAAs, e.g., distributed programming skills are required to handle the communication between the different components [6]. The SAAs rely on communication networks for the different parts to communicate with each other, something that can fail to happen when the networks become unavailable [19]. In such cases, data coming from the sensors can be lost, and this needs to be handled in the implementation. Lastly, micro-controller programming skills are required to specify how the application can receive data from sensors or process data near the source using sensors. This means that implementing all the different parts of SAAs can take considerable time.

Low code development tools (LCDTs) have risen as an alternative that domain experts outside software engineering (such as agricultural extension workers) can use to implement software applications [6, 22]. By definition, LCDTs are visual programming environments (VPEs) in which applications are constructed by dragging, dropping, and connecting visual components that represent different tasks in the application [22, 29]. To explain it in context, let's consider constructing *DiscoSense*, a sensing application for soil moisture and temperature conditions using LCDTs. The *DiscoSense* application requires connecting a soil moisture sensing component, a temperature sensing component, and a component to receive and display the data on the mobile phone. In addition, *DiscoSense* requires components that can process the data on the sensor (edge) and only send the aggregate values to the mobile phone. To avoid losing data when the network becomes unavailable, the application requires storing data on the sensor. The unavailable network can also affect the communication between the application and the server. As such, *DiscoSense* requires that the sensors collecting soil moisture and temperature data communicate directly to the mobile phone without going through a centralised server. The existing LCDTs partially support the above issues, such as environment sensing capabilities. In addition, the existing LCDTs are provided on the cloud through Platform-as-a-Service [22]. This can hinder the functioning of SAAs in areas that experience poor network coverage. To the best of our knowledge, none of the existing LCDTs

offers support for: 1) low infrastructure networking for sensors to communicate directly to SAAs running on mobile devices, 2) components to support computation at the edge to process data near the source, and 3) components to support offline accessibility at the edge to store data on sensors when the network connection becomes unavailable. As such, none of the existing LCDTs can be used to implement *DiscoSense* or similar SAAs. For instance, DisCoPar [31–33], the LCDT that we consider in this work lacks components that can execute at the edge and directly communicate with applications running on smartphones or tablets without going through a centralised server.

In this paper, we extend DisCoPar [31–33] with an additional execution point to host components that can run on the edge. We implement components for environment sensing and performing computations at the edge that are domiciled in the new scope. We further advance the policies for offline accessibility on mobile devices to limit connecting a chain of successive offline accessibility components. In addition, we implement components for offline accessibility at the edge. In our implementations, we ensure that the components executing at the edge can communicate directly to those on mobile devices without going through a centralised server. This results in DisCoPar-Kilimo (DisCoPar-K), a low code development tool that we present in this paper. For validation, we show how DisCoPar-K can improve the development of SAAs by implementing smart agriculture use cases on it. From a theoretical perspective, we identify different properties that LCDTs should have to support implementing SAAs. We use the identified properties to compare different LCDTs that exist in the literature and motivate our work. To the best of our knowledge, our contribution is unique in the context of LCDTs. The rest of the paper is organised as follows. Section 2 presents the motivation and background. Section 3 describes DisCoPa-K together with the extensions added to support implementing SAAs. Section 4 describes the validation and discussions of the implemented use cases. Lastly, Sect. 5 presents the conclusions and gives directions for future work.

2 Motivation and Background

To motivate our approach, consider the scenario of corn seeding and sprouting, where the farmer's goal is to obtain the *maximum yield* from seeds planted. This is a representative scenario for SAAs derived from El-Sanatawy et al. [11], Sudozai et al. [26], and our interactions with extension workers and farmers in Kenya. To achieve the *maximum yield* goal, the crop must develop 'well' during all the 'cultivation' phases, i.e., planting, sprouting, developing, and maturity. Specifically, during the stage that encompasses seeding and sprouting of corn, maximum yield is ensured whenever 'optimal' sprouting is achieved. Optimal sprouting means that from the number of seeds planted, there is a threshold on the number of plants which does not affect the expected yield. To achieve such a threshold, farmers have to keep track of vital environmental conditions such as soil moisture and temperature that influence the sprouting of corn seeds [35]. The soil moisture gives an indication of the available water content to support the sprouting of corn seeds, i.e., inadequate soil moisture and extremely low or high temperatures negatively affect seed sprouting. As such, these conditions cannot be monitored in isolation.

Researchers have invented metrics that can give a general overview for soil moisture and temperature conditions, such as the *soil heat capacity* metric [35]. The soil heat

capacity metric is important in explaining the interaction between soil moisture and temperature. As such, this metric can help farmers to react and take appropriate actions to ensure the optimal sprouting of corn seeds. Currently, the metric is measured using field observation approaches, such as calorimetric and force-restore methods [13]. The current practices can be improved by implementing the corn seeding and sprouting scenario as a smart agriculture application using LCDTs, since extension workers do not have a strong background in software programming. However, implementing the scenario requires LCDTs to support the following properties.

Environment sensing. The property is important to help in monitoring the prevailing soil moisture and temperature conditions. In the context of LCDTs, this property refers to whether LCDTs have in-built environment sensing capabilities for soil moisture and temperature conditions. In our scenario, the soil moisture and temperature conditions that are vital for corn seeding and sprouting cannot be read in isolation.
Computation at the edge. *Edge computation* is necessary to transform the data collected near the source into meaningful information. In our scenario, the computation that combines the soil moisture and temperature into the *soil heat capacity* metric can be performed at the edge. To minimise the number of requests sent to the server over communication networks, other computations, such as the average soil moisture and temperature, can be done at the edge before the aggregate values are sent to the mobile device.
Offline accessibility at the edge. In our scenario, we assume that the application for sensing the soil moisture and temperature conditions is programmed by the domain expert (agricultural extension worker) and used by the farmer. As mentioned before, the network connection may become unavailable and this can make the different parts of SAAs not communicate with each other. The 'availability' of the soil moisture and temperature data is key to making correct decisions about corn seeding and sprouting. Therefore, the microcontroller that is hosting sensors must be capable of *keeping all the data* that the sensors generate whenever the network becomes unavailable. It is assumed that farmers that are using the sensing application visit the farm at least once every day.
Low infrastructure networking. The *low infrastructure networking* property refers to whether different parts of a smart agriculture application constructed using LCDTs can communicate with each other without going through a centralised server as illustrated in Fig. 1. Relying on centralised cloudhosted servers can be a challenge in remote farms that experience unreliable internet connections [19].

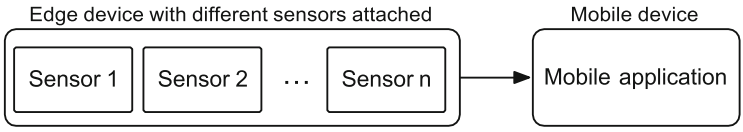

Fig. 1. Edge device featuring sensors communicating to a mobile application without a centralised cloud-hosted server.

In the next section, we use the above properties to perform a state-of-the-art (SOTA) analysis for LCDTs.

2.1 SOTA of LCDTs for Building SAAs

Table 1 shows a summary of the SOTA for different LCDTs. The LCDTs that were included in this summary were based on support for a web-based VPE. Secondly, we included LCDTs that are targeted for environmental sensing. All surveyed LCDTs provide support for web-based VPEs and environment sensing capabilities. However, in some tools like Node-RED, the sensing capability needs to be constructed into the target smart agriculture application. DisCoPar supports offline accessibility on mobile devices using in-database storage. None of the surveyed LCDTs supports low infrastructure networking, computation at the edge, and offline accessibility at the edge.

Table 1. Summary comparison of different low code development tools.

Tool	We-based VPE	Sensing capabilities	Offline accessibility	Edge computation	Low infrastructure networking
Mendix [15, 17]	✓	✓	✗	✗	✗
Blynk [12]	✓	✓	✗	✗	✗
AtmosphericIoT [3]	✓	✓	✗	✗	✗
Zenodys [34]	✓	✓	✗	✗	✗
Axonize [4]	✓	✓	✗	✗	✗
FRED [9]	✓	✓	✗	✗	✗
Node-RED [1]	✓	✓	✗	✗	✗
Simplifier [25]	✓	✓	✗	✗	✗
Salesforce [11, 23]	✓	✓	✗	✗	✗
D-NR [8]	✓	✓	✗	✗	✗
uFlow [27]	✓	✓	✗	✗	✗
DDFlow [18]	✓	✓	✗	✗	✗
WotKit Processor [7]	✓	✓	✗	✗	✗
glue.things [16]	✓	✓	✗	✗	✗
DisCoPar [31–33]	✓	✓	✓	✗	✗

From the LCDTs presented in Table 1, in DisCoPar: 1) computation tasks in applications are highly conceptualised into visual components for novice developers, and 2) the tool supports implementing web and mobile applications using drag-and-drop and point-and-click graphical user interface techniques. These techniques can be intuitive and useful to domain experts for constructing SAAs. Therefore, in this paper, we consider DisCoPar and extend it with components to support the properties that we previously identified for developing SAAs.

2.2 DisCoPar

DisCoPar embraces a flow-based programming approach in which an application is built out of different visual components, each representing a computation task [31–33]. The application is represented as a directed acyclic graph of n connected components. The example in Fig. 2 shows an application composed of three connected components (C1, C2, and C3) in one linear graph. Components execute on receiving data from external sources or upstream components. The computation results are sent out on the output ports (e.g., $C_1(out)$ and $C_2(out)$ in Fig. 2) to the downstream components. Upstream components precede the reference component backwards, while downstream components succeed the reference component forward. In Fig. 2, C1 is an upstream component to C2, while C3 is a downstream component to C2. Components can have zero or multiple input and output ports. In Fig. 2, components C1 and C2 have one output port each; C2 and C3 have one input port each. In terms of the task performed, the components in Fig. 2 can be classified into either *source* components (e.g., C1), *processor* components (e.g., C2) or *sink* (e.g., C3) components. As the name suggests, the *source* components generate data, while the *processor* components perform some processing and transformation of data flowing through the application graph. Some of the *sink* components can be used as viewing monitors to display data flowing through the application and build graphical user interfaces for applications.

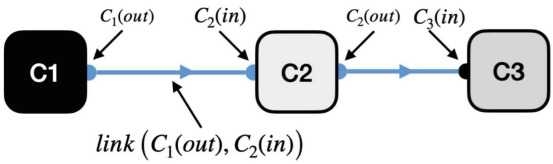

Fig. 2. Application flow graph featuring component ports and links.

Components are linked together via connections or arcs. DisCoPar's VPE only allows end-users to perform correct connections among data types on different components. In this regard, the composition mechanism is based on the supported data type, denoted by a distinct port colour. Therefore, given two components A and B, we can define $P_{input}(B)$ and $P_{output}(A)$ as the sets of input and output ports on components B and A, respectively. Outport $j \in P_{output}(A)$ and input port $k \in P_{input}(B)$ are compatible if they support the same data type such that they have the same port colour e.g., $c_1(out)$ and $c_2(in)$ in Fig. 2 or at least $k \in P_{input}(B)$ is coloured black e.g., $c_3(in)$ in Fig. 2, to accept any data. This implies that components A and B are compatible if $\exists j \in P_{output}(A)$ and $\exists k \in P_{input}(B)$ such that j and k are compatible. Component links have a colour coding that is inherited from the colour coding in the output port of the source component. In Fig. 2, $link(c_1(out), c_2(in))$ inherits the colour coding for port $c_1(out)$ in C1. Application designers can create connections between components in different scopes the same way they create connections between components in the same scope. Connections made between components

residing in different devices are automatically handled by DisCoPar. For communication between components and graphs in different devices, DisCoPar uses web sockets (Socket.IO[1]) and RxJs[2].

Component Categories: DisCoPar applications run on mobile phones and communicate to a server backend or a web-based dashboard on the server. Depending on where the execution happens, components can be classified into three categories/scopes: mobile, server, and web components. The mobile scope contains components that execute on mobile devices (e.g., smartphones and tablets). The server scope contains components that execute on the server side for data processing. Lastly, the web scope contains components for web-based analysis and visualisation. Such components are used to build the dashboard for the server side. Some components can span multiple scopes, e.g., mobile and web scopes. Each scope houses respective components on the component menu represented as component categories, i.e., mobile, server, and dashboard.

3 DisCoPar-K

DisCoPar-K is an extension of DisCoPar to support environment sensing capabilities, computation at the edge, offline accessibility at the edge, and low infrastructure networking for SAAs. In this section, we describe the architectural overview and components that comprise DisCoPar-K.

Architectural Overview: Figure 3 shows the architecture of DisCoPar-K. The architecture consists of four different parts labelled 1, 2, 3, and 4 in Fig. 3, i.e., sensors at the edge, a mobile application, a server with a database, and a web dashboard for visualisation.

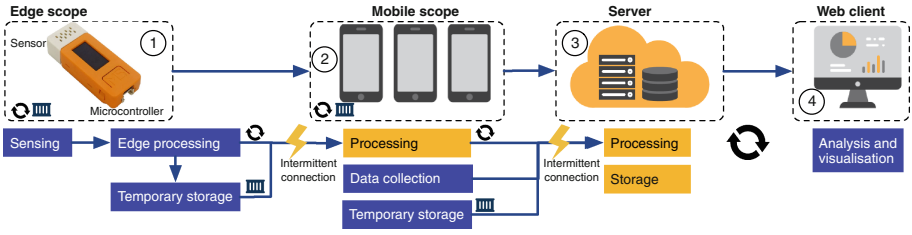

Fig. 3. Architectural overview of DisCoPar-K.

The work presented in this paper focuses on the parts labelled 1 (the edge scope) and 2 (the mobile scope) in Fig. 3. The edge scope hosts components that can execute on microcontrollers that have sensors attached to them. The sensors are used to gather data such as soil moisture and air temperature. The data collected is initially processed at the edge before being sent directly to mobile devices for further processing. The lightning strikes in Fig. 3 indicate intermittent network connections. When the network becomes

[1] https://socket.io/.
[2] https://rxjs.dev/.

unavailable, the collected data can be stored temporarily at the edge for transmission to mobile devices when the network becomes available. From mobile devices, the data can be sent to the server for storage and visualisation on a web dashboard. On mobile devices, the data can also be stored when the network becomes unavailable for transmission to the server when the network becomes available.

DisCoPar-K extensions: The extensions that comprise DisCoPar-K fall into the categories described below.

Visual Programming Environment: Components from the mobile, server and web scopes could not fully support implementing SAAs. As such, we extended DisCoPar with the edge scope as shown in the component menu in Fig. 4 (highlighted in yellow colour) to house components that support: 1) environment sensing, 2) computation at the edge and 3) offline accesibility at the edge. On the VPE, these components are distinguished by the blue colour. For example, in Fig. 4, the *ReadSoilMoisture* component is coloured blue and executes at the edge (i.e., sensors), while the *UnWrap* component is coloured black and executes on the mobile phone. Figure 4 shows the visual programming environment for DisCoPar-K with the added edge scope components.

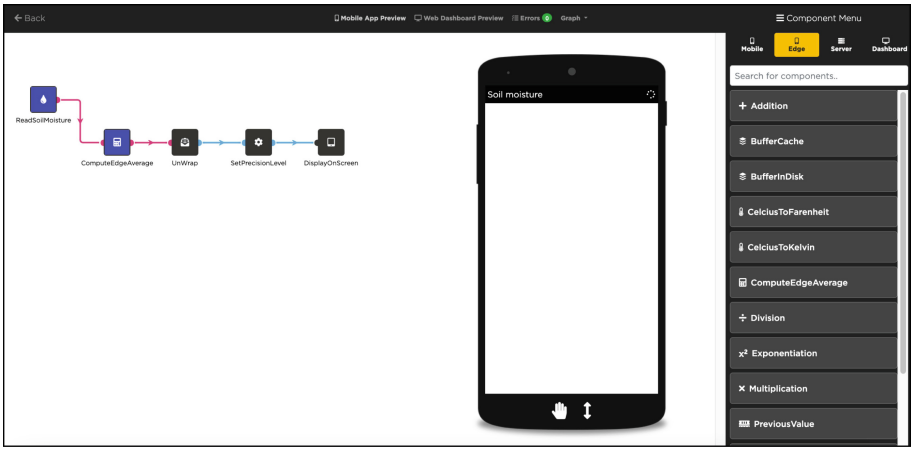

Fig. 4. DisCoPar-K's VPE showing the component menu.

On top of the existing components in DisCoPar, we added more components to meet different application requirements (goals) in the context of SA. Some of the requirements that were implemented into components include:

– *Keeping track on the number of connected devices:-* To accomplish this goal, we added the *ConnectedDevices* and *DataArrayToTable* components that function as follows. The *ConnectedDevices* component subscribes to an event to receive messages that contain the identity of each connected device. The received message is processed to remove duplicate entries, leaving only unique connections using the *DataArrayToTable* component. The processed message is then converted into a dataset for display as a list using the *DisplayAsList* component. Each unique connection is counted

to determine the number of connected devices that are displayed on the screen using the *DisplayOnScreen* component. In the *DataArrayToTable* component, each received message is stored in an array, and the array is then converted into a set to remove duplicates and return a new array with unique entries. Each element in the new array is converted into observation to meet the design requirements of DisCoPar-K. Each observation is then added to a global counter that keeps the total number of connected devices.

- *Accumulating data from different devices:-* To accomplish this goal, we added the *DeviceAccumulator* component that can accumulate data from several sensor devices. The component receives input data and stores it on a map using the device identifier as the key and the payload as the value. As a configuration setting, the component takes the number of connected devices from which to accumulate data. On its output port, the component sends only the values of data stored on the map and immediately resets the map to start accumulating new data.
- *Filtering data coming from sensors:-* We added the *UnWrap* and *UnWrapForSpecificDevice* components to accomplish this goal. The *UnWrap* component filters the payload from sensor data, while *UnWrapForSpecificDevice* component filters the payload for specific sensor devices. The *UnWrapForSpecificDevice* component takes as a configuration setting the identifier of the device to filter data for.
- *Generating and showing notifications:-* For this goal, we added the *SetThreshold* and *GenerateAndShowAlert* components. The *SetThreshold* component is configurable and takes the threshold or limit as a setting parameter. The set parameter is periodically sent out of the output port of the component. The *GenerateAndShowAlert* is configurable to specify the notification messages. For these two components to perform their tasks, we modified the existing *Compare* component to receive two inputs (the threshold from the *SetThreshold* component and some input data) and output a true/false value that can trigger the *GenerateAndShowAlert* component to generate and show notifications.
- *Sensing capabilities:-* To support sensing at the edge, we added the *ReadSoilMoisture*, *ReadTemperature*, and *ReadHumidity* components to read soil moisture, air temperature, and humidity, respectively. The three components directly receive input from the sensors.
- *Computing metrics at the edge:-* For computation at the edge, we added the following components: *SetConstant*, *Addition*, *Subtraction*, *Multiplication*, *Exponentiation*, and *ComputeEdgeAverage* components. The names of these components are indicative of the tasks that they perform.

Offline Accessibility Policies: We advance DisCoPar's offline accessibility both at the mobile and edge scope. DisCoPar-K implements policies to ensure that the offline accessibility components that execute on the mobile whenever used in an application graph can only be the last ones in the chain to connect to the server side. This means that an offline accessibility component on the mobile side cannot be connected to another mobile component or another offline accessibility component in a chain of successive components. For instance, in Fig. 5, the *InDatabaseBuffering* component that supports offline accessibility in the application can only be connected to the *ObservationDatabase*

component that runs on the server side. In a similar way, while implementing edge applications, the buffering component on the edge should be the last one to connect to the mobile side downstream as shown in Fig. 6.

Fig. 5. Application flow graph showing the *BufferObservation* connecting to *ObservationDatabase* component on the server-side.

Computation and Offline Accessibility at the Edge: In this work, we introduce edge components to perform computation at the edge. The edge components execute on the microcontroller on which sensors are attached. For instance, the *ReadSoilMoisture* component in Fig. 6 executes the *ReadSoilMoisture()* function which reads soil moisture from sensors.

At the edge, we implement a *BufferCache* component for buffering data in memory when the network becomes unavailable. The buffering in this component is based on the number of records. The buffer size can be specified as a configuration setting for the component. When the network becomes unavailable, sensor readings are stored until the buffer size is full. At this point, the oldest record is removed from memory to create room for new data. When the network becomes available, additional metadata is appended to the buffered data and then sent to the mobile application. The metadata contains the unique identifier for the sensor and the identifier of the connection link between the *BufferCache* component and the mobile component downstream, e.g., the link between the *BufferCache* and *UnWrap* components in Fig. 6.

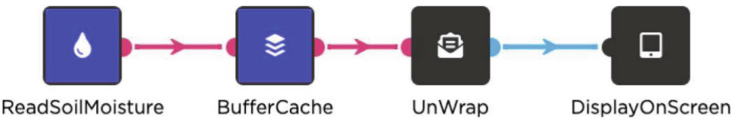

Fig. 6. Application flow graph featuring the *BufferCache* component in use.

Low Infrastructure Networking: The notion of communicating directly to a server is abstracted to allow sensors to communicate directly to applications running on mobile devices, as illustrated in Fig. 7. The sensing environment runs the edge application created by the domain expert. The edge application runs on top of a lightweight JavaScript engine[3] and Arduino[4]. We use web sockets (Socket.IO[5]) to communicate events (observables) carrying an event name and a payload to the mobile device. All clients that are subscribed to the event can then receive the payload from that particular event. Data coming from the sensors is packaged to carry a unique identifier for the sensor device, the payload, and the identifier of the link (connection) for the last blue component connecting to the black components. The link is exported as part of the graph deployed to the edge from the main application graph.

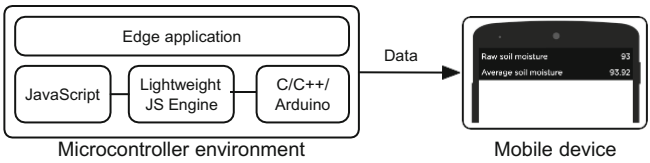

Fig. 7. Low infrastructure networking configuration.

Application Design and Deployment in DisCoPar-K: Figure 8 shows the workflow for designing applications in DisCoPar-K. Designing and validating the application is done on the VPE following the design choices for component composition and association. The entire application flow graph is exported and deployed to the mobile phone. For the edge, only the edge graph is exported and deployed to the microcontroller with sensors attached.

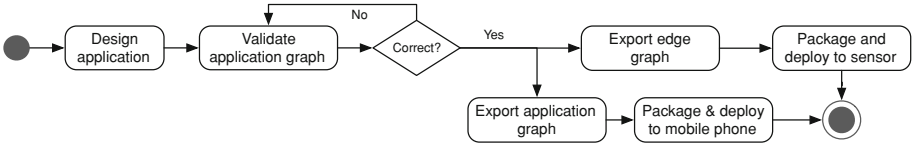

Fig. 8. Flow graph for application design and deployment in DisCoPar-K.

[3] https://duktape.org/.
[4] https://www.arduino.cc/en/software.
[5] https://socket.io/.

Exporting and Deploying the Edge Application: From an initial application graph, we build and deploy the edge application to the microcontroller with sensors attached. To build and export the edge application, we utilise the depth-first search method [28]. Using this method, we start from a root node and traverse the initial graph to generate the edge application. The root node is any blue (edge) component in the initial graph that has no predecessor component. We keep all root nodes in a list and process each item in the list. During processing, each neighbour of the root node is determined and visited. The visited nodes are kept in a list and their dependencies are determined. The dependencies are considered children of the visited nodes, kept in an ordered list. Using this information, we then build a segment of the initial application graph that is exported and deployed to the edge. For disjointed application graphs, we keep a list of visited nodes and iterate through all the items in the list. For each node in the list, we treat it as a root node and make a recursive call to all nodes that can be visited. For components that have a state, we export the state and pass it as an argument to the executed component function. On the microcontroller, the exported application is executed on top of the lightweight JavaScript engine (Fig. 7).

Runtime Environment at the Edge: At the edge, we use Arduino[6] and Duktape[7] to support edge applications. Duktape is a lightweight JavaScript engine that can run on microcontrollers. The engine permits integrating JavaScript programs with Arduino or C/C++ programs. This allows for deploying and executing JavaScript code on microcontrollers with sensors attached. We use the lightweight engine to allow implementation of the entire DisCoPar-K applications using one base language.

4 Validation and Discussion

To validate this work, we adopt a scenario-based approach and seek to answer the following questions: *1) How does DisCoPar-K fulfil the requirements and properties of the implemented scenarios?* and *2) How can DisCoPar-K as a low code development tool be used to improve the development of SAAs?* By definition, the scenario describes the set of interactions between different actors in a system and can comprise a concrete sequence of interaction steps (i.e., instance scenario) or a set of possible interaction steps (i.e., type scenario) [21]. More concretely, the validation scenario is conceptualised as a flow of computation tasks in the application [2]. In the scenario-based approach, the requirements of the scenario are validated against the expected behaviour of the application [2, 14, 21]. As such, we derived different scenarios that we use as case studies to validate this work. In the subsequent sections, we present each of the derived scenario and its implementation details.

[6] https://www.arduino.cc/en/software.
[7] https://duktape.org/.

4.1 Case Study 1: Monitoring Soil Moisture and Temperature

This case study was derived from the running example in Sect. 2 on corn seeding and sprouting. The goal of the use case is to ensure optimal sprouting of corn by keeping optimal soil moisture and temperature conditions. As such, this case study requires implementing the following requirements.

- Environment sensing to sense soil moisture and temperature.
- Computation at the edge to compute the averages of the data coming from sensors and soil heat capacity at the edge.
- Offline accessibility to buffer data at the edge when the network becomes unavailable.
- The microcontroller hosting the soil moisture and temperature sensors needs to send data directly to the mobile device (e.g., smartphone) without going through a centralised server.
- Track the global view of all soil moisture and temperature sensors installed on a farm. As such, the number of connected devices needs to be tracked.
- Accumulate soil moisture and temperature data from multiple sensors.
- Determine the maximum and minimum values for the accumulated soil moisture and temperature data.
- Compute the average of the accumulated soil moisture and temperature data.
- Track data for specific soil moisture and temperature sensors.
- Generate notifications based on specified limits and visualise average soil moisture over time.

The above requirements describe the expected behaviour of the implemented application. We compute the soil heat capacity (Q) using $Q = 4.2 \times 10^3 \times V \times (0.2 + W) \times \Delta T$, where V is the soil volume, W is the soil moisture, and ΔT is the change in temperature [35].

Fulfilling Key Properties for the Case Study: In this section, we describe how DisCoPar-K fulfils the requirements of case study 1.

1) *Environment sensing:* Figure 9 illustrates using the *ReadSoilMoisture* component to read and send soil moisture data. The data is sent to the *UnWrap* component that runs on the mobile device.

Fig. 9. Component for sensing soil moisture and sending data to mobile devices.

2) *Computing average soil moisture at the edge:* Figure 10 illustrates using the *ComputeEdgeAverage* component to compute the running average for the soil moisture at the edge. The running average for soil moisture is computed since the data is sampled continuously as a stream. The *UnWrap* component unpacks the average soil moisture as the payload and sends it to the next component downstream.

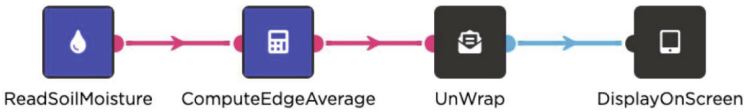

Fig. 10. Flow graph featuring computing the average soil moisture at the edge.

3) *Offline accessibility at the edge:* Figure 11 illustrates using the *BufferCache* component for offline accessibility at the edge. The component supports in-memory data buffering based on the number of records. The buffer size defines the maximum number of records stored.
4) *Microcontrollers hosting sensors sending data directly to mobile devices:* In the applications presented in Fig. 9, Fig. 10, Fig. 11 and Fig. 12, the blue (edge) components communicate directly with the black (mobile) components without going through a centralised server.

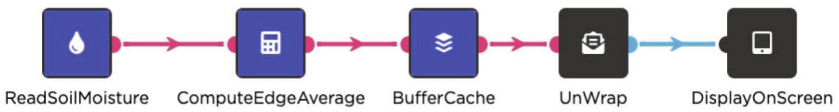

Fig. 11. Flow graph featuring buffering data at the edge for offline accessibility.

5) *Generating and showing notifications:* In Fig. 12, we use the *SetThreshold* component to specify the limit for generating notifications. The threshold value is compared with the incoming average soil moisture data using the *Compare* component. This component receives two inputs, i.e., the threshold and the data value. The output from the component triggers generating and showing a notification to the *GenerateAndShowAlert* component downstream.

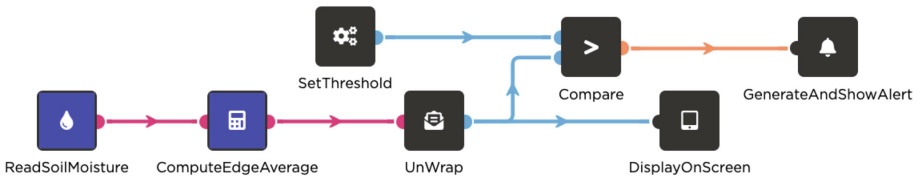

Fig. 12. Flow graph featuring setting thresholds and generating notifications.

6) *Tracking the global view and accumulating soil moisture and temperature data from multiple devices:* The application flow graph in Fig. 13 uses the *ConnectedDevices* component and the *DataArrayToTable* components to show the number of connected devices. The *DisplayOnScreen* component shows the number of connected devices on the screen, while the *DisplayAsList* shows the list of the connected components on the screen.

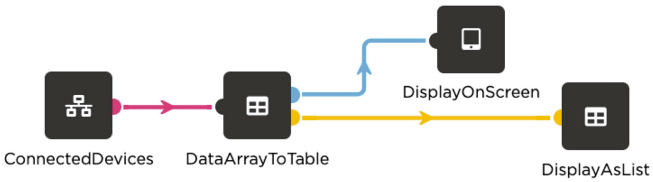

Fig. 13. Flow graph featuring showing the number of connected devices.

Figure 14 shows how to accumulate the average soil moisture data using the *DeviceAccumulator* component. The blue components run on the individual sensor devices, while the *DeviceAccumulator* runs on the mobile phone. The accumulated data is processed further to give more meaningful information, such as the global average for soil moisture.

Fig. 14. Flow graph featuring accumulating data from multiple sensor devices.

7) *Tracking data for specific soil moisture and temperature sensors:* Data coming from multiple devices can be unpacked, such that data for specific devices can be inspected. In Fig. 15, we use the *UnWrapForSpecificDevice* component to unpack soil moisture for a specific device.

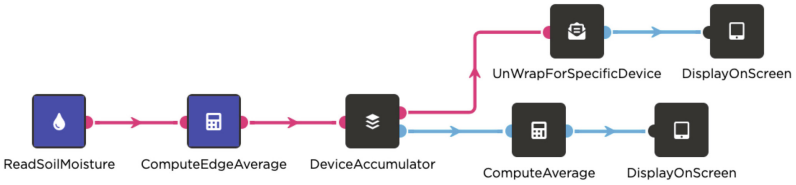

Fig. 15. Flow graph featuring unwrapping data for specific devices.

8) *Determining the maximum and minimum value for soil moisture accumulated from several devices:* In Fig. 16, we use the *ComputeMaximum* and *ComputeMinimum* components to determine the maximum and minimum values for data received from multiple devices. The determined values are displayed on the mobile screen using the *DisplayOnScreen* components.

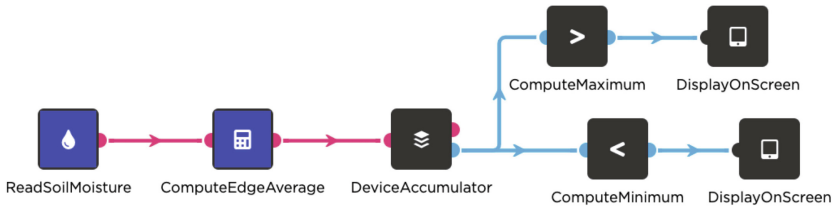

Fig. 16. Determining the maximum and minimum for data from multiple devices.

9) *Visualising average soil moisture over time:* In Fig. 17, we use the *PlotSoilMoisture-OverTime* component to visualise the average soil moisture over time. In addition to the average soil moisture, this component plots both the maximum and minimum thresholds. The maximum and minimum thresholds are set as constraints in the component configuration setting. The component receives, as input, the average soil moisture computed using the *ComputeAverage* component. The *ComputeAverage* component computes the average soil moisture for data accumulated from multiple devices using the *DeviceAccumulator* component.

Fig. 17. Flow graph featuring plotting the average soil moisture on a line chart.

Application Flow Graph and Preview of the Resulting Application: Figure 18 shows the overall application flow graph for monitoring soil moisture and temperature. The soil heat capacity is computed at the edge and the outcome is communicated to the mobile device. Parts 1 and 2 of Fig. 18 present the preview of the resulting application. The line chart shows the average soil moisture over time and the notifications generated displayed as pop-up messages on the mobile phone are generated at 57.5% soil moisture. Part 3 shows the deployed soil moisture (circled red) and temperature (circled white) sensors.

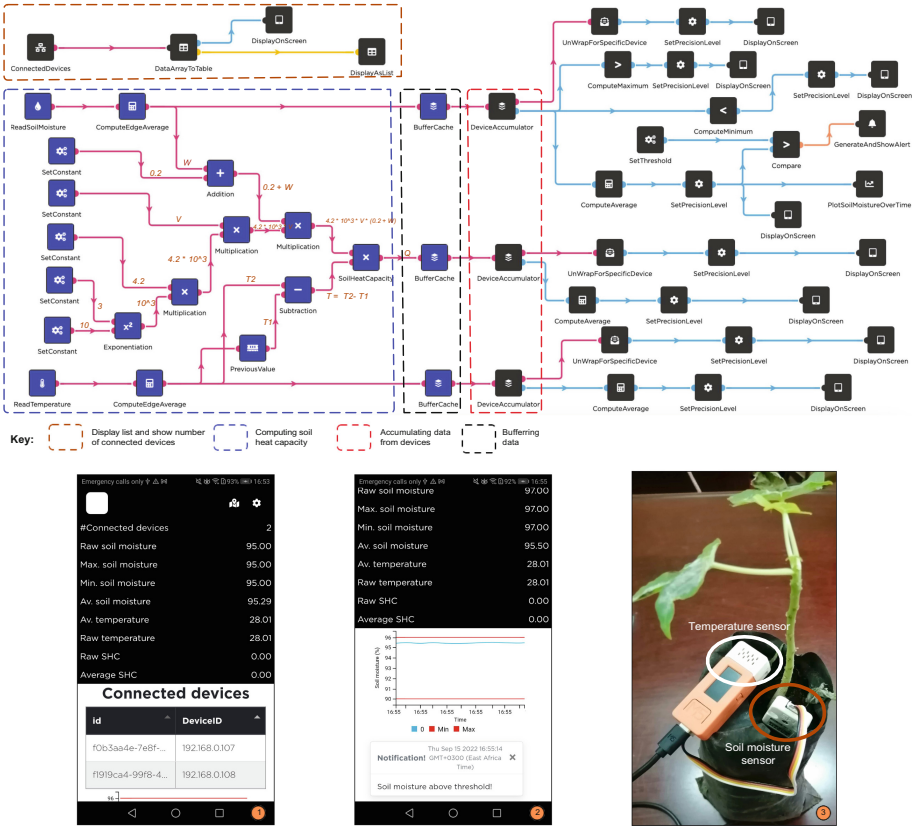

Fig. 18. Overall flow graph and preview of the mobile application for monitoring soil moisture and temperature.

4.2 Case Study 2: Monitoring Humidity and Data Collection

This use case is adapted from Serikul et al. [24]. In this use case, the goal of the farmer is to store paddy rice under optimal humidity conditions to maintain the quality of the rice and attract good prices in the market. The rice is packaged in paddy bags and stored in a warehouse after harvesting. At the warehouse, the paddy bags are stacked on top of each other. Previously, the humidity of the stored rice was randomly measured by inserting a digital humidity meter into selected paddy bags. Since in the warehouse the paddy bags are stacked on top of each other, it is difficult to measure the humidity in every bag. As such, this use case requires installing humidity sensors in random rice bags (e.g., per stack) as a representative of the entire warehouse. The humidity sensors collect data and send it to the farmer's mobile phone. The farmer uses the received humidity data to develop rice storage management plans. As such, this case study requires implementing the following requirements.

– Tracking average humidity data on a gauge chart.
– Tracking humidity data over time (e.g., on a line chart).

– Notifying the farmer when the humidity level goes beyond set limits.
– Collecting the humidity data into a database for storage and creating reports as CSV (comma-separated values) files.

Application Flow Graph and Preview of the Resulting Application: Figure 19 shows the preview of the application flow graph that meets the requirements of this use case. Tracking humidity is done using the *ReadHumidity* component. The average humidity is computed at the edge using the *ComputeAverage* component and its accumulation from different devices is done using the *DeviceAccumulator* component. Tracking humidity for individual paddy rice bags (i.e., individual humidity sensors) is done using the *UnWrapForSpecificDevice* component. Before sending the data to the database, it is converted into observations by the *DataToObservation* component. To handle network outages, the application utilises the *InDatabaseBuffering* component. Humidity data is collected into a database using the *ObservationDatabase* component and exported to CSV report via the *DisplayAsTable* component. The humidity data for each device is plotted on a gauge chart using the *PlotGaugeChart* component. The average humidity over time is plotted on a line chart via the *PlotHumidityOverTime* component. Humidity thresholds are set using the *SetThreshold* component. The *Compare* component performs the comparison of the set threshold and incoming humidity data. Lastly, alerts are generated using the *GenerateAndShowAlert* component.

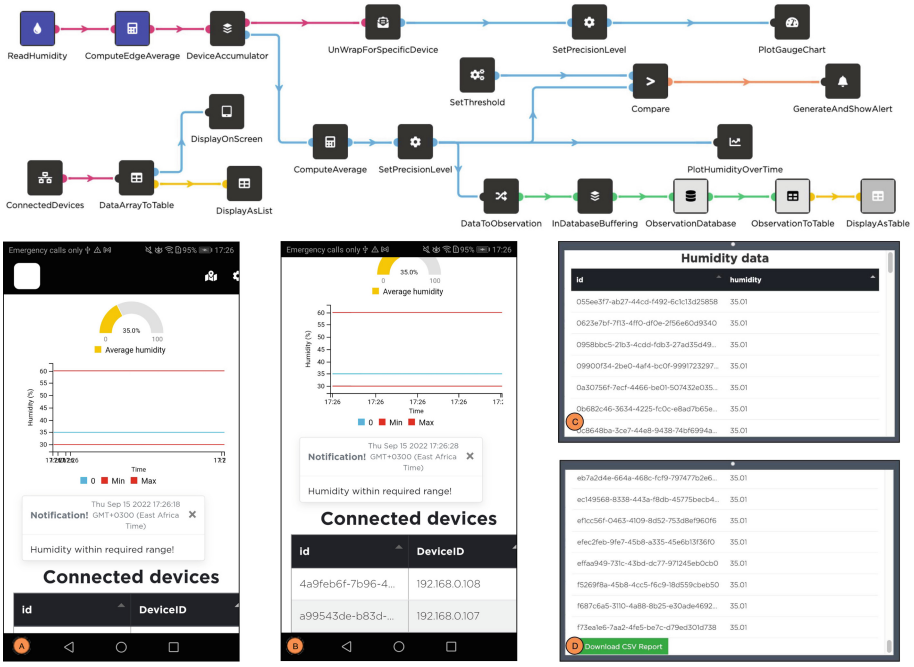

Fig. 19. Flow graph and preview of the humidity monitoring and data collection application.

Parts A and B of Fig. 19 show the preview of the resulting mobile application featuring the average humidity on a gauge chart and line chart. In addition, part B shows the generated notification and the list of connected devices. The list of connected devices is updated in real-time using the *ConnectedDevices* component to allow tracking when devices go off. Parts C and D show the humidity data displayed on a web dashboard. The data on the dashboard can be exported as a CSV report for further analysis by clicking the "Download CSV Report" button in part D. In comparison to the application presented in [24], our implementation adds 1) offline availability using the *InDatabaseBuffering* component to store humidity data when the network becomes unavailable, and 2) setting thresholds and generating notifications when the humidity goes beyond the required limit.

4.3 Case Study 3: Tracking Application Data

From our interaction with farmers in developing regions, they often collect plant specific data to monitor growth and development. For instance, at the sprouting stage, the farmers need to track the number of leaves per plant, the colour of the leaves and plant height. As such, the goal of this use case is to collect and track plant-specific data. Tracking the plant-specific data can be done using quick response codes as plant labels. Once the labels are scanned, a data entry survey is invoked that the farmers can use to directly input the data into the application. The data is sent from the application to the server from where it can be visualised on a dashboard. This use case requires implementing the following requirements.

- Scanning (reading) plant labels and invoking a data collection survey.
- Sending the data input into the application to a database and displaying it on a dashboard.
- Displaying collected data on a dashboard.
- Exporting the data stored in the database as a report in a CSV file for further analysis.

Application Flow Graph and Preview of the Resulting Application: The application shown in Fig. 20 fulfils the above requirements. Reading the plant labels is done using the *ReadQRCode* component. The *ObservationPopUpSurvey* component is used to generate the data collection survey. Part 1 of Fig. 20 shows the preview of the data collection survey on a mobile phone. The data collected is sent to the server for storage using the *ObservationDatabase* component and displayed on a web dashboard using the *DisplayAsTable* component, as shown in part 2 of Fig. 20. From the web dashboard, the data can be exported as a CSV file for further analysis.

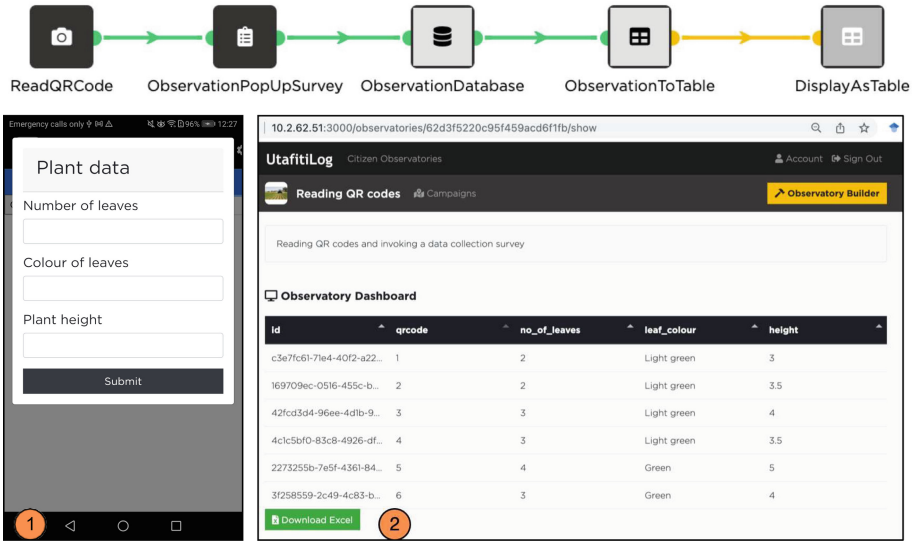

Fig. 20. Preview for the data collection application with data on a dashboard.

4.4 Discussion

In this work, our validation sought to answer how DisCoPar-K fulfils the requirements and properties of the implemented scenarios and how it can be used to improve the development of SAAs.

Fulfilling the requirements and properties of the implemented scenarios: On this issue, DisCoPar-K offers components for sensing and monitoring environmental conditions (e.g., soil moisture, temperature and humidity). In addition, DisCoPar-K offers components for performing computations at the edge, buffering data at the edge when the network becomes unavailable, and supporting microcontrollers to communicate with mobile devices without a centralised server. Data received on the server side can be visualised on web dashboards.

Improving the development of SAAs: This issue is demonstrated in several ways. First, the visual components, drag-and-drop, point-and-click utilities, prebuilt UI components, and binding to data sources that DisCoPar-K provides can be used intuitively. Secondly, the pre-built components that DisCoPar-K provide hideaway issues like memory management and distribution, which can take considerable time to configure manually. As such, DisCoPar-K can speed up the application development process and make software programming more accessible to domain experts outside software engineering. This means that LCDTs in general can motivate small-scale and medium-scale farmers to adopt and use mobile computing technologies in their farming activities. Lastly, although the case studies presented in this work focus on sensing and data collection, the same methodology can be applied to other types of SAAs and other domains.

5 Conclusion and Future Work

Low code development tools provide visual programming environments with components that can be used to construct smart agriculture applications. The components abstract and transparently provide a software infrastructure for constructing smart agriculture applications. We believe that the visual components are easier to use, especially for non-experienced developers. However, to meet requirements for different domains, the low code development tools need to have the necessary pre-requisite components that can be used to implement different applications. These components provide application properties and support computational tasks that are specific and tailored to a specific domain. In a nutshell, this paper identifies properties in low code development tools to support implementing smart agriculture applications and presents DisCoPar-K, a low code development tool that supports low infrastructure networking, computation at the edge, and offline accessibility at the edge. Domain experts with less programming experience can use DisCoPar-K to implement smart agriculture applications. For future work, we aim to perform a heuristic-based evaluation of smart agriculture applications constructed using low code development tools.

Acknowledgement. This work is supported by the Legumes Centre for Food and Nutrition Security (LCEFoNS) programme which is funded by VLIR-UOS. The programme is a North-South Collaboration between the Katholieke Universiteit Leuven, Vrije Universiteit Brussel (both in Belgium) and Jomo Kenyatta University of Agriculture and Technology (Kenya).

References

1. Node-RED: Low-code programming for event-driven applications. https://nodered.org/. Accessed 30 Aug 2021
2. Arnold, D., Corriveau, J.P., Shi, W.: Scenario-based validation: beyond the user requirements notation. In: 2010 21st Australian Software Engineering Conference, pp. 75–84. IEEE (2010). https://doi.org/10.1109/ASWEC.2010.29
3. AtmosphericIoT: Simplify your IoT development with Atmosphere IoT Studio. https://atmosphereiot.com/studio/. Accessed 26 Aug 2021
4. Axonize: The smarter way to realize smart business potential. https://www.axonize.com. Accessed 31 Aug 2021
5. Babou, C.S.M., Sane, B.O., Diane, I., Niang, I.: Home edge computing architecture for smart and sustainable agriculture and breeding. In: Proceedings of the 2nd International Conference on Networking, Information Systems & Security. NISS19, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3320326.3320377
6. Bexiga, M., Garbatov, S., Seco, J.A.C.: Closing the gap between designers and developers in a low code ecosystem. In: MODELS 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3417990.3420195
7. Blackstock, M., Lea, R.: IoT mashups with the WoTKit. In: Proceedings of 2012 International Conference on the Internet of Things, IOT 2012, Wuxi, China, pp. 159–166 (2012). https://doi.org/10.1109/IOT.2012.6402318
8. Blackstock, M., Lea, R.: Toward a distributed data flow platform for the Web of Things (Distributed Node-RED). In: ACM International Conference Proceeding Series, pp. 34–39 (2014). https://doi.org/10.1145/2684432.2684439

9. Blackstock, M., Lea, R.: FRED: A hosted data flow platform for the IoT. In: Proceedings of the 1st International Workshop on Mashups of Things and APIs. MOTA 2016, Association for Computing Machinery, New York (2016). https://doi.org/10.1145/3007203.3007214
10. Choudharyor, N., Arya, V.: Salesforce IoT cloud platform. In: Singh Mer, K.K., Semwal, V.B., Bijalwan, V., Crespo, R.G. (eds.) Proceedings of Integrated Intelligence Enable Networks and Computing. Algorithms for Intelligent Systems, pp 301–309. Springer, Singapore (2021). https://doi.org/10.1007/978-981-33-6307-6_31
11. El-Sanatawy, A.M., El-Kholy, A.S.M., Ali, M.M.A., Awad, M.F., Mansour, E.: Maize seedling establishment, grain yield and crop water productivity response to seed priming and irrigation management in a mediterranean arid environment. Agronomy **11**(4), 756 (2021). https://doi.org/10.3390/agronomy11040756
12. Ermi S.T., Rif'an, M.: Internet of Things (IoT): BLYNK framework for smart home. In: 3rd UNJ International Conference on Technical and Vocational Education and Training 2018, pp. 579–586 (2019). https://doi.org/10.18502/kss.v3i12.4128
13. Gao, Z., et al.: A novel approach to evaluate soil heat flux calculation: an analytical review of nine methods. J. Geophys. Res.: Atmos. **122**(13), 6934–6949 (2017). https://doi.org/10.1002/2017JD027160
14. Gregoriades, A., Sutcliffe, A.: Scenario-based assessment of nonfunctional requirements. IEEE Trans. Softw. Eng. **31**(5), 392–409 (2005). https://doi.org/10.1109/TSE.2005.59
15. Henkel, M., Stirna, J.: Pondering on the key functionality of model driven development tools: the case of mendix. In: Forbrig, P., Günther, H. (eds.) BIR 2010. LNBIP, vol. 64, pp. 146–160. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16101-8_12
16. Kleinfeld, R., Steglich, S., Radziwonowicz, L., Doukas, C.: Glue.things: a mashup platform for wiring the internet of things with the internet of services. In: Proceedings of the 5th International Workshop on Web of Things, WoT 2014, pp. 16–21. Association for Computing Machinery, New York (2014). https://doi.org/10.1145/2684432.2684436
17. Michael, L., Field, D.: Mendix as a solution for present gaps in computer programming in higher education. In: Twenty-Fourth Americas Conference on Information Systems, pp. 1–5 (2018)
18. Noor, J., Sandha, S.S., Garcia, L., Srivastava, M.: DDFLOW visualized declarative programming for heterogeneous IoT networks on Heliot Testbed platform: demo abstract. In: Proceedings of the International Conference on Internet of Things Design and Implementation, IoTDI 2019, ACM, New York, NY, USA, pp. 287–288 (2019). https://doi.org/10.1145/3302505.3312598
19. O'Grady, M., Langton, D., O'Hare, G.: Edge computing: a tractable model for smart agriculture? Artif. Intell. Agric. **3**, 42–51 (2019). https://doi.org/10.1016/j.aiia.2019.12.001
20. Ragasa, C.: Effectiveness of the lead farmer approach in agricultural extension service provision: Nationally representative panel data analysis in Malawi. Land Use Policy **99**, 104966 (2020). https://doi.org/10.1016/j.landusepol.2020.104966
21. Ryser, J., Glinz, M.: A scenario-based approach to validating and testing software systems using statecharts. In: 12th International Conference on Software and Systems Engineering and their Applications (ICSSEA 2099). CNAM (1999). https://doi.org/10.5167/uzh-205008
22. Sahay, A., Di Ruscio, D., Pierantonio, A.: Understanding the role of model transformation compositions in low-code development platforms. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3417990.3420197
23. Salesforce: Discover low-code tools to reimagine workflows and increase productivity. https://www.salesforce.com/products/platform/low-code/. Accessed 26 Aug 2021

24. Serikul, P., Nakpong, N., Nakjuatong, N.: Smart farm monitoring via the Blynk IoT platform: case study: humidity monitoring and data recording. In: 2018 Sixteenth International Conference on ICT and Knowledge Engineering, pp. 70–75. IEEE (2018). https://doi.org/10.1109/ICTKE.2018.8612441
25. Simplifier: Enterprise Apps Made Simple. https://simplifier.io/en/. Accessed 27 Aug 2021
26. Sudozai, M.I., Tunio, S., Chachar, Q., Rajpar, I.: Seedling establishment and yield of maize under different seed priming periods and available soil moisture. Sarhad J. Agric. **29**, 515–528 (2013)
27. Szydlo, T., Brzoza-Woch, R., Sendorek, J., Windak, M., Gniady, C.: Flow-based programming for IoT leveraging fog computing. In: 2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pp. 74–79 (2017). https://doi.org/10.1109/WETICE.2017.17
28. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972). https://doi.org/10.1137/0201010
29. Waszkowski, R.: Low-code platform for automating business processes in manufacturing. IFAC-PapersOnLine **52**(10), 376–381 (2019). https://doi.org/10.1016/j.ifacol.2019.10.060,13thIFACWorkshoponIntelli-gentManufacturingSystemsIMS2019
30. Wolfert, S., Ge, L., Verdouw, C., Bogaardt, M.: Big data in smart farming: a review. Agric. Syst. **153**, 69–80 (2017). https://doi.org/10.1016/j.agsy.2017.01.023
31. Zaman, J.: DISCOPAR: A Visual Reactive Flow-Based Domain-Specific Language for Constructing Participatory Sensing Platforms. Ph.D. thesis, Vrije Universiteit Brussel (2018)
32. Zaman, J., Kambona, K., De Meuter, W.: DISCOPAR: A visual reactive programming language for generating cloud-based participatory sensing platforms, pp. 31–40. REBLS 2018, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3281278.3281285
33. Zaman, J., Kambona, K., De Meuter, W.: A reusable & reconfigurable citizen observatory platform. Futur. Gener. Comput. Syst. **114**, 195–208 (2021). https://doi.org/10.1016/j.future.2020.07.028
34. Zenodys: A Fully Decentralised Data and Service Marketplace for Everyone. https://www.zenodys.com/wp-content/uploads/zenodys-ico-whitepaper.pdf. Accessed 30 Aug 2021
35. Zhang, Z., et al.: The change characteristics and interactions of soil moisture and temperature in the farmland in Wuchuan county, inner Mongolia China. Atmosphere **11**(5), 503 (2020). https://doi.org/10.3390/atmos11050503