



VRIJE  
UNIVERSITEIT  
BRUSSEL



Dissertation submitted in fulfilment of the requirements for the degree of  
Doctor of Sciences of the Vrije Universiteit Brussel

# **DISCOPAR-KILIMO: A LOW-CODE DEVELOPMENT ENVIRONMENT**

Geared Towards Smart Agriculture  
Applications

**Isaac Nyabisa Oteyo**

**January 2024**

**Promotors:**

Prof. Dr. Elisa Gonzalez Boix, Vrije Universiteit Brussel

Prof. Dr. Wolfgang De Meuter, Vrije Universiteit Brussel

Dr. Angel Luis Scull Pupo, Vrije Universiteit Brussel

**Faculty of Sciences and Bio-engineering Sciences**

# DisCoPar-Kilimo: A Low-Code Development Environment

## Geared Towards Smart Agriculture Applications

Isaac Nyabisa Oteyo

Dissertation submitted in fulfilment of the  
requirements for the degree of Doctor of Sciences of the Vrije  
Universiteit Brussel

January 26, 2024

### **Jury:**

Prof. Dr. Elisa Gonzalez Boix, Vrije Universiteit Brussel (promotor)  
Prof. Dr. Wolfgang De Meuter, Vrije Universiteit Brussel (promotor)  
Dr. Angel Luis Scull Pupo, Vrije Universiteit Brussel (promotor)  
Prof. Dr. Ann Nowe, Vrije Universiteit Brussel (chair)  
Prof. Dr. Abdellah Touhafi, Vrije Universiteit Brussel (secretary)  
Prof. Dr. Robert Hirschfeld, University of Potsdam, Germany  
Prof. Dr. Engineer Bainomugisha, Makerere University, Uganda

Vrije Universiteit Brussel  
Faculty of Sciences and Bio-engineering Sciences  
Department of Computer Science  
Software Languages Laboratory

© 2023 Isaac Nyabisa Oteyo

Printed by  
Crazy Copy Center Productions  
VUB Pleinlaan 2, 1050 Brussel  
Tel/fax: +32 2 629 33 44  
crazycopy@vub.ac.be  
www.crazycopy.be

ISBN: 9789464948059

NUR: 958/965/989

THEME: UMZ

The work in this dissertation has been funded by a PhD fellowship of the VLIR-UOS - Grant number KE2017IUC037A101.

All rights reserved. No part of this publication may be produced in any form by print, photo-print, microfilm, electronic or any other means without permission from the author.

# Abstract

Smart agriculture applications (SAAs) improve farming activities in modern farms. Today, designing and implementing SAAs is difficult and costly. Typically, SAAs orchestrate distributed components deployed in micro-controllers, smartphones, and cloud services. Implementing SAAs requires highly skilled engineers as it entails carefully handling distribution to enable the different parts of the system to communicate. However, “skilled developers are hard to find”. Moreover, the network infrastructure in rural areas is limited and unreliable, resulting in inexistent or intermittent connections. This thesis aims to simplify the implementation of SAAs and study software techniques to enable domain experts to implement them.

Smart agriculture applications are often constructed using conventional programming languages that require software development knowledge, which domain experts usually do not have. Low-code development environments (LCDEs) have emerged as an alternative for users lacking technical expertise. The LCDEs offer visual programming environments with “ready-to-use” components, making software development accessible to all technical skill levels. However, certain limitations hinder the widespread usage of LCDEs to develop SAAs. First, the state-of-the-art LCDEs lack components for specifying edge computations on devices installed in the environment (e.g., crop fields and farms) and assume that sensors gather data and send it to the cloud or server. Second, there are no components to support coordinating the communication between the different parts that compose SAAs, e.g., coordinating the communication between the edge and mobile components. Lastly, the output of the computations done on the devices installed in the environment must be sent to other components on the mobile phones or servers over communication networks, which can fail. The existing LCDEs have limited support for handling failures between the components composing SAAs when the



networks become intermittent or unavailable.

In this dissertation, we first identify four categories of components that aim to facilitate the development of SAAs using a low-code development environment. The components encompass functionalities to observe and monitor prevailing environmental conditions, support computation on the devices installed in the environment, i.e., computation at the edge, coordinate the communication between the different components that compose SAAs, and effectively handle partial network failures. We prototype the four component categories by extending a low-code development environment, DisCoPar, with “ready-to-go” components for SAAs. The resulting low-code development environment, which we named DisCoPar-Kilimo, adopts a flow-based programming environment where applications are represented as interconnected nodes that transmit data to one another. The nodes represent software components that perform specific computational tasks in the application. The components are presented in a palette from which domain experts can pick and use them to construct software. The key contributions of this dissertation consist of a set of properties and readily deployable components for building SAAs. Technically, we also contribute DisCoPar-Kilimo, a domain-specific low-code development environment for implementing SAAs.

To assess the effectiveness of DisCoPar-Kilimo, we employed a scenario-based approach and successfully implemented seven representative smart agriculture scenarios on it. Four of these scenarios were based on the identified properties, while the remaining three were derived from existing literature and our experiences with agricultural extension workers (i.e., farmers) in Kenya. The implemented scenarios demonstrate how DisCoPar-Kilimo can be used to construct SAAs and its flexibility in implementing SAAs. From the implemented scenarios, DisCoPar-Kilimo can be used intuitively by domain experts as it hides away application development issues like memory management and coordinating communication, which can take considerable time to configure manually. In conclusion, DisCoPar-Kilimo is unique and can be considered a first in new and future-generation LCDEs for SAAs.

# Samenvatting

Slimme landbouwapplicaties (SAA's) worden gebruikt om de landbouwactiviteiten in moderne boerderijen te verbeteren. Vandaag de dag is het ontwerpen en implementeren van SAA's moeilijk en duur. Typisch zijn SAA's systemen die gedistribueerde componenten in microcontrollers, smartphones en cloudservices orkestreren. Er zijn hoogopgeleide ingenieurs nodig om SAA's te implementeren, omdat er zorgvuldig met (network) distributie moet worden omgegaan om de verschillende onderdelen waaruit het systeem is opgebouwd met elkaar te laten communiceren. Bekwame ontwikkelaars zijn echter “moeilijk te vinden”. Verder is de netwerkinfrastructuur in op het platteland beperkt en onbetrouwbaar, wat er voor zorgt dat er vaak geen of haperende verbindingen zijn. Dit proefschrift beoogt de implementatie van SAA's en softwaretechnieken te bestuderen om domeinexperts in staat te stellen ze te implementeren.

SAA's worden vaak opgebouwd met behulp van tekstuele PL's die kennis van softwareontwikkeling vereisen, kennis die domeinexperts vaak missen. Low-code ontwikkelomgevingen (LCDE's) zijn opgedoken als alternatief voor gebruikers zonder technische expertise. LCDE's bieden visuele programmeeromgevingen met “kant-en-klare” componenten, waardoor softwareontwikkeling toegankelijker wordt voor alle technische vaardigheidsniveaus. Er zijn echter een aantal limitaties die de verspreiding van LCDE's voor de ontwikkeling van SAA's in de weg staan. Ten eerste missen de nieuwste LCDE's componenten voor het specificeren van berekeningen op apparaten die in de omgeving zijn geïnstalleerd (bijvoorbeeld boerderijen). Ten tweede zijn er geen componenten voor het coördineren van de communicatie tussen apparaten in de omgeving. Tot slot is er beperkte ondersteuning voor het afhandelen van storingen tussen de verschillende componenten wanneer netwerken onbeschikbaar worden.

In dit proefschrift identificeren we vier categorieën van componen-

ten gericht op het faciliteren van de ontwikkeling van SAA's met behulp van een LCDE. De componenten omvatten functionaliteiten voor het observeren van omgevingscondities, het ondersteunen van berekeningen op apparaten die geïnstalleerd zijn in de omgeving, het coördineren van de communicatie tussen de verschillende componenten en tot slot het effectief afhandelen van netwerkstoringen. We prototypen vier componentencategorieën door een bestaande LCDE, DisCoPar, uit te breiden met “kant-en-klare” componenten die zijn afgestemd op SAA's. De resulterende LCDE, die we DisCoPar-Kilimo hebben genoemd, gebruikt een flow-gebaseerde programmeeromgeving waar applicaties worden voorgesteld als onderling verbonden bouwblokken die gegevens naar elkaar kunnen verzenden. De bouwblokken vertegenwoordigen softwarecomponenten die specifieke rekentaken uitvoeren in de toepassing. De componenten worden gepresenteerd in een palet waaruit domeinexperts kunnen kiezen en ze gebruiken om software te bouwen. De belangrijkste bijdrage van dit proefschrift is de identificatie van de essentiële eigenschappen en gemakkelijk inzetbare componenten die nodig zijn voor het bouwen van SAA's. Op technisch gebied dragen we bij met DisCoPar-Kilimo, een domeinspecifieke LCDE voor het implementeren van SAA's.

Om de effectiviteit van DisCoPar-Kilimo te beoordelen, hebben we een op scenario's gebaseerde validatiemethode toegepast en met succes zeven representatieve slimme landbouwscenario's erop geïmplementeerd. Vier van deze scenario's waren gebaseerd op de geïdentificeerde eigenschappen, terwijl de overige drie waren afgeleid uit bestaande literatuur. De geïmplementeerde scenario's laten zien hoe DisCoPar-Kilimo kan worden gebruikt om SAA's te bouwen en hoe flexibel het is om SAA's te implementeren. Uit de geïmplementeerde scenario's blijkt dat DisCoPar-Kilimo intuïtief kan worden gebruikt door domeinexperts, omdat het problemen bij het ontwikkelen van toepassingen, zoals geheugenbeheer en het coördineren van communicatie, die veel tijd kunnen kosten om handmatig te configureren, verbergt. Als conclusie kan gesteld worden dat DisCoPar-Kilimo uniek is en beschouwd kan worden als de eerste in zijn soort van een nieuwe en toekomstige generatie LCDE's voor SAA's.

# Acknowledgement

This journey began over half a decade ago, with several people coming into play. First, I would like to thank my promotors, colleagues, friends, and family. I want to thank my promotors, Professor Dr. Elisa Gonzalez Boix, Professor Dr. Wolfgang De Meuter, and Dr. Angel Luis Scull Pupo, for their guidance over the years. Wolf, thank you for the short meeting on April 13, 2017, in Nairobi. That meeting changed the course of my academic journey. Elisa, thank you for steering me in the right direction throughout this research period. Most importantly, thank you for picking me up from the airport in Zaventem in September 2017, giving me a quick tour and introduction to Brussels, and finally delivering me to the “Cuba house” in Brussels, which eventually became my second home. Thank you, Scull, for the countless meetings that we had in the evenings and during the weekends on my work.

Moreover, I would like to thank all current SOFTies and ex-SOFTies, especially the DISCOers (DIStribution and CONcurrency) people, for all the feedback you provided in every research presentation of my work. I thank the secretaries of our department, who were always ready to help me not only with academic matters but also with visa- and residency-related matters every year. Thank you, Brigitte! Lara (not at SOFT anymore), thanks for all your patience, keen listening, and prompt action on my requests.

I want to sincerely thank the members of my jury (Prof. Dr. Ann Nowe, Prof. Dr. Abdellah Touhafi, Prof. Dr. Robert Hirschfeld, and Prof. Dr. Engineer Bainomugisha) for the time that you spent reading this dissertation, your suggestions and feedback to improve its final version – the version whose acknowledgement you are reading now. Thanks to the entire team (simply the “legumes” people) at the Legume Centre of Excellence in Food and Nutritional Security (LCEFoNs) for all the

support, from writing the project to securing funds from VLIR–UOS and giving me a chance to pursue this study under the project. Thanks to Prof. Dr. Stephen Kimani and your counterpart in the North for being great stewards for the project “*número cuatro*”.

Many thanks to all my friends and teachers who contributed to my education from the early years of my academic life till now. You are too many to name individually, but I am thankful for the knowledge and human values you taught me. Also, I would like to thank Olga Lydia Hernandez Castañeda, Patrick Vanderheere, and Cirelda Hernandez Castañeda who made me part of their family in Brussels. Thanks for bringing together a small gang (myself, Scull, Camilo, Yuniór, Freddie, Jose Luis, etc.) to your house. I want to thank every member of my big (and extended) family, especially my parents (all who, in one way or another, call me their child) and all my siblings (everyone who fits the title of brother, sister, nephew, and niece) in every way that applies for all the support you have accorded me during the entire time of the study. Lastly, I would like to thank Ann and our “small gang”, especially for allowing me to spend considerable time away from your lives and patiently waiting for me to pursue my academic endeavours.

This work was funded by a PhD scholarship of the VLIR–UOS to promote global North and global South collaboration.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Smart Agriculture Applications . . . . .	3
1.1.1	Developed vs Developing Regions . . . . .	3
1.1.2	Implementing Smart Agriculture Applications . . . . .	4
1.2	Problem Statement . . . . .	5
1.3	Our Approach . . . . .	6
1.4	Contributions . . . . .	8
1.4.1	Supporting Publications . . . . .	8
1.5	Dissertation Outline . . . . .	10
<b>2</b>	<b>State of the Art of Smart Agriculture Applications</b>	<b>13</b>
2.1	Smart Agriculture Processes . . . . .	13
2.2	Driver Scenarios for Smart Agriculture . . . . .	15
2.2.1	Sensing Farm Conditions . . . . .	15
2.3	Properties for Smart Agriculture Applications . . . . .	22
2.4	State-of-the-Art of Smart Agriculture Applications . . . . .	23
2.4.1	Applications that Support Sensing . . . . .	24
2.4.2	Applications that Support Computation at the Edge . . . . .	25
2.4.3	Applications that Support Handling Partial Failures . . . . .	25
2.4.4	Discussion . . . . .	26
2.5	State-of-the-Art of Low-Code Development Environments . . . . .	26
2.5.1	Notation Used in Flow-Based VPLs . . . . .	28
2.5.2	Review of Existing Low-Code Environments . . . . .	30
2.6	Conclusion . . . . .	33
<b>3</b>	<b>DisCoPar</b>	<b>35</b>

3.1	Architectural Overview of DisCoPar . . . . .	35
3.1.1	DisCoPar Layers . . . . .	37
3.2	DisCoPar Visual Programming Environment . . . . .	38
3.2.1	DisCoPar Components . . . . .	38
3.2.2	Graph Validation . . . . .	44
3.2.3	Handling Partial Failures . . . . .	45
3.3	DisCoPar by Example . . . . .	45
3.4	Conclusion . . . . .	48
<b>4</b>	<b>DisCoPar-Kilimo</b>	<b>51</b>
4.1	Our Approach in a Nutshell . . . . .	51
4.2	Architectural Overview of DisCoPar-Kilimo . . . . .	53
4.3	DisCoPar-Kilimo Visual Programming Environment . . . . .	55
4.3.1	Ensuring Support for Environment Sensing . . . . .	55
4.3.2	Ensuring Support for Computation at the Edge . . . . .	58
4.3.3	Ensuring Support for Coordination with the Edge . . . . .	60
4.3.4	Ensuring Support for Handling Partial Failures . . . . .	60
4.3.5	Tracking Connected Edge Devices . . . . .	67
4.3.6	Accumulating Data from Multiple Edge Devices . . . . .	68
4.4	Developing and Deploying Applications by Example . . . . .	71
4.4.1	Example application . . . . .	72
4.4.2	Deploying the example application . . . . .	74
4.5	Extensions to DisCoPar . . . . .	75
4.6	Conclusion . . . . .	76
<b>5</b>	<b>DisCoPar-Kilimo Implementation</b>	<b>77</b>
5.1	Basic Building Blocks . . . . .	77
5.1.1	Application Graphs . . . . .	77
5.1.2	Executing Application Graphs . . . . .	78
5.1.3	Basic Application Example . . . . .	78
5.1.4	Implementing Components in DisCoPar . . . . .	79
5.1.5	Distributed Connections . . . . .	81
5.2	DisCoPar-Kilimo . . . . .	81
5.2.1	Computation at the Edge . . . . .	81
5.2.2	Environment Sensing . . . . .	83
5.2.3	Components for Computation at the Edge . . . . .	85

5.2.4	Handling Partial Failures on the Mobile Scope . . .	86
5.2.5	Validating Flow-Graphs to Handle Partial Failures .	91
5.2.6	Handling Partial Failures at the Edge . . . . .	93
5.2.7	Tracking Connected Edge Devices . . . . .	95
5.2.8	Accumulating Data from Multiple Edge Devices . .	97
5.3	Deploying Applications . . . . .	100
5.3.1	Designing Applications in DisCoPar-Kilimo . . . .	101
5.3.2	Deploying DisCoPar-Kilimo Applications . . . . .	104
5.4	Conclusion . . . . .	107
<b>6</b>	<b>Validation</b>	<b>109</b>
6.1	Validation Approach . . . . .	109
6.2	Validation Scenarios . . . . .	110
6.2.1	Scenario 1: Monitoring Soil Moisture . . . . .	110
6.2.2	Scenario 2: Computing Average Soil Moisture and Keeping Data at the Edge . . . . .	112
6.2.3	Scenario 3: Monitoring Soil Moisture Using more than one Edge Device . . . . .	113
6.2.4	Scenario 4: Tracking Connected Edge Devices . . .	115
6.2.5	Scenario 5: Tracking and Monitoring Paddy Rice Storage Conditions . . . . .	117
6.2.6	Scenario 6: Collecting Data Using Mobile Applica- tions . . . . .	121
6.2.7	Scenario 7: Monitoring Soil Moisture and Temper- ature in Corn Seeding and Sprouting . . . . .	122
6.3	Discussion . . . . .	125
6.4	Conclusion . . . . .	126
<b>7</b>	<b>Conclusion</b>	<b>127</b>
7.1	Problem Statement Revisited . . . . .	127
7.2	Research Approach Revisited . . . . .	128
7.3	Contributions . . . . .	129
7.4	Shortcomings and Future Work . . . . .	129
7.5	Concluding Remarks . . . . .	131
<b>A</b>	<b>DisCoPar Application Graph</b>	<b>133</b>



<b>B</b>	<b>Companion Functions for Edge Components</b>	<b>137</b>
<b>C</b>	<b>Extracting and Exporting Edge Graph</b>	<b>139</b>
<b>D</b>	<b>Implementation of the Networking Code for Coordination with the Edge</b>	<b>143</b>
<b>E</b>	<b>Deployment</b>	<b>145</b>
E.1	Resources Required . . . . .	145
E.2	Mobile Application . . . . .	146
E.3	Edge Application . . . . .	146

# List of Figures

1.1	An example illustration of different modern farming technologies. (a) Auto-pilot tractors and sprayer machines, (b) Crop sensing, (c) Documenting crop fields and (d) Monitoring animal welfare. . . . .	2
1.2	An example application that shows a farmer using a mobile phone to receive data about a corn field. The data comes from sensors attached to a microcontroller deployed to the corn field. . . . .	3
1.3	DisCoPar-Kilimo in action featuring a canvas with a smart agriculture application for sensing soil moisture. Components are drawn from the component menu, dropped on the canvas and connected to compose applications. . . . .	7
2.1	Illustration of key smart agriculture processes grouped into four key areas. The processes are data collection, transformation and processing, dissemination, evaluation and impact assessment. . . . .	14
2.2	Sensing soil moisture at the edge. The soil moisture data is directly sent to the mobile device. . . . .	17
2.3	Summary scenario for monitoring soil moisture in different parts of a large farm, processing it on edge devices and sending it directly to the farmer's mobile phone. . . . .	18
2.4	An illustration of rice storage in paddy bags. The storage is subdivided into islands, making it easy to measure and monitor humidity manually. . . . .	20

2.5	Example Node-RED low-code development environment featuring an application composed of connected nodes. The application reads and displays humidity data and generates alerts based on set conditions. The application reads and displays humidity data and generates notifications based on a threshold value. . . . .	27
2.6	Application flow graph showing components, their names, ports and connections. The colouring on the component ports and connections shows the type of data the ports emit or accept. The arrows on the connections show the direction in which data flows through the flow graph. . . . .	28
3.1	Architectural overview of DisCoPar [Zam18]. The architecture features three parts: (1) the mobile clients, (2) the server and (3) the web client (dashboard). The lightning strikes show intermittent network connections between the mobile client and the server. . . . .	36
3.2	Visual programming environment of DisCoPar showing the canvas, an application graph composed of components with different execution scopes, the component menu and emulations for mobile devices and the web dashboard. . . . .	39
3.3	Menu options available when right-clicking a component on the DisCoPar canvas. . . . .	41
3.4	The configuration window for the <i>DisplayOnScreen</i> component. The label to display is specified under the “Heading” section. . . . .	41
3.5	Port typing and highlighting compatible component ports in DisCoPar. This example highlights all ports that accept numeric data with the electric plug symbol. . . . .	43
3.6	Supported data types in DisCoPar. The supported data types are distinguished by colour, i.e., each data type has a distinct colour. . . . .	43
3.7	Graph validation error message examples in DisCoPar. In this example, the error messages show the components that are not connected. . . . .	44
3.8	Implementation of an application showing the use of the <i>InDatabaseBuffering</i> component to handle partial failures. .	45

3.9	An example application in DisCoPar for measuring and monitoring noise levels. All the components in this application execute on the mobile scope. . . . .	47
4.1	Architectural overview of DisCoPar-Kilimo. The architecture depicts four execution scopes. The edge scope is novel and features edge components. The mobile scope features mobile components. The server scope features server components. Lastly, the web scope features web components. .	54
4.2	Component scopes of DisCoPar-Kilimo on the visual programming environment. The scopes correspond to the architectural model depicted in Figure 4.1. . . . .	57
4.3	Example environment sensing application. . . . .	58
4.4	Application example with four connected components. . . .	59
4.5	Window for configuring the <i>record-based</i> offline accessibility component. The number of records to keep is specified as a numerical value. . . . .	62
4.6	Window for configuring the <i>memory-based</i> offline accessibility component for specifying the maximum memory for data storage. The memory size is specified in bytes. . . . .	63
4.7	Window for configuring the <i>time-based</i> offline accessibility component. The configuration specifies the maximum time to keep records in memory, i.e., the lease window. The time is specified in minutes. . . . .	63
4.8	Window for configuring the <i>hybrid-based</i> offline accessibility component. Time is specified in minutes, and the number of records to keep is numerical. . . . .	64
4.9	Configuration window for the <i>BufferData</i> component to specify the number of records to store in a buffer in memory.	65
4.10	Flow-graph for <i>in-memory</i> buffering at the edge when the network connection becomes unavailable. . . . .	65
4.11	Flow-graph for <i>on-disk</i> data buffering at the edge when the network connection becomes unavailable. . . . .	66
4.12	Flow-graph for connecting an offline accessibility component to the server side. The offline accessibility component cannot connect to another downstream offline accessibility or mobile component successively. . . . .	67

4.13	Flow chart for tracking the number of connected edge devices. The input to the flow chart is an event with details for the type of edge devices. . . . .	68
4.14	Example application for tracking connected devices. The application is composed of four mobile components. . . . .	69
4.15	Configuration window for specifying the number of edge devices accumulating soil moisture in the <i>GatherMoistureReadings</i> component. The number of devices is specified as a numerical value. . . . .	69
4.16	Flow chart for accumulating data from multiple edge devices. The input to the process is a message from each edge device. . . . .	70
4.17	Flow-graph for an example application for accumulating soil moisture data. The application is composed of one edge component and several mobile components. . . . .	70
4.18	Window for configuring the <i>UnWrapForSpecificDevice</i> component to specify the edge device for which to filter data. . . . .	71
4.19	Steps for creating an application in DisCoPar-Kilimo. (a) Accessing DisCoPar-Kilimo platform, (b) Creating an application, (c) Specifying general application details, (d) Accessing the application builder VPE and (e) Complete application flow-graph on the VPE. . . . .	72
4.20	Deployed edge devices for a DisCoPar-Kilimo application for monitoring soil moisture. . . . .	73
4.21	Example flow-graph of an application for monitoring soil moisture in DisCoPar-Kilimo. The application is composed of one edge component and four mobile components. . . . .	74
4.22	Exporting the mobile and edge application flow graphs in DisCoPar-Kilimo. . . . .	75
5.1	Simple application showing a component named <i>TransmitData</i> that transmits data received on its output port. . . . .	78
5.2	Application design interface of DisCoPar-Kilimo showing the VPE (canvas and component menu). The VPE also shows the emulation for mobile applications. . . . .	102
5.3	Flow chart for building the entire application graph in DisCoPar-Kilimo. . . . .	102

5.4	Process for extracting the edge application graph. . . . .	105
6.1	Flow-graph of the application for monitoring soil moisture. . . . .	111
6.2	Preview of the application for monitoring soil moisture de- ployed to crops in a greenhouse. . . . .	112
6.3	Flow-graph of an application for computing average soil moisture and keeping data at the edge. The application is composed of three edge components and two mobile com- ponents. . . . .	113
6.4	Flow-graph of an application for monitoring soil moisture using more than one edge device. . . . .	114
6.5	Preview of the application for monitoring soil moisture us- ing more than one edge device. . . . .	115
6.6	Flow-graph of an application for tracking general informa- tion of connected edge devices. . . . .	116
6.7	Preview of the application for tracking connected edge de- vices. . . . .	116
6.8	Flow-graph of an application for tracking and monitoring humidity levels in storage areas for paddy rice. . . . .	118
6.9	Configuring the <i>SetThreshold</i> component to specify the time interval for sending the set value on the output port. . . . .	119
6.10	Mobile application preview for tracking humidity levels in storage areas for paddy rice. Humidity is below the 75% threshold, and no alert is generated. . . . .	120
6.11	Dashboard application preview for tracking and monitoring humidity levels in storage areas for paddy rice. . . . .	120
6.12	Flow-graph of an application for tracking application data. . . . .	121
6.13	Preview for tracking application data. The application fea- tures a survey for data entry and a web dashboard for data display. . . . .	122
6.14	Flow-graph of an application for monitoring soil moisture and temperature in corn seeding and sprouting. . . . .	124
6.15	Application preview for monitoring farm conditions in corn seeding and sprouting. The preview shows the maximum, minimum and average soil moisture. . . . .	125



# List of Tables

- 2.1 Summary state-of-the-art of SAAs concerning the four identified properties. HPFM refers to handling partial failures on the mobile, while HPFE refers to handling partial failures at the edge. . . . . 24
- 2.2 Summary of LCDES concerning the properties identified in Section 2.3. HPFM refers to handling partial failures on the mobile, and HPFE refers to handling partial failures at the edge. . . . . 30
- 4.1 Summary of edge and mobile components of DisCoPar-Kilimo. EDGE refers to data coming from edge devices in JSON format. OBSERVATION refers to data created by data-producing processes, NUMERIC refers to numerical data, DATASET refers to a collection of related data sets, and ALL refers to any data. . . . . 56





# List of Listings

5.1	Implementation of the basic building block of applications in DisCoPar. . . . .	79
5.2	Component execution scopes in DisCoPar. It features the server, mobile and web component execution scopes. . . . .	81
5.3	Defining the edge scope. The edge scope is defined as a class that extends the <i>Client</i> class. . . . .	82
5.4	Adding the <i>edge</i> scope to the global object. This allows it to be globally accessible in DisCoPar-Kilimo. . . . .	82
5.5	Implementation of the component for reading soil moisture at the edge. . . . .	83
5.6	Example exported edge graph showing function calls. This example calls the <i>ReadSoilMoisture</i> companion function and passes its computation results to the <i>SendData</i> function. . .	84
5.7	Implementation of the edge function to interact with the network. The function calls a low-level function <i>sendDataToNetwork</i> . . . . .	84
5.8	Implementation of the companion function of the <i>ReadSoilMoisture</i> component. The function assumes the same name as the component. . . . .	85
5.9	Implementation of the friend function of the <i>ReadSoilMoisture</i> companion function. . . . .	85
5.10	Implementation of the component for keeping previous values.	86
5.11	Implementation of the <i>Subtraction</i> component that finds the difference between two numerical values. . . . .	86
5.12	Implementation of the companion function of the <i>PreviousValue</i> component. . . . .	87

5.13	Implementation of the companion function of the <i>Subtraction</i> edge component. . . . .	87
5.14	Implementation of the blue-print for the offline accessibility components. . . . .	87
5.15	Implementation of the record-based offline accessibility component. . . . .	88
5.16	Implementation of the memory-based offline accessibility component. . . . .	89
5.17	Implementation of the time-based offline accessibility component. . . . .	90
5.18	Implementation of the hybrid offline accessibility component.	92
5.19	Implementaion of the validation to drop connections between successive offline accessibility components. . . . .	93
5.20	Implementation for validating flow-graphs that connect successive offline accessibility components. . . . .	94
5.21	Implementation of the on-disk buffering edge component. . . . .	94
5.22	Companion edge function for the <i>BufferOnDisk</i> component.	94
5.23	<i>Friend function</i> of the on-disk storage companion function.	95
5.24	Implementation for tracking connected devices on the minimalist server. . . . .	96
5.25	Implementation of the <i>ConnectedDevices</i> component. . . . .	97
5.26	Implementation of the <i>DataArrayToTable</i> component. . . . .	98
5.27	Implementation of the <i>DeviceAccumulator</i> component. . . . .	99
5.28	Format of data coming from the edge. . . . .	99
5.29	Filtering general payload of data coming from edge devices.	100
5.30	Filtering payload of specific edge devices. . . . .	101
5.31	Implementation for building the edge graph. . . . .	103
A.1	JSON representation of an application flow-graph in DisCoPar. . . . .	133
B.1	Library of companion edge functions that are invoked when edge components execute. . . . .	137
C.1	Extracting and exporting the edge graph. . . . .	139
D.1	Communication bridge running on the mobile phone . . . . .	143

# List of Abbreviations

<b>AI:</b>	Artificial intelligence.
<b>API:</b>	Application programming interface.
<b>CSS:</b>	Cascading style sheets.
<b>CSV:</b>	Comma separated values.
<b>DAG:</b>	Directed acyclic graph.
<b>DOM:</b>	Document object model.
<b>DSL:</b>	Domain-specific language.
<b>GPS:</b>	Global positioning system.
<b>GUIs:</b>	Graphical user interfaces.
<b>HTML:</b>	Hyper text markup language.
<b>ICT:</b>	Information and communication technology.
<b>IoT:</b>	Internet of Things.
<b>JSON:</b>	JavaScript object notation.
<b>LCDEs:</b>	Low-code development environments.
<b>SAAs:</b>	Smart agriculture applications.
<b>UIs:</b>	User interfaces.
<b>VPE:</b>	Visual programming environment.
<b>VPL:</b>	Visual programming language.



# Chapter 1

## Introduction

Globally, there is an increasing demand for food and nutrition security [AAUS<sup>+</sup>19, MLJ<sup>+</sup>20, BCIR22, SGR22]. Agriculture is one of the most important pillars of national income in developing countries [SBK<sup>+</sup>21]. For example, in Tanzania and Zambia, 75% of the total population derives livelihoods from agriculture [MAG16]. The increase in demand for food can be addressed through improving and optimising farming processes using information and communication technologies (ICTs).

Smart agriculture refers to the application of ICTs such as big data analysis, the Internet of Things (IoT), artificial intelligence (AI), remote sensing, robotics, blockchain and smart sensors in modern farming activities [WFHB17, GBP<sup>+</sup>23a]. Each of the above technologies serves different roles in smart agriculture. For example, AI techniques such as machine learning can be used to predict the prevailing farm conditions in smart agriculture [JHKS23, FSW<sup>+</sup>23]. Blockchain can be used to develop more traceable agri-food supply chains [CLDAOPMPA20]. Lastly, IoT can enable data sharing in a farm using embedded devices with networking technologies, e.g., Bluetooth and wireless sensor networks [CLDAOPMPA20, STM22].

The technologies mentioned above for modern farming can be used to drive farming activities in different ways, as shown in Figure 1.1. In what follows, we explain examples of applications of the above technologies in modern farming.

*Autopilot tractors and sprayer machines.* Modern tractors and sprayer machines that are equipped with a global positioning system (GPS)

can accurately drive themselves through farm fields without drivers as illustrated in Figure 1.1a [RJK<sup>+</sup>17]. The GPS can be tied to the tractor’s steering wheel to automatically keep the machines on track and free the operators from driving them. Automated guidance is essential for tillage because it eliminates human error from overlap, saving fuel and equipment usage hours.

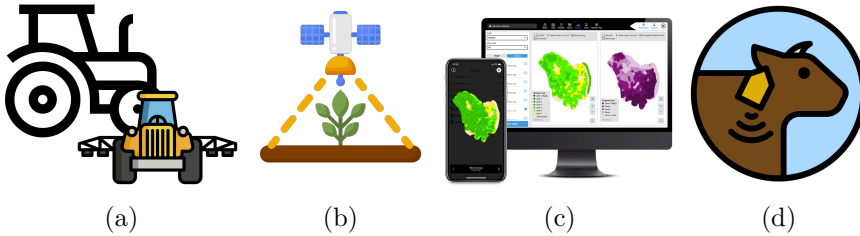


Figure 1.1: An example illustration of different modern farming technologies<sup>1</sup>. (a) Auto-pilot tractors and sprayer machines, (b) Crop sensing, (c) Documenting crop fields and (d) Monitoring animal welfare.

*Crop sensing.* Crop sensors, as illustrated in Figure 1.1b, can help farmers effectively apply fertiliser to maximise its uptake [RJK<sup>+</sup>17]. Additionally, crop sensing can help reduce the potential for leaching and runoff into groundwater [LMBS23]. Crop sensors can inform fertiliser-applying equipment when and how much to apply in real time. Optical sensors can show how much fertiliser a plant may need based on the light reflected in the sensor. Soil moisture sensors can help farmers activate or deactivate irrigation systems with a button on their smartphones [RJK<sup>+</sup>17]. This can increase flexibility and allow farmers to precisely control water resources and other inputs, such as fertiliser, applied during irrigation.

*Documenting crop fields.* GPS has also improved and simplified the process of documenting crop yields and farm resources as illustrated in Figure 1.1c e.g., water and fertiliser [LBJ01, RJK<sup>+</sup>17]. For instance, harvesting equipment can use GPS coordinates to move through crop fields, calculating yields per field to make yield maps. The yield maps allow the farmer to determine which crop varieties produced the best, worst, or most consistent yields over varying conditions.

---

<sup>1</sup>\*Image source: <https://www.flaticon.com>

*Monitoring animal welfare.* Sensors can help farmers to effectively and efficiently monitor the welfare of farm animals, e.g., when the animals require care from an expert [CJS<sup>+</sup>17, CSJK17]. The sensors can be attached to the animals as illustrated in Figure 1.1d to monitor their behaviour and trigger alerts when abnormal behaviour is observed, e.g., long periods of inactivity.

## 1.1 Smart Agriculture Applications

Smart agriculture applications orchestrate components deployed in microcontrollers, smartphones, and cloud services. For example, Figure 1.2 depicts a farmer using a mobile phone to receive data about a cornfield. In particular, soil moisture data is gathered and sent by a microcontroller installed in a cornfield. The microcontroller interprets signals from an attached soil moisture sensor and sends them to the farmer’s mobile phone. The farmer can also receive additional data from other sensors deployed to the farm, e.g., location data from a GPS sensor deployed to a tractor moving around the farm.

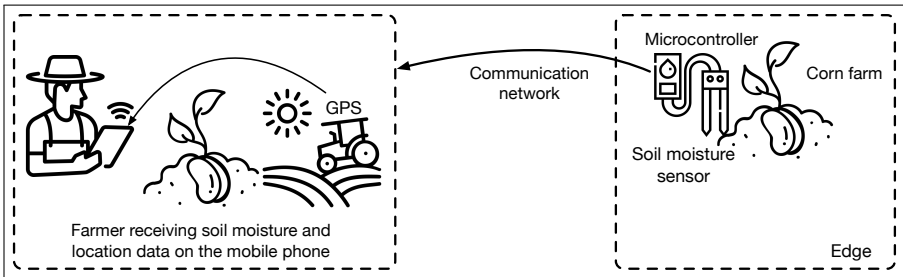


Figure 1.2: An example application that shows a farmer using a mobile phone to receive data about a corn field. The data comes from sensors attached to a microcontroller deployed to the corn field.

### 1.1.1 Developed vs Developing Regions

This dissertation focuses on the development of smart agriculture applications. In Section 2.2, we derive properties to support the development of these applications in developing countries. Technologies supporting smart agriculture applications have already been used in developed countries. In



what follows, we compare smart agriculture processes for developed and developing countries to sketch the research context.

Compared to developing countries, developed countries adopted smart agriculture technologies early and applied them in modern farms' food production activities. For example, sensing applications have been used to collect data for monitoring plants and animals [BSS<sup>+</sup>15, GBP<sup>+</sup>23b]. The data can be processed on the working machines (e.g., tractor) before sending it to other components. However, developed countries still face challenges when using the adopted technologies for smart agriculture applications. For example, 80% of the 24 million households in the USA do not have reliable, affordable, and high-speed broadband connection in rural areas [BSZ<sup>+</sup>21]. Moreover, we observe that farmers in rural areas (e.g., Germany and the USA) are still experiencing unreliable network connections [BSZ<sup>+</sup>21]. With unreliable network connections, many farmers (e.g., in Germany and the USA) opt to gather data on mobile devices while in the crop fields and upload it to the cloud after they return home or the office where they can access good connectivity [KKE<sup>+</sup>19, BSZ<sup>+</sup>21]. Unlike farmers in developing regions, farmers in developed countries have either a technical or university-level education [KKE<sup>+</sup>19].

Farmers in developing countries face more issues that hinder them from adopting smart agriculture applications. First, farmers must pay high servicing costs to connect mobile phones to communication networks [WS15, MAGT18, ETDS20]. Moreover, farmers encounter bureaucracy from service providers and intermediaries [MAGT18, RMBCC23]. Like developed countries, beyond the urban centres in developing countries, the infrastructure that can support using modern farming technologies needs to be improved [MMR<sup>+</sup>21]. For example, remote areas in developing countries experience network coverage that is limited or nonexistent [NU16, MAGT18, BBF<sup>+</sup>19, OLO19, ETDS20]. Therefore, this limits the usage of smart agriculture applications that rely on access to cloud services and that assume reliable network access.

### 1.1.2 Implementing Smart Agriculture Applications

Designing and implementing smart agriculture applications is difficult and costly in developed or developing countries [CJS<sup>+</sup>17, BBKR23]. These applications require highly skilled engineers to implement them since they entail carefully handling distribution to enable the different parts that

compose them to communicate [RSS<sup>+</sup>23]. Moreover, these are complex and expensive pieces of software that require a combination of different technical skills to implement [BGS20]. Skilled developers are hard to find, and domain experts often need more software development knowledge [BGS20, BWO<sup>+</sup>20, BM21, RSS<sup>+</sup>23].

This dissertation aims to empower domain experts to design and implement smart agriculture applications. In the context of this dissertation, a domain expert is an agricultural extension worker or simply an extension worker. An extension worker is an employee of the government or a private company that offers advisory services to farmers. The extension workers have either technical or university-level education. The extension workers are the target audience of our research, and we expect they can be trained to implement basic applications. To implement applications, extension workers need to be able to read and understand software applications, such as the code for simple programs. For example, the extension workers can implement data collection applications that they can give to farmers to collect data about their crop fields or animals and send it to them. As such, the extension workers build smart agriculture applications that farmers use as the end-users.

## 1.2 Problem Statement

Smart agriculture applications are often constructed using textual programming languages. Low-code development environments have risen as an alternative that domain experts outside software engineering can use to implement software applications that suit their needs [BGS20, SDRP20, Was19, RKdL<sup>+</sup>22, MP23]. The low-code development environments provide “ready-to-go” visual components that can be used to construct software applications and make software development more accessible to all technical skill levels [MLDGd23]. However, certain limitations hinder the widespread usage of low-code development environments for developing smart agriculture applications.

First, the state-of-the-art low-code development environments assume that devices installed in the environment (e.g., crop fields and farms) only gather sensor data and send it to a server or cloud. As such, they lack building blocks for specifying computations on those devices embedded in the environment. Moreover, the communication between the compo-

nents running on the embedded devices and those on mobile phones or the server must be coordinated. However, the state-of-the-art low-code development environments lack the infrastructure and components to support the coordination. Lastly, the output of the computations done on the embedded devices must be sent to other components on mobile phones or the server over communication networks, which can fail. The existing low-code development environments have limited support for handling the failures between the components that compose smart agriculture applications, especially when the networks become unavailable.

Based on the above observations, this dissertation is guided by the following thesis statement:

*Empowering domain experts to implement smart agriculture applications requires a low-code development environment that supports environment sensing, computation at the edge, coordination with the edge and handling partial failures.*

### 1.3 Our Approach

To accomplish its vision, this dissertation introduces DisCoPar-Kilimo<sup>2</sup>, a domain-specific low-code development environment geared for smart agriculture applications. We draw our inspiration and build on the work of DisCoPar, a low-code development environment for implementing citizen science applications for participatory sensing and campaigning [Zam18]. Figure 1.3 shows a screenshot of the DisCoPar-Kilimo visual programming environment in action.

Our work, presented in this dissertation, is at the intersection of smart agriculture applications, low-code development environments, component-based software engineering, and flow-based programming. This dissertation generally focuses on extending a low-code development environment with properties for building smart agriculture applications, as summarised below.

*Environment sensing.* We devise dedicated components to support environment sensing in a low-code development environment. The

---

<sup>2</sup>Kilimo means agriculture or farming (the art or science of cultivating the ground) in Swahili.

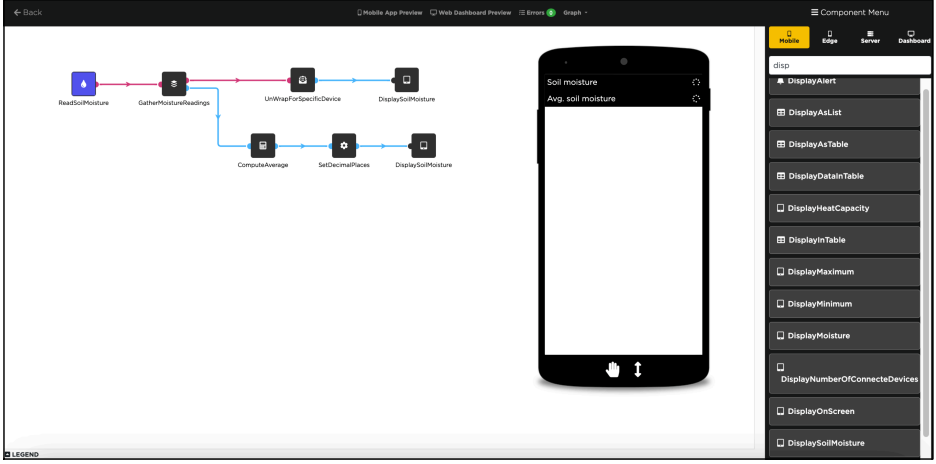


Figure 1.3: DisCoPar-Kilimo in action featuring a canvas with a smart agriculture application for sensing soil moisture. Components are drawn from the component menu, dropped on the canvas and connected to compose applications.

components are essential to monitor prevailing conditions in modern farms effectively.

*Computation at the edge.* We devise components to support computations at the edge. The edge computations can process data near the source before being sent to the farmer’s mobile phone.

*Coordination with the edge.* Conventionally, components in smart agriculture applications rely on a centralised server to communicate. We devise a mechanism for coordinating communication between components on edge devices and those on mobile devices. In addition, the mechanism ensures that tracking connected edge devices can happen. Lastly, the mechanism ensures that data from the edge devices eventually reaches the server.

*Handling partial failures when networks become unavailable.* Implementing computation at the edge introduces a point of failure when edge components communicate with mobile components. We devise mechanisms to handle partial failures when the networks become unavailable and make applications offline available and accessible.

## 1.4 Contributions

This dissertation makes the following contributions.

- As our first contribution, the dissertation proposes a set of properties (cf. Section 2.3) that a low-code development environment needs to support implementing smart agriculture applications. The proposed properties include environment sensing, computation at the edge, coordination with the edge, and handling partial failures when networks fail. The dissertation implements the above properties as features of a low-code development environment.
- Our second and main contribution is DisCoPar-Kilimo, a domain-specific low-code development environment for implementing smart agriculture applications. The domain-specific low-code development environment is based on *flow-based programming* where applications are represented as graphs of interconnected nodes that stream data to each other. The implementation of DisCoPar-Kilimo adds the following extensions to DisCoPar. First, DisCoPar-Kilimo extends the architecture of DisCoPar with a new execution scope that we call the *edge scope*. The edge scope hosts components for performing computations at the edge and environment sensing. Second, DisCoPar-Kilimo devises infrastructure for extracting and deploying the edge graph to edge devices. Third, DisCoPar-Kilimo introduces more policies to handle partial failures on the mobile. In addition, it introduces policies and components to handle partial failures at the edge. Lastly, DisCoPar-Kilimo devises a mechanism to allow edge components to communicate with mobile components directly.

### 1.4.1 Supporting Publications

In this section, we list the supporting publications and explain their relation to the work in this dissertation.

- [OMK<sup>+</sup>21]:- Isaac Nyabisa Oteyo, Matteo Marra, Stephen Kimani, Wolfgang De Meuter and Elisa Gonzalez Boix. A Survey on Mobile Applications for Smart Agriculture. *SN Computer Science*, 2(4):1–16, 2021. <https://doi.org/10.1007/s42979-021-00700-x>.

This publication surveys mobile applications in smart agriculture and introduces a taxonomy for classifying them. It presents pertinent software engineering issues important to smart agriculture applications, as detailed in Chapter 2.

- [OSZ<sup>+</sup>21]:- Isaac Nyabisa Oteyo, Angel Luis Scull Pupo, Jesse Zaman, Stephen Kimani, Wolfgang De Meuter and Elisa Gonzalez Boix. Building Smart Agriculture Applications Using Low Code Development Tools: The Case for DisCoPar. In *Proceedings of the IEEE AFRICON 2021*, pages 562–565, September 13–15, 2021, Arusha – Tanzania, 2021, IEEE. <https://doi.org/10.1109/AFRICON51333.2021.9570936>.

---

This publication identifies properties of low-code development environments that can support implementing smart agriculture applications, as we detail in Chapter 2 and Chapter 3.

- [OSZ<sup>+</sup>23]:- Isaac Nyabisa Oteyo, Angel Luis Scull Pupo, Jesse Zaman, Stephen Kimani, Wolfgang De Meuter and Elisa Gonzalez Boix (2023). Easing Construction of Smart Agriculture Applications Using Low Code Development Tools. In *Longfei, S., Bodhi, P. (eds) Mobile and Ubiquitous Systems: Computing, Networking and Services*, MobiQuitous 2022. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 492, pp 21–43. Springer, Cham. [https://doi.org/10.1007/978-3-031-34776-4\\_2](https://doi.org/10.1007/978-3-031-34776-4_2).

---

This publication introduces and presents the final version of DisCoPar-Kilimo, the technical contribution of this dissertation as we detail in Chapter 4 and Chapter 5. In addition, it introduces the extensions and contributions made to DisCoPar to yield DisCoPar-Kilimo. It also defines the first iteration of smart agriculture scenarios presented in Chapter 6. Lastly, the paper implements the defined scenarios in DisCoPar-Kilimo.

In addition to the publications highlighted above, we also published

one journal article and three conference articles that helped shape the ideas behind this dissertation [OKB<sup>+</sup>18, OT20, OKZ<sup>+</sup>20, GKOK23].

## 1.5 Dissertation Outline

The rest of this dissertation is organised as follows.

### **Chapter 2: State-of-the-Art of Smart Agriculture Applications.**

This chapter presents a state-of-the-art of smart agriculture applications and low-code development environments. The chapter starts by describing key farming processes for smart agriculture that are important to this dissertation. A description of representative driver scenarios for smart agriculture applications follows this. The chapter identifies essential properties for smart agriculture applications based on those scenarios. Then, it uses the identified properties to analyse the state-of-the-art of different smart agriculture applications. The chapter uses the identified properties to identify shortcomings in state-of-the-art domain-specific low-code development environments.

**Chapter 3: DisCoPar.** This chapter describes DisCoPar, a low-code development environment where we will prototype our contributions. The chapter presents the architectural overview of DisCoPar and describes the different layers comprising DisCoPar. Afterwards, the chapter describes the relevant components and how they are connected to form an application and presents how applications in DisCoPar are validated for errors. Then, it presents a sample application implemented in DisCoPar. Lastly, it concludes by highlighting the limitations of DisCoPar that are addressed in this dissertation.

**Chapter 4: DisCoPar-Kilimo.** This chapter describes the design of components for proofing the concepts we propose in this dissertation and how they were introduced in DisCoPar-Kilimo. First, the chapter describes the critical features of DisCoPar-Kilimo. Then, the chapter presents the architectural overview of DisCoPar-Kilimo and how it supports the properties identified in Chapter 2 for smart agriculture applications. The chapter then describes how to design and deploy smart agriculture applications using DisCoPar-Kilimo.

Lastly, the chapter concludes by describing the extensions and contributions that DisCoPar-Kilimo makes to DisCoPar.

**Chapter 5: DisCoPar-Kilimo Implementation.** This chapter presents the implementation of DisCoPar-Kilimo and the challenges faced while implementing it. First, the chapter presents the basic building blocks relevant to implementing DisCoPar-Kilimo. Second, the chapter details the implementation of the architecture and features of DisCoPar-Kilimo. Lastly, it describes how to design and deploy applications using DisCoPar-Kilimo.

**Chapter 6: Validation of DisCoPar-Kilimo.** This chapter presents the validation for DisCoPar-Kilimo. For this dissertation, we adopt a scenario-based validation approach. The chapter implements all the driver scenarios for smart agriculture presented in Chapter 2 in DisCoPar-Kilimo. Lastly, the chapter concludes by discussing the implemented scenarios.

**Chapter 7: Conclusion and future work.** This chapter summarises the dissertation, revisits the research problem and approach, provides an overview of the contributions of this dissertation, discusses its shortcomings, and gives some directions for future work.





## Chapter 2

# State of the Art of Smart Agriculture Applications

Different technologies have been developed over time to facilitate the work of farmers, optimise the use of resources in farms, and increase farm yields. Today, the agricultural sector is accelerating its transformation by adopting and using connected devices (e.g., sensors) in the farm, giving rise to smart agriculture. As already explained in Chapter 1, smart agriculture refers to the incorporation of ICTs into modern farming processes for improved management of farm activities, such as monitoring farm conditions [NPSO96, CJS<sup>+</sup>17, PWT<sup>+</sup>18, BSDN19].

This chapter presents the state-of-the-art smart agriculture applications (SAAs) and low-code development environments (LCDEs). The chapter begins by highlighting the critical processes in smart agriculture that form the basis for SAAs. The chapter then presents seven driver scenarios that serve as the basis for distilling four properties a low-code development environment should have for building SAAs. Lastly, the chapter performs a state-of-the-art analysis of SAAs and LCDEs concerning those properties.

### 2.1 Smart Agriculture Processes

As mentioned in Section 1.1, SAAs orchestrate components deployed to microcontrollers, smartphones and cloud services. These components improve operations in modern farming, such as collecting and analysing data,

## CHAPTER 2. STATE OF THE ART OF SMART AGRICULTURE APPLICATIONS

---

planning, monitoring and controlling [RFMM20]. Therefore, smart agriculture is sustained by key processes that can be summarised into four categories as identified by Wolfert *et al.* [WGVVB17], namely:

1. Data collection.
2. Data transformation and processing.
3. Data dissemination.
4. Evaluation and impact assessment.

Figure 2.1 illustrates the above smart agriculture processes. We describe each of these processes below.

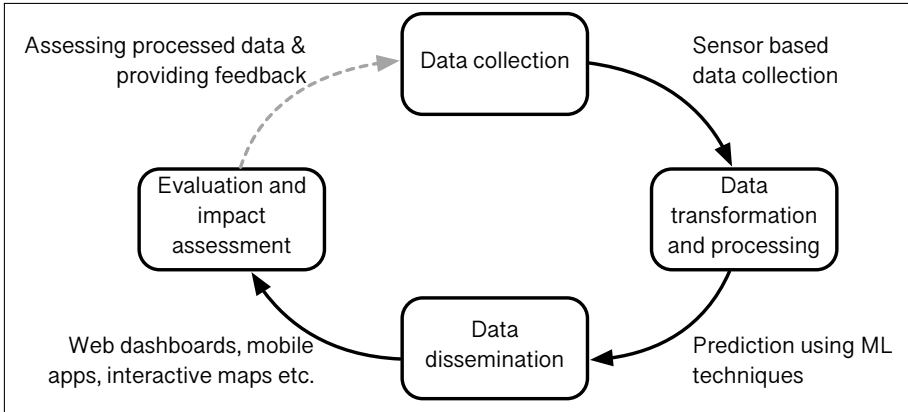


Figure 2.1: Illustration of key smart agriculture processes grouped into four key areas. The processes are data collection, transformation and processing, dissemination, evaluation and impact assessment.

*Data collection* refers to the process of acquiring data from the environment (e.g., farms, crop fields etc.) [WGVVB17]. The data collection process can be automated or manual. Sensors or embedded devices can be used for automated data collection, e.g., motion cameras to capture crop images or temperature and humidity sensors to monitor farm conditions. Farmers often collect data manually by entering data directly via web forms or dedicated applications [JYG<sup>+</sup>16].

*Data transformation and processing* refers to converting collected data into meaningful information for farmers to use in making appropriate decisions. For example, accelerometer values collected from

wearable devices can be converted to determine animal location and movements. The transformation and processing may benefit from statistical tools available on data analytic platforms. As mentioned in Chapter 1, machine learning can also be used to process the data for prediction.

*Data dissemination* serves to communicate the transformed and processed data (i.e., information) meaningfully to the farmer. Information is communicated to farmers to help them make informed decisions. The data dissemination involves web dashboards, mobile applications, and interactive maps to present processed data to farmers. For example, interactive maps can be used to communicate yield information like heat maps.

*Evaluation and impact assessment* process analyses the transformed data to give feedback to the data collection process. Farmers in developing regions rely on agricultural extension services to implement technologies that can be used to modernise smart agriculture processes. Agricultural extension workers offer the agricultural extension services [DAEA18, KAMH21]. They facilitate and support people engaged in agricultural activities to solve problems, obtain information and provide advisory feedback to farmers and farming communities [DBR20, EEE21]. Hence, the extension workers are deemed the primary contact persons for making agriculture and its related activities more effective in developing regions [DAEA18].

## 2.2 Driver Scenarios for Smart Agriculture

As mentioned in Section 1.1, SAAs are varied and support different processes in agriculture as described in Section 2.1. This section introduces seven driver scenarios used throughout this dissertation to distil key properties for SAAs.

### 2.2.1 Sensing Farm Conditions

Consider the domain of smart farming, where farmers and extension workers aim to obtain optimal yields, i.e., increased crop production based on the prevailing farm conditions. To accomplish this goal, the farmers collect data using sensor devices installed on the farm. Farmers can take

## CHAPTER 2. STATE OF THE ART OF SMART AGRICULTURE APPLICATIONS

---

different actions depending on the farm size. We assume that the same crop (e.g., corn) is grown in all the driver scenarios. In this setting, the parameters of interest are soil moisture and temperature, as they are vital environmental conditions that influence the sprouting of seeds [ZPP<sup>+</sup>20].

The soil moisture indicates the available water content to support the sprouting of seeds. For example, inadequate soil moisture and extremely low or high temperatures negatively affect seed sprouting. On the other hand, temperature affects the maintenance and movement of water and gases in the soil. It regularly corresponds with essential crop growth functions such as root formation and photosynthesis. Temperature anomalies significantly affect the movement of water and gas in soils. Soil moisture is integral to water cycles in land (terrestrial) ecosystems and the basis for crop survival. Therefore, these two conditions are relevant for obtaining optimal yields. For example, when soil moisture decreases and the land surface temperature increases, dryness becomes more severe, making it harder for crops to grow and develop.

In the context of smart agriculture, we describe seven scenarios.

*Scenario 1: Monitoring soil moisture.* This scenario aims to allow farmers to measure and monitor soil moisture. The scenario considers a farm that is small in physical size, e.g., a research garden or greenhouse in a university environment. For example, a farmer can have a small greenhouse next to the house. Given that the greenhouse is next to the farmer’s house, the scenario assumes that the farmer is always present or close to the farm as depicted in Figure 2.2. Inside the small greenhouse, the farmer has a water tap that is manually operated (i.e., closed and opened) to water the crops based on the conditions of the soil moisture.

From a technological point of view, the scenario considers that the farmer has a smart agriculture application with a reliable internet connection. Only one device (e.g., M5StickC) with a soil moisture sensor attached is sufficient to collect soil moisture data for the entire greenhouse. Since the farmer has a reliable network connection, access and connection to the sensors used for collecting soil moisture data is always guaranteed. The data is sent immediately and directly to the farmer’s mobile phone without any computations performed on it.

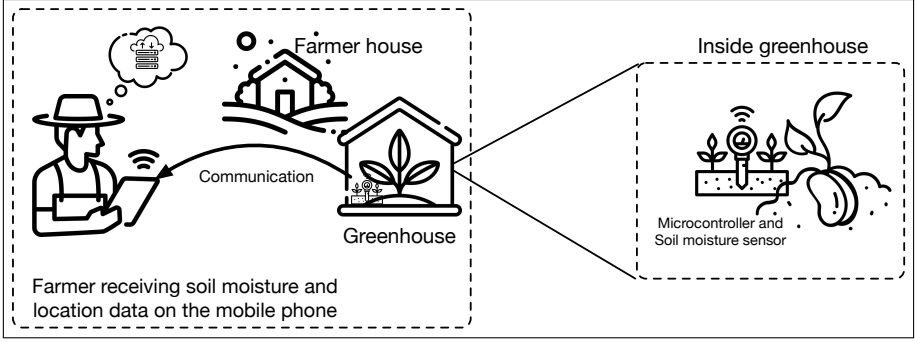


Figure 2.2: Sensing soil moisture at the edge. The soil moisture data is directly sent to the mobile device.

*Scenario 2: Computing average soil moisture and keeping data at the edge.* In this scenario, the farmer still wants to monitor the soil moisture. However, in this scenario, we assume the farmer can be absent from the farm. Therefore, the soil moisture sensor is only sometimes connected to the farmer’s mobile phone. Hence, the farmer must wait to receive soil moisture data from the greenhouse garden on the mobile phone. As such, the scenario assumes that the farmer visits the greenhouse garden once daily (e.g., early morning) to check the soil moisture and water the crops. For this to happen, the farmer needs to know the soil moisture levels when away from the garden.

From a technological point of view, the soil moisture data is collected by soil moisture sensors attached to a microcontroller. In this scenario, a single microcontroller with a sensor attached is sufficient to measure and monitor the soil moisture conditions. The soil moisture sensor continuously collects data even when the farmer is away from the farm. When the farmer is away, the device at the farm and the companion mobile application running on the farmer’s mobile phone are not connected to the network. To avoid data loss between the device at the farm and the mobile phone, data can be stored on the devices deployed to the greenhouse garden. However, the devices deployed to the farm are often resource-constrained and storing every data value from the sensors can be impractical.

*Scenario 3: Monitoring soil moisture using more than one edge device.*

## CHAPTER 2. STATE OF THE ART OF SMART AGRICULTURE APPLICATIONS

Similar to the previous two scenarios, the goal of this scenario is to monitor soil moisture, albeit for a large farm. The farmer must monitor the soil moisture and ensure sufficient water is available to irrigate the crops. Some parts of the farm may already have enough soil moisture content and may wait to require irrigation immediately and vice versa. From a technological point of view, more than one device is required to collect the soil moisture data for a large farm because conditions can vary rapidly between two locations within the farm. Hence, multiple devices are required to collect the soil moisture data. To monitor the soil moisture, the large farm can be subdivided into smaller portions and soil moisture sensors deployed to each portion as illustrated in Figure 2.3.

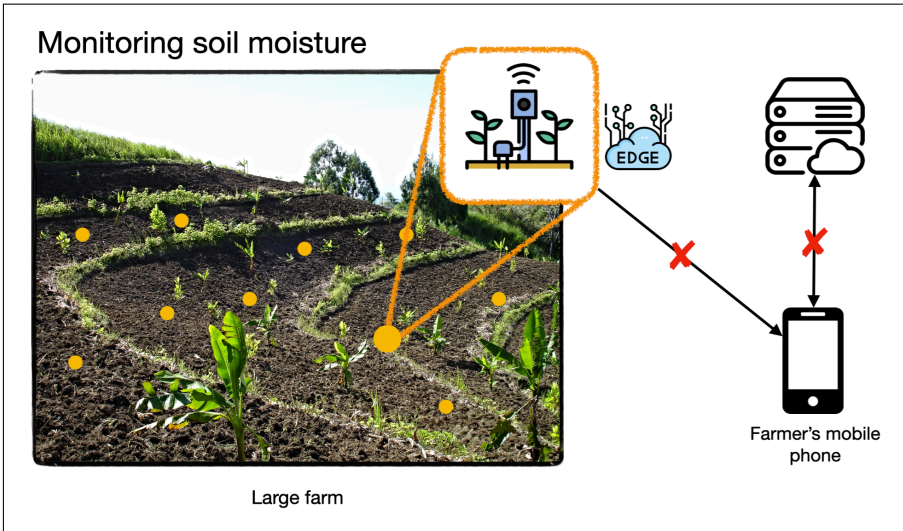


Figure 2.3: Summary scenario for monitoring soil moisture in different parts of a large farm, processing it on edge devices and sending it directly to the farmer's mobile phone.

We consider the smaller portions of the farm as the edge and the devices deployed to each portion as *edge devices*. Data from soil moisture sensors is processed on edge devices before being sent directly to the farmer's mobile phone. From the mobile phone, the data can be sent to the server. The network connections can fail while sending the data from the devices installed on the farm to the mobile device and when sending the data from the mobile device to

the server.

Similar to the second driver scenario, the farmer visits the farm once a day. Immediately after the farmer visits the farm, data from all the installed devices must be accumulated on the farmer’s mobile phone. Additionally, the data accumulated on the farmer’s mobile phone needs to be processed to enable the farmer to get meaningful information for decision-making. Accumulating data from different devices installed on the farmer is a complex undertaking.

*Scenario 4: Tracking connected edge devices.* This scenario assumes a similar setting to scenario three; its goal is to track connected devices deployed to the farm. The scenario considers a large farm with many sensors installed as depicted in Figure 2.3. The farmer must track the deployed devices to know when they go off and schedule device maintenance activities or replacements. Additionally, the farmer can track soil moisture data coming from specific devices and use the data to plan for irrigation activities on the farm.

Technologically, the sensing devices can go off because of battery drain or mechanical malfunctioning. On each visit to the farm, the farmer needs to know the number of active devices and those that are not active. The scenario considers vital information, such as the network status for each device, which the farmer can use to determine when the devices go off. For example, all the devices installed on the farm for data collection can be listed on the farmer’s mobile phone each time the farmer visits the farm.

In addition to the above four scenarios, we consider two scenarios from the literature and one scenario that is based on our experiences with agricultural extension workers in Kenya.

*Scenario 5: Tracking and monitoring paddy rice storage conditions.* This scenario is derived from existing literature [SNN18]. The farmer’s goal in this scenario is to monitor humidity in rice storage areas. Paddy rice should be stored in ideal conditions to prevent grain quality from deteriorating after harvesting. Under natural conditions, stored rice grains undergo chemical changes within themselves. The chemical changes are significantly related to the environmental conditions (e.g., the relative humidity). Regardless of the methods



## CHAPTER 2. STATE OF THE ART OF SMART AGRICULTURE APPLICATIONS

of storage, (1) the rice grains must be kept dry, and (2) the grains must be protected from adverse environmental conditions and pests [AKKM19].

The moisture content of rice grains is closely related to the relative humidity. Hence, ambient relative humidity significantly influences the quality of stored rice. Additionally, the relative humidity influences the increase or decrease of pest population that affects stored rice. For example, when the relative humidity increases to 70%, pests' multiplication increases. A low moisture content with low humidity can significantly reduce pest infestation. Figure 2.4 shows rice storage in paddy bags stacked on each other. The farmer has to monitor the relative humidity manually using a humidity meter.



Figure 2.4: An illustration of rice storage in paddy bags. The storage is subdivided into islands, making it easy to measure and monitor humidity manually.

Technologically, humidity sensing devices can be deployed to the rice bags. The devices can measure the humidity levels and send the data to the farmer's mobile phone. Since the storage areas can be enormous, they can be divided into portions and humidity sensors can be deployed to each portion. This can improve the manual process of measuring the humidity levels.

*Scenario 6: Collecting data using mobile applications.* This scenario aims to help farmers collect crop data using mobile applications. From our interaction with agricultural extension workers in Kenya,

we observed that they often collect plant-specific data to monitor their growth and development. For instance, at the sprouting stage, the extension workers can collect data on the number of leaves per plant, colour, and height. They use this data to determine if the plant is developing well and detect any abnormality that may require intervention. Counting the organs of a plant, such as leaves, is important in estimating relevant traits in crop breeding [AUS20]. Technologically, this scenario focuses on using mobile forms to help collect and track crop data [OKW<sup>+</sup>19]. Additionally, the scenario relies on plant labels to identify individual plants in the field. The scenario assumes the plant labels are QR codes generated using an external application.

*Scenario 7: Monitoring soil moisture and temperature in corn seeding and sprouting.* This scenario is derived from El-Sanatawy et al. [ESEKA<sup>+</sup>21], Sudozai et al. [STCR13] and our interactions with agricultural extension workers in Kenya. The goal is to help farmers obtain the maximum yield from corn crops. This depends on several factors. First, for the corn plants to reach maturity, the crop must develop ‘well’ during all the ‘cultivation’ phases, i.e., planting, sprouting, developing and maturing. Specifically, during the stage that encompasses seeding and sprouting of corn, maximum yield depends on achieving ‘optimal’ sprouting. To this end, extension workers must help farmers keep track of vital environmental conditions (e.g., soil moisture, temperature, etc.) that influence the sprouting of corn seeds [ZPP<sup>+</sup>20].

Adequate soil moisture content helps activate metabolic enzymes, increasing cell division and proliferation. For example, corn seeds that are sown in 60 – 65% available average soil moisture have a sprouting rate of 77.36%, while those sown in 55 – 60% available average soil moisture have a sprouting rate of 78.34% after seven days of planting. After 15 days of seed planting, a decrease in the available average soil moisture (50 – 55%) significantly reduces the sprouting of corn seeds to 62.45% [STCR13, ESEKA<sup>+</sup>21]. Besides soil moisture, temperatures below 10° Celsius retards corn growth, while temperatures above 38° Celsius affect the corn yield by stimulating pollen development. Pollen is essential for corn fertilisation [HP15]. However, high temperatures cannot severely affect corn pol-

lination if adequate soil moisture is available [HD18].

Finally, altitude influences the number of days from planting to flowering and maturity of corn because the rate of development of corn is affected by the air temperature [STCR13]. The higher the temperatures, the faster corn can grow and develop. Conversely, lower air temperature retards the development of corn and extends the time taken to reach flowering and maturity. The time from planting to emergence is highest (26 – 28 days) when corn is produced during cooler conditions. Metrics such as soil heat capacity can be used to correlate the available soil moisture and temperature [ZPP<sup>+</sup>20].

All the above scenarios point to four key properties of SAAs that we describe in the subsequent section.

### 2.3 Properties for Smart Agriculture Applications

Based on the scenarios presented in Section 2.2 and our experiences and interviews with extension workers in Kenya, we have identified four core properties inherent to SAAs, which we detail in what follows.

*Environment sensing ( $P_1$ ).* SAAs must support data collection on the prevailing crop farm conditions, such as soil moisture and temperature. Small farms (e.g., micro gardens) require only one device, while large farms require multiple devices. Data from multiple devices can be accumulated before further processing on the mobile phone or the server. Each data value can be mapped to the device from which it originates. For all the installed devices, it is essential to track general information to determine when the devices are inactive and plan for maintenance.

*Computation at the edge ( $P_2$ ).* Since extension workers and farmers in developing countries may have limited or intermittent access to cloud services, it is essential to perform computations on the devices installed at the farm (i.e., devices at the edge) to collect data instead of relying on a centralised server, which may be inaccessible due to network connection issues or cost, as the farmers cannot afford

it. Hence, the data collected should be transformed near the source before sending it to the farmer’s mobile phone.

*Coordination with the edge ( $P_3$ ).* Devices at the edge hosting sensors that collect data on environmental conditions must be able to send the collected data directly to the farmer’s mobile phone. The farmer is mobile and may not always be available at the farm. The mobile phone is closer to the devices at the edge when the farmer is available at the farm and far away from those devices when the farmer is unavailable. Therefore, mechanisms are required to coordinate how the devices at the edge communicate with the mobile devices.

*Handling partial failures ( $P_4$ ).* SAAs are distributed across devices at the edge, mobile phones and possibly the cloud. Partial failures can occur as a result of intermittent network connections or due to the farmer’s mobility. For example, the farmer can visit the farm daily to receive the data collected. Therefore, the device gathering the soil moisture and temperature data from sensors must be capable of working offline and keeping all sensor data. The stored data must be accessible when required, e.g., when the farmer visits the farm. Handling partial failures requires SAAs to be offline, available, and accessible. Offline data availability and accessibility can happen at the edge devices hosting sensors and on the farmer’s mobile phone.

We use the above four properties to perform a state-of-the-art study of smart agriculture applications that we present in the subsequent section.

## 2.4 State-of-the-Art of Smart Agriculture Applications

In this section, we describe the state-of-the-art of SAAs for sensing. Table 2.1 provides a summary of the applications that we analysed and the properties that they support. We also include their focus area and the sensors they use. The focus area indicates the specific sub-domain within agriculture where the application is used.

Most existing applications included in the state-of-the-art analysis focus on crop farming, animal farming and sensing. In terms of sensing, most applications exploit at least one sensor. To the best of our knowl-

## CHAPTER 2. STATE OF THE ART OF SMART AGRICULTURE APPLICATIONS

edge, none of the surveyed applications supported coordination with the edge and handling partial failures at the edge.

Application	Focus area	Sensors used	P <sub>1</sub>	P <sub>2</sub>	P <sub>3</sub>	P <sub>4</sub>	
						HPFM	HPFE
PocketLAI [CFC <sup>+</sup> 13, OMC <sup>+</sup> 16]	Estimating crop water needs	Phone camera	✓	✗	✗	✗	✗
WISE [BAAB15]	Irrigation scheduling	Soil moisture	✓	✗	✗	✗	✗
EVAPO [MVD19]	Estimating evapotranspiration	Temperature & humidity	✓	✗	✗	✗	✗
PIS [ISY <sup>+</sup> 17]	Soil moisture sensing	Soil moisture	✓	✗	✗	✗	✗
RaGPS [MMJRCFP11]	Solar radiation monitoring	Light	✓	✗	✗	✗	✗
SmartIrrigation [VLA <sup>+</sup> 16]	Irrigation scheduling	Soil moisture	✓	✗	✗	✗	✗
Crop Water Stress [PWL <sup>+</sup> 19]	Vine water stress monitoring	Soil moisture	✓	✗	✗	✗	✗
pCAPS [HHRHGM <sup>+</sup> 16]	Monitoring crop water needs	Phone camera	✓	✗	✗	✗	✗
VitiCanopy [DFG <sup>+</sup> 16, FBPT12, FPEOF <sup>+</sup> 14]	Estimating crop water needs	Phone camera	✓	✗	✗	✗	✗
SmartfLAIr [BSJA16]	Crop yield estimation	Light	✓	✗	✗	✗	✗
PETEFA [PMHT18]	Geo-referenced soil analysis	GPS	✓	✗	✗	✗	✗
eFarm [YST <sup>+</sup> 17]	Geo-tagging land data	GPS	✓	✗	✗	✗	✗
PAMS [YCLF13]	Managing land spatial data	GPS	✓	✗	✗	✓	✗
cFertigUAL [PCSMC <sup>+</sup> 17]	Fertiliser estimation	Temperature & humidity	✓	✗	✗	✓	✗
BaoKhao [YS12]	Leaf colour estimation for N fertiliser	Phone camera	✓	✗	✗	✗	✗
SnapCard [FCO <sup>+</sup> 16]	Crop spraying	Phone camera	✓	✗	✗	✗	✗
MobiCrop [LCAD13, LCJD13]	Sharing information on pesticides	GPS	✓	✗	✗	✓	✗
DropLeaf [MSA <sup>+</sup> 18]	Crop health	Phone camera	✓	✗	✗	✗	✗
AgriMaps [JEM <sup>+</sup> 16]	Land management	GPS	✓	✗	✗	✗	✗
LandPKS [HBB <sup>+</sup> 16]	Soil assessment	GPS	✓	✗	✗	✗	✗
SOCt [ADCB13]	Soil assessment	GPS	✓	✗	✗	✗	✗
SIFSS [ADCB13]	Soil assessment	GPS	✓	✗	✗	✗	✗
GeoFoto [MCCGdGF12]	Land identification	GPS	✓	✗	✗	✗	✗
MapIT [FSC13]	Equipment tracking	GPS	✓	✗	✗	✗	✗
SafeDriving [LK13]	Equipment tracking	GPS	✓	✗	✗	✗	✗
SmartHof [CJS <sup>+</sup> 17]	Monitoring animal health	Accelerometer	✓	✓	✗	✗	✗
SmartFarm [CSJK17]	Monitoring animal health	Accelerometer	✓	✓	✗	✗	✗
BioLeaf [MOA <sup>+</sup> 16]	Leaf health monitoring	Phone camera	✓	✗	✗	✗	✗
Plant Disease [Pet17, Pet19]	Plant disease diagnosis	Phone camera	✓	✗	✗	✗	✗
Canopeo [PO15]	Estimating canopy development	Phone camera	✓	✗	✗	✗	✗
vitisFlower [AMG <sup>+</sup> 15]	Flower assessment	Phone camera	✓	✗	✗	✗	✗
vitisBerry [AIMP <sup>+</sup> 18]	Berry assessment	Phone camera	✓	✗	✗	✗	✗
FruitSize [WKW <sup>+</sup> 18]	Fruit size assessment	Phone camera	✓	✗	✗	✗	✗
PulAm [PMFZRCAG19]	Crop pest monitoring	–	✗	✗	✗	✓	✗
UbiQON [WS18]	Mushroom monitoring	Temperature& humidity	✓	✗	✗	✗	✗
Blynk based app [SNN18]	Paddy rice monitoring	Temperature& humidity	✓	✗	✗	✗	✗
iDee [ADCB13]	Water assessment	Phone camera	✓	✗	✗	✗	✗
ConnectedFarm [RYM <sup>+</sup> 15]	Environment monitoring	Temperature& humidity	✓	✗	✗	✗	✗
SmartFarmKit [MPT <sup>+</sup> 17]	Mushroom and maize monitoring	Temperature& humidity	✓	✗	✗	✗	✗
WheatCam [CKR19]	Risk management	Phone camera	✓	✗	✗	✗	✗
SMILEX [RSS <sup>+</sup> 10]	Tracking sick plants	GPS	✓	✗	✗	✗	✗
IRIS [MSR <sup>+</sup> 18]	Data collection	Soil moisture	✓	✗	✗	✗	✗
iFarm [MUH <sup>+</sup> 13]	Data collection	–	✓	✗	✗	✗	✗
GeoFarmer [ECA <sup>+</sup> 19]	Data collection	GPS	✓	✗	✗	✓	✗

Table 2.1: Summary state-of-the-art of SAAs concerning the four identified properties. HPFM refers to handling partial failures on the mobile, while HPFE refers to handling partial failures at the edge.

### 2.4.1 Applications that Support Sensing

As shown in Table 2.1, all the applications surveyed except one support the environment sensing property. The applications exploit various sensors such as soil moisture, phone cameras, temperature, humidity, global positioning system (GPS), accelerometer and ambient light sensors. Each application exploits one specific sensor depending on the focus area.

### 2.4.2 Applications that Support Computation at the Edge

We observe that only two applications (SmartHof [CJS<sup>+</sup>17] and SmartFarm [CSJK17]) offload part of their computations to the edge to reduce delays that can be experienced when communicating with cloud-hosted services. Both applications were designed to monitor animal welfare and follow the classic 3-tier architecture, which has three components: cloud, edge, and sensing. The cloud provides data processing and storage, while Raspberry Pi devices are used for data collection via sensors at the edge. In SmartHof, the animal carries the Raspberry Pi, which uses temperature and accelerometer sensors to collect body temperature and movement. The accelerometer data is processed on the Raspberry Pi to determine the animal’s physical location and the number of steps it has taken. The application then uses the processed data to trigger an alarm about the animal’s health when it cannot move by correlating data in the cloud processing component. The mobile application acts as an interface to manage farm configurations and evaluate animal welfare factors in the cloud. Both applications enable the farmers to visualise and interact with the farm in real time.

### 2.4.3 Applications that Support Handling Partial Failures

In Table 2.1, we show applications that exhibit handling partial failures on the mobile phone and those that exhibit handling partial failures at the edge. The few applications that handle partial failures do so on the mobile phone using local storage [PMFZRCAG19], synchronisation [MUH<sup>+</sup>13], client-side databases [YCLF13, PMFZRCAG19] and caching [LCAD13, LCJD13, PCSMC<sup>+</sup>17, ECA<sup>+</sup>19]. For instance, MobiCrop [LCAD13, LCJD13] uses caching to offer offline accessibility. GeoFarmer [ECA<sup>+</sup>19] uses local phone storage that synchronises to a central database when the application is online. iFarm’s [MUH<sup>+</sup>13] client side is synchronised to the server when the network is available to guarantee that up-to-date data is available for local computations.

PAMS [YCLF13] and cFertigUAL [PCSMC<sup>+</sup>17] use a database on the client side to store data upon network disconnection. cFertigUAL [PCSMC<sup>+</sup>17] uses a persistent database on the client side to minimise the number of requests that can be made to the server. This way, the application can function offline between client requests’ windows to the server.

PulAm [PMFZRCAG19] exploits an SQLite database that runs locally on the mobile device to minimise network connection issues. The changes that are made to the database when in offline mode are inserted into a queue. When the internet connection becomes available, the contents of the queue are sent and synchronised to the server.

### 2.4.4 Discussion

Most of the applications studied support the environment-sensing property. A varied number of sensors are used by the applications presented in Table 2.1 including the phone camera, ambient light sensor, temperature and humidity sensors, soil moisture sensor, solar radiation sensor, accelerometer and lastly, the GPS sensor. Most of the applications presented in Table 2.1 exploit the phone camera as a sensor to capture images for further analysis.

Few applications support handling partial failures only between the mobile and cloud service or server. Applications that use sensors external to the mobile phone first send the data to a server, and the mobile phone fetches that data from the server. To the best of our knowledge, none of the applications analysed offers support for (1) handling partial failures at the edge and (2) coordination with the edge. All the applications presented in Table 2.1 have been implemented using textual programming languages.

## 2.5 State-of-the-Art of Low-Code Development Environments

As mentioned in Chapter 1, designing and implementing SAAs requires a combination of skills. In this thesis, we focus on helping domain experts (such as agricultural extension workers) with limited programming experience build SAAs. The applications built by the extension workers can be used by farmers who benefit from the advisory services of the extension workers.

LCDEs can serve as an alternative that domain experts outside software engineering (such as agricultural extension workers) can use to implement software applications [BGS20, SDRP20]. The main aim of the LCDEs is to reduce the development and maintenance effort required to implement applications for digital-savvy persons with limited program-

## 2.5. STATE-OF-THE-ART OF LOW-CODE DEVELOPMENT ENVIRONMENTS

ming experience [SLPF22, RKdL<sup>+</sup>22, GS22, MP23]. This section gives an overview of LCDEs that can be used to construct applications as shown in Figure 2.5. The application in Figure 2.5 reads humidity data, displays it and generates notifications based on a threshold value.

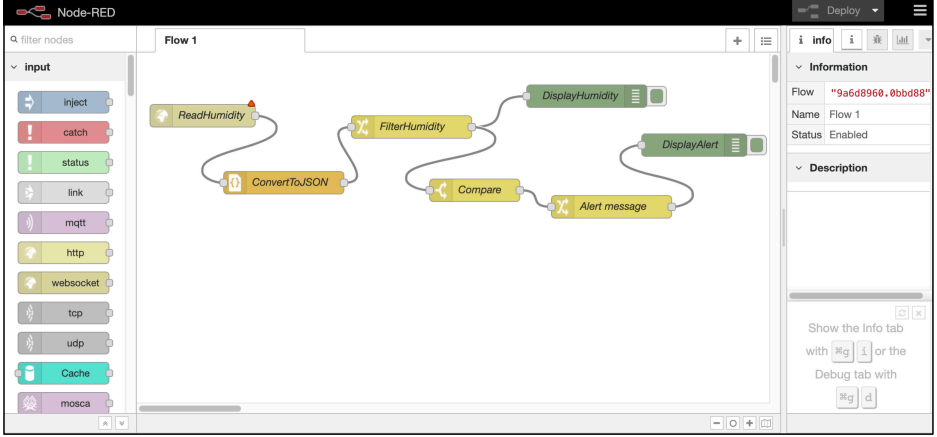


Figure 2.5: Example Node-RED low-code development environment featuring an application composed of connected nodes. The application reads and displays humidity data and generates alerts based on set conditions. The application reads and displays humidity data and generates notifications based on a threshold value.

LCDEs provide visual programming environments (VPEs) in which applications are constructed by dragging, dropping and connecting visual components that represent different computational tasks in the application [Was19, SDRP20, SDRIP23, MLDGd23]. The VPEs embody visual programming languages (VPLs). The VPLs provide pre-built (i.e., “ready-to-go”) components that the domain experts can use to construct and configure software applications. The pre-built components can make constructing software applications more intuitive for domain experts since they are already tested and ready to use [KFHB21].

Different kinds of VPLs exist in the literature, such as block-based VPLs [RMMH<sup>+</sup>09], icon-based VPLs [Cha87a, Cha87b], form-based VPLs [RSF17], and flow-based VPLs [CK02]. In this dissertation, we focus on *flow-based VPLs* because the LCDEs we consider naturally use them. In the subsequent section, we explain the notation used in flow-based VPLs and then review the existing LCDEs concerning the four properties



identified in Section 2.3.

### 2.5.1 Notation Used in Flow-Based VPLs

Flow-based VPLs use a specific notation to represent program features and compose software applications. In this section, we explain this notation.

*Application graph.* Flow-based VPLs use graphs to represent applications. In this dissertation, application graphs or flow graphs are used interchangeably. A flow graph is an acyclic network of nodes and connections describing data flow during application execution. For instance, Figure 2.6 shows an application graph composed of four nodes. Nodes C1 and C2 send data to C3, which in turn sends its output to node C4.

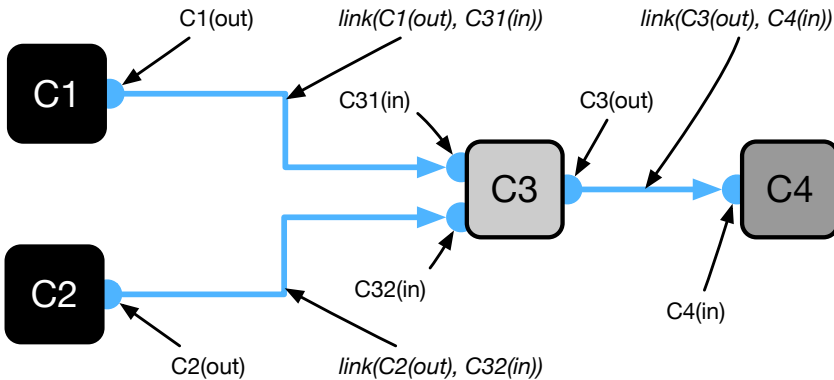


Figure 2.6: Application flow graph showing components, their names, ports and connections. The colouring on the component ports and connections shows the type of data the ports emit or accept. The arrows on the connections show the direction in which data flows through the flow graph.

*Nodes and node connections.* Flow-based VPLs use nodes to represent computation tasks that can modify data. Each node has a name representative of the computation task it performs. Nodes can have input ports, output ports or both. The input ports function as receptors for connections that supply data to the node. The output nodes send the computation results from the node to the subsequent “downstream” nodes in the application flow graph. For example, in

Figure 2.6, nodes **C1** and **C2** have only one output port each, node **C3** has two input ports and one output port and lastly, node **C4** has only one input port. In this case, **C1** and **C2** are said to be *source* nodes, **C3** a *processor* node and **C4** a *sink* node.

Nodes are joined together via links. The links serve as communication channels between the nodes. For instance, in Figure 2.6, nodes **C1** and **C3** are joined together by link,  $link(C1(out), C3_1(in))$ , while nodes **C3** and **C4** are joined together by link,  $link(C3(out), C4(in))$ . In terms of implementation, the node connections can be constructed as simplex or duplex channels. The simplex channels have only the forward channel, while the duplex channels have both a forward and backward channel. The forward channel carries information (data) and delivers it to the next component in the application flow graph. The backward channel can send acknowledgement messages to “upstream” components. This is useful for applications that require confirmatory messages as proof of successful transactions, e.g., proof that data has been saved in a database.

*Data and data types.* In LCDEs, data travels through links that supply inputs for nodes. The data is transformed into a new form for output in the nodes. *Data types* are used to specify the kind of data a node can receive on its input port(s) or emit on its output port(s). Some LCDEs allow developers to join any node to another regardless of the data being shared between the two nodes [BL14]. Hence, developers can make errors in application designs by connecting nodes that send data that is not accepted by the receiving node. Other LCDEs provide the concept of data types through the input and output ports of nodes [ZKD18, ZKD21]. Therefore, the input ports can only accept connections from *compatible output ports*. Often, the ports present physical cues to guide the application designer on compatible ports. For example, in DisCoPar, each port has a colour representing the data type it emits or accepts [ZKD18, ZKD21].

*Node compatibility.* Two successive nodes are compatible if the upstream node’s output port emits data acceptable by the downstream node’s input port. A node’s input and output ports enforce the data type that it emits or receives. Hence, two nodes *A* and *B* are compatible if the output port of node *A* is compatible with the input port of *B*. In Figure 2.6, the output ports of nodes **C1** and **C2** are compatible with

the input ports of node **C3**, thus making node  $\{\mathbf{C1}, \mathbf{C3}\}$  and  $\{\mathbf{C2}, \mathbf{C3}\}$  compatible. The node compatibility concept can be used to enforce type-checking rules, which help programmers avoid errors during application design. Therefore, the VPLs can disallow connections between nodes that are not compatible. For instance, in DisCoPar, connections between incompatible nodes are automatically dropped when composing applications [ZKD18, ZKD21].

### 2.5.2 Review of Existing Low-Code Environments

This section reviews the existing LCDEs based on the smart agriculture properties identified in Section 2.3. Table 2.2 shows a summary review of the existing LCDEs based on their support for the properties explained in Section 2.3.

Tool	$P_1$	$P_2$	$P_3$	$P_4$	
				HPFM	HPFE
Node-RED [BL14]	✓	✗	✗	✗	✗
WotKit processor [BL12]	✓	✗	✗	✗	✗
NoFlo [Nof]	✓	✗	✗	✗	✗
Apache Ni-Fi [NiF]	✓	✗	✗	✗	✗
NetLab Toolkit [Net]	✓	✗	✗	✗	✗
DDFlow [NTGS19]	✓	✗	✗	✗	✗
DisCoPar [Zam18, ZKD18, ZKD21]	✓	✗	✗	✓	✗

Table 2.2: Summary of LCDES concerning the properties identified in Section 2.3. HPFM refers to handling partial failures on the mobile, and HPFE refers to handling partial failures at the edge.

From Table 2.2, all the reviewed LCDEs offer support for environment sensing, albeit not for smart agriculture-related parameters like temperature and soil moisture. All the tools presented in Table 2.2 do not support (1) components for sensing environmental conditions in crop fields or farms, (2) computation at the edge, (3) coordination with the edge, and (4) handling partial failures at the edge. DisCoPar supports handling partial failures on mobile devices. Therefore, to use the tools in constructing smart agriculture applications, the tools need to be extended with (1) components for computation at the edge, (2) mechanism for coordinating mobile components with the edge and (3) components for handling partial

failures at the edge.

Each of the tools supports core concepts for constructing applications. To better understand each tool, we describe the core concepts for each tool in what follows.

*Node-RED*: Application programs in Node-RED are referred to as *flows* [BL14]. The *flows* consist of *nodes* connected by *wires*, i.e., links. To the application developer, Node-RED provides a VPE called a *flow canvas* for designing and deploying *flows*. The VPE consists of a flow editor with node templates that can be dragged and dropped into the flow canvas. The flows in Node-RED require a centralised server to coordinate communication between nodes. Node-RED is implemented in JavaScript using the Node.js<sup>1</sup> framework. Flows in Node-RED are saved into a flow file. Edge devices must be connected to the server hosting the Node-RED flows to deploy them to the devices. They only send data, and edge devices cannot host flows. During execution, the flow file is read, and nodes are instantiated to correspond to the node type in the flow file. On instantiation, source nodes can subscribe to external services, listen for data on their input port(s), get data from a sensor, or begin processing HTTP requests.

*Web of Things (WotKit) processor*: The WotKit processor is an IoT mashup toolkit [BL12]. Concretely, the processor allows users to process sensor data and react to real-time updates from sensors and other external systems [BL12, GBLL15]. The data model for the WotKit processor consists of sensors with fields describing the sensors connected to the system. The processor accumulates data from various sensors and allows users to find and subscribe to the sensor data of interest. The sensor data is processed as it is pushed into the system from components and visualised on a dashboard using widgets. The processor is implemented in Java using the Spring framework and leverages the Java concurrency framework in its execution engine. The WotKit processor provides a browser-based visual data-flow editor as the primary interface. Processor users can create data flow programs called *pipes*. The pipes are made up of *modules* that are connected with *wires*. Typical applications implemented using the processor are IoT mashup applications that require a centralised

---

<sup>1</sup><https://nodejs.org/en>

server to coordinate communication.

*NoFlo*: NoFlo<sup>2</sup> is a flow-based programming environment for JavaScript that runs both in the browser and Node.js. A program in NoFlo is defined as a flow graph of nodes connected via edges. The nodes react to information packets or incoming messages. When a node receives messages on its input ports, it performs a predefined operation, yielding results that it sends to other nodes on its output ports. Components communicate by sending messages or packets through a centralised server. Therefore, once components are deployed, they must send their data to the NoFlo instance via web sockets. When applications in NoFlo are executed, NoFlo creates a live graph, instantiates the components used in the graph and links them together. In the current implementations of NoFlo, the existing sensing components use sensors embedded in mobile phones.

*Apache Ni-Fi*: This is a low-code development environment built to automate data flow between systems. It supports various data formats like logs, geolocation data and social feeds. The platform provides a configurable and web-based user interface. Programs in Apache Ni-Fi<sup>3</sup> are composed of *processors* that are connected using *flows*. The *processor* is a module that fetches data from the source system or stores it in the destination system. Other *processors* can also add attributes or change content in the flow file. Constructing applications is done by adding *processors* and other components to the Ni-Fi canvas. The *processors* are configured and connected for Apache Ni-Fi to know what to do with each flow file after processing it.

*NetLab toolkit*: This is a drag-and-drop IoT-based application development environment hosted on the cloud. The NetLab<sup>4</sup> toolkit provides a web interface to connect sensors with smart widgets. Designing applications is done using widgets. The widgets represent software components for processing data. The widgets are connected to enable data to flow between them. The widgets can be configured and adjusted on how they process data. Other widgets can be used to build the graphical user interface for standalone applications.

---

<sup>2</sup><https://noflojs.org/>

<sup>3</sup><https://nifi.apache.org/>

<sup>4</sup><https://www.netlabtoolkit.org/>

*DDFlow*: This LCDE provides an environment for programming applications that span a diverse and dynamic IoT network [NTGS19]. Applications in DDFlow are defined as a sequence of nodes in a data flow graph. A node is a computational abstraction that represents a stateful function. As a stateful function, the node can optionally map inputs to outputs. Inputs and outputs are *key-value* dictionaries (i.e., JSON messages) that contain application data and metadata such as timestamps. Nodes correspond to at least one instantiation of a task that must be deployed onto a device in the network. Nodes are connected via wires representing a data flow graph connection.

*DisCoPar*: This is a visual reactive and flow-based domain-specific language was built for constructing smart sensing applications [Zam18, ZKD21]. Programs in DisCoPar are directed acyclic graphs (DAGs) where each node of the graph consists of a component instance and where the edges represent real-time data streams. DisCoPar’s component layer is written in JavaScript. As a result, the source code of a component can be deployed both on the client and server sides, as the latter relies on Node.js. We describe DisCoPar in detail in Chapter 3.

## 2.6 Conclusion

From the LCDEs presented above, (1) individual computation tasks in applications are highly conceptualised into visual components for novice developers, and (2) the tools support implementing web and mobile applications using drag-and-drop and point-and-click visual programming techniques. These techniques can be intuitive and valuable to domain experts for constructing SAAs.

In this chapter, we first identify four properties of SAAs. Secondly, we perform a state-of-the-art analysis of SAAs and LCDEs based on the identified properties. To the best of our knowledge, none of the existing low-code development environments offers support for 1) computation at the edge, 2) coordinating with the edge, and 3) handling partial failures at the edge when network connections become unavailable. DisCoPar has the most potential to be used for constructing SAAs. Still, it lacks the following: (1) custom sensing components for smart agriculture, (3)

computation at the edge and components that can execute at the edge, (3) features for coordinating mobile components with the edge, and (4) handling partial failures at the edge.

In our work, we design components to support the four properties of SAAs, and we implement them by extending DisCoPar (see Chapter 4 and Chapter 5). In particular, we design components that can handle partial failures at the edge, components that can be used for sensing environmental conditions, components that can perform computations at the edge, and support coordinating mobile components with the edge. The choice of DisCoPar was motivated by the fact that it already offered support for offline accessibility on mobile phones. Before explaining our work, we discuss the necessary background details for DisCoPar in the next chapter.

# Chapter 3

## DisCoPar

This work builds on DisCoPar [Zam18], a component-based low-code development environment inspired by flow-based programming [Mor10] for constructing participatory campaigning applications in citizen science. Recall from Chapter 2 that DisCoPar already offers support for sensing capabilities and handling partial failures on mobile devices, making it a good choice for prototyping our research on low-code development environments for smart agriculture applications. This chapter describes the necessary information on DisCoPar to understand our contributions.

### 3.1 Architectural Overview of DisCoPar

Figure 3.1 shows the DisCoPar architecture, which consists of three different parts, namely, mobile clients, a server with a data store and a web-based client. Each part of DisCoPar plays a different role.

*Mobile clients* gather data through onboard sensors and user input and upload it to the server for aggregation and analysis. Mobile client users can also receive real-time feedback, e.g., data gathering coordination instructions and visualisations.

*Server* is used for data processing and long-term storage. Note that citizen science applications utilise campaigns for data collection that require a set of steps to be followed. The data collected varies in form and type. Hence, many different processing strategies can be used to analyse the collected data. For example, determining noise



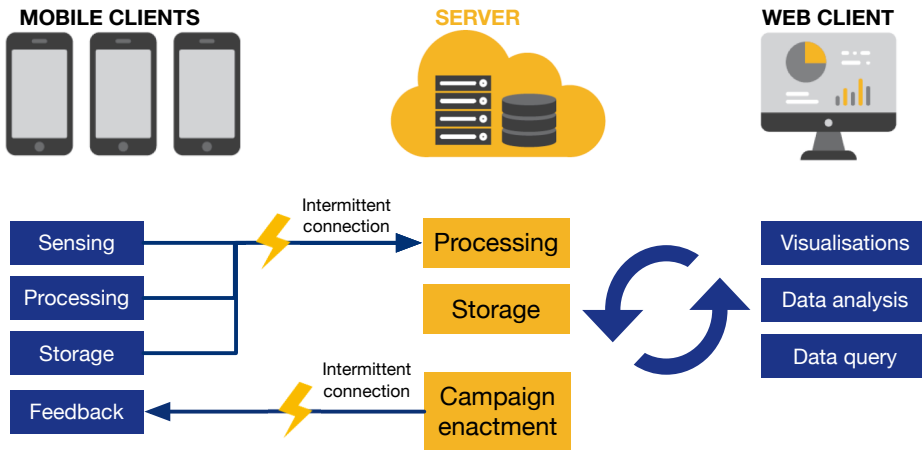


Figure 3.1: Architectural overview of DisCoPar [Zam18]. The architecture features three parts: (1) the mobile clients, (2) the server and (3) the web client (dashboard). The lightning strikes show intermittent network connections between the mobile client and the server.

levels at a particular location requires repeated data collection for the noise measurements and the GPS coordinates associated with each measurement. The noise levels can also be visualised on a map.

*Web-based client* provides end-users with a dashboard they can use to reason about the data collected by the mobile clients, e.g., graph visualisations or map-based plotting of data samples on a dashboard.

DisCoPar follows the classical client/server architecture featuring a centralised server for coordination and communication. Each of the three parts of DisCoPar designates different *execution scopes* for components. A component’s scope can be mobile, server, or dashboard (i.e., the web client) depending on where they execute as shown in Figure 3.1. Mobile clients can transparently establish connections to the server-side data processing using dedicated components. Figure 3.2 shows a screenshot of the visual programming environment of DisCoPar. The left side of the figure shows the canvas for composing applications, while the right side shows the component menu with three component categories. The component categories correspond to the three execution scopes mentioned above. Application designers can drag and drop components from any component

category onto the canvas. The canvas features a mobile phone emulation and a web dashboard that enable application designers (developers) to preview the expected result of the application.

### 3.1.1 DisCoPar Layers

The design of DisCoPar is conceptually split into two layers: (1) a graph layer and (2) a component layer that contains the source code for each component and the necessary abstractions to compose them. The graph layer provides the visual syntax for programming applications, i.e., it provides the application development environment. Beneath the graph and component layer lies an execution engine built using JavaScript. To execute an application, the constructed graph is loaded onto the execution engine of DisCoPar in JSON format (Appendix A). The execution engine then initialises the components in the graph by loading their JavaScript source code from the component layer. Initialising the components enables the execution engine to create data streams between the components. Each component activates automatically based on data availability on its incoming ports.

As mentioned before, DisCoPar provides a built-in library of components for data gathering and participatory sensing. More concretely, DisCoPar supports the following categories of components.

*Sensing components:* These components gather data through sensors embedded in the mobile device (e.g., noise, GPS coordinates) or via user input, e.g., questionnaires and social tagging. Sensing components can either directly upload their data to the server for aggregation and analysis or perform pre-processing on the mobile device.

*Data processing components:* This category of components includes aggregation and data analysis. For example, grouping sensor measurements according to geographical locations and averaging the grouped measurements. Lastly, the components include filtering data based on meta-information, such as the identity of the devices generating the data.

*Feedback components:* These components handle interaction with the user, e.g., instructions on how to perform measurements. Some components can be used to inform participants of incorrect sensing behaviour, e.g., not collecting data in the assigned location. It

is essential to send this feedback to participants to correct their behaviour, which may affect the quality of the gathered data.

## 3.2 DisCoPar Visual Programming Environment

As previously mentioned, DisCoPar embraces a flow-based and visual programming approach in which an application comprises different components, each representing a computation task [Zam18, ZKD18, ZKD21]. A DisCoPar application is thus represented as a directed acyclic graph similar to the graph shown in Section 2.5 (Figure 2.6). The graph nodes are processes, while the edges are the connections between ports. Constructing a graph in flow-based programming can either be done using a textual domain-specific language [Bur99] or graphically using a visual programming tool or environment [Cha87a, RMMH<sup>+</sup>09, RSF17]. In DisCoPar, graphs are created through a web-based visual programming environment. The visual programming environment allows users to compose components into an application graph through drag-and-drop actions. Figure 3.2 presents the visual programming environment showing the canvas and the component menu for DisCoPar. The canvas is a pane for composing components into applications. In the sample application shown in Figure 3.2, the *DisplaySoundLevel* and *DisplayOnScreen* are mobile components that create labels with their respective data on the mobile device. The *SoundPressureLevel* is a mobile component that creates a button on the mobile device that can be clicked to start recording sound level measurements. The *DisplayAsTable* is a web component that displays data on the web dashboard. The *ObservationToTable* is a server component for fetching data from a database. Lastly, the *ObservationDatabase* is a server component storing recorded data in a database.

### 3.2.1 DisCoPar Components

As mentioned in Section 3.1, DisCoPar applications consist of components that run on mobile phones and communicate to a server backend or a web-based dashboard on the server. The component name, execution scope, ports, and task define all DisCoPar components. Once published, the components are listed in their respective grouping (categories) in the

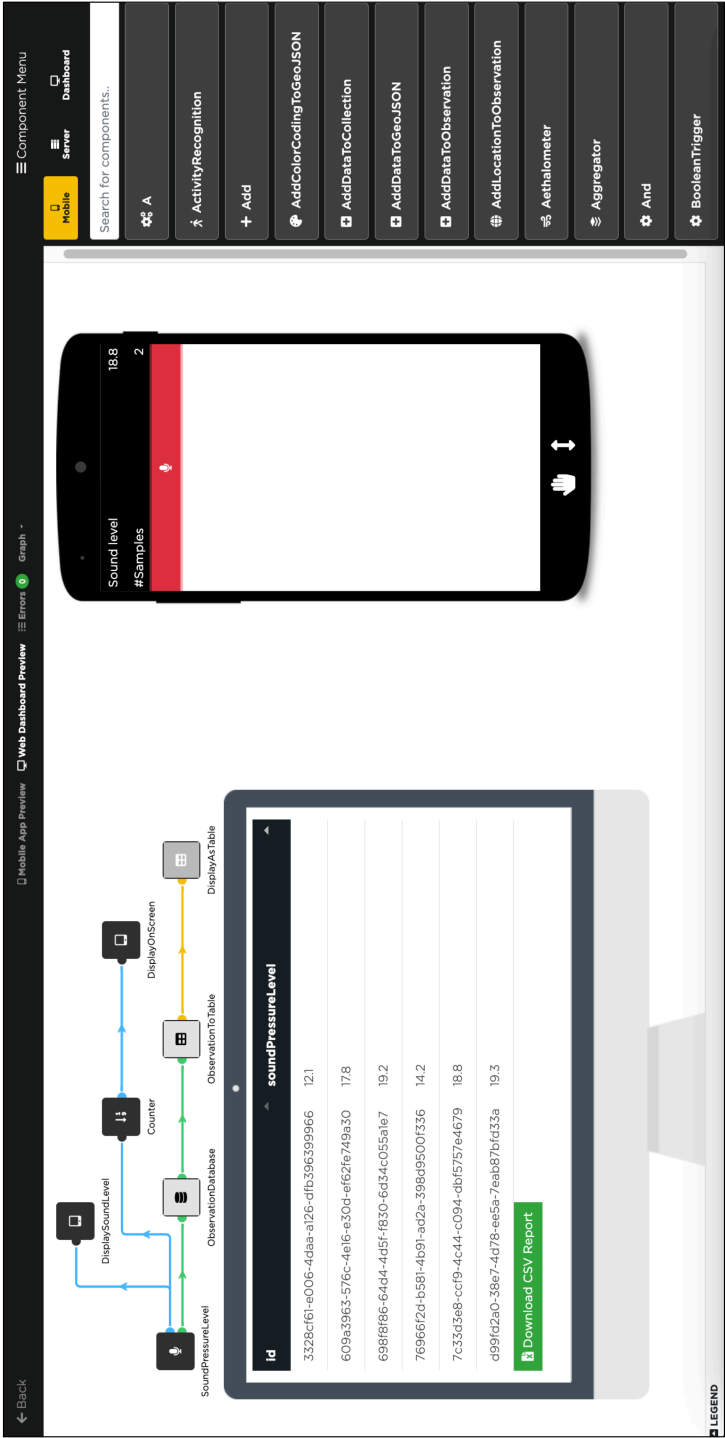


Figure 3.2: Visual programming environment of DisCoPar showing the canvas, an application graph composed of components with different execution scopes, the component menu and emulations for mobile devices and the web dashboard.

visual component library<sup>1</sup>.

At the implementation level of DisCoPar, an instance of a component is called a *process*. The *process* is an asynchronously executing piece of logic. Multiple processes of the same component can be simultaneously active. A process is stateful and can access its internal state and ports but not other processes. Processes communicate by sending and receiving structured data chunks called *information packets*. A process is activated when it receives *information packets* on one of its input ports. Processes automatically activated on application startup, such as processes emitting constant values on their output ports, are exceptions to this rule.

### 3.2.1.1 Component Connections

Components send information packets to other connected components. Connections between components provide the path on which information packets can “flow” from one process’s output port to another process’s input port. The connections can be implemented using bounded buffers or first-in, first-out queues. The size of the buffer or queue is referred to as the *connection capacity*. When the connection capacity is 0, the information packets are transferred immediately between the sending and receiving processes.

In the visual programming environment, the connections are represented as lines linking an output port to an input port. A connection’s *colour* indicates the *type of data* flowing from output to input ports. We explain this later by example in Section 3.2.1.4. The connections can be established between components that have different execution scopes. Connecting components that have a mobile execution scope to those that have a server scope automatically and transparently creates a distributed application.

### 3.2.1.2 Component Configuration

When right-clicking a component on the canvas, a menu appears as shown in Figure 3.3, which enables the application designer to delete the component from the flow graph, open its configuration window or check more information about the component. Figure 3.4 depicts the configuration

---

<sup>1</sup>Mobile, server and web component categories.

window of the *DisplayOnScreen* component used in Figure 3.2. This component displays the data it receives on the screen and can be configured to change the labels. Each component drawn on the canvas corresponds to one process of that particular component when the application is deployed. Since each process has its configuration, two processes of the same component can behave differently using different settings. This mechanism is also referred to as *process configuration*. For example, the application developer can add two *DisplayOnScreen* components on the canvas but configure them differently.

The application designer’s actions on the canvas are immediately and persistently updated. As such, there is no need to explicitly “save” a graph. The canvas also provides emulations for the mobile application and web dashboard that can help application designers see the final look and feel of the application as shown by Figure 3.2.

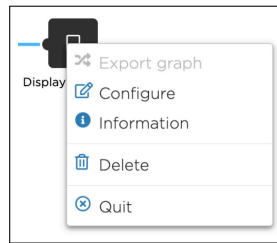


Figure 3.3: Menu options available when right-clicking a component on the DisCoPar canvas.

Figure 3.4: The configuration window for the *DisplayOnScreen* component. The label to display is specified under the “Heading” section.

### 3.2.1.3 Component Ports

Component ports are the communication points between components. Each port in DisCoPar is named to make it easy to refer to. A component can have multiple input or output ports. The input ports provide a receive functionality to dequeue information packets from a connection's buffer. The output ports provide a send functionality to queue information packets into the port of a connected process. A process can send data to or receive data from any of its ports. A process reacts to data arrival on an input port and executes some code.

The executed code can produce some output, which is then sent via one of its output ports to another component, e.g., to display the value on the application's graphical user interface. It is possible for a component not to have an input port, in which case it acts as a *source* component in the application graph. The *source* is a component that automatically produces some output, e.g., sensor components such as the *SoundPressureLevel* component. Similarly, a component without an output port acts as a *sink* component in the application graph. A *sink* is a component that consumes data, such as the *DisplayOnScreen* component that displays messages received as shown in Figure 3.2.

### 3.2.1.4 Port Typing

DisCoPar features port typing. This implies that ports can only be connected if the output of one component can serve as the input of another component. As such, to connect two successive components, they must be type-compatible. The canvas includes a visual feedback mechanism showing the port typing constraint on connections. When dragging a connection from an output port, only input ports with matching colours, i.e., those that can accept input from that particular output port, are highlighted and can be used. The exception to this rule is input ports of the *Any* type, which are always highlighted as they accept any input.

Figure 3.5 shows the port typing principle when an application developer creates a connection from the output port of the *SoundPressureLevel* component. The blue output port of the *SoundPressureLevel* component emits *numeric* data. In this example, there are two compatible ports: the input ports that are coloured blue because they accept *numeric* values and the input ports that are coloured black. After all, they accept any data.

The input ports of the *ComputeAverage*, *ComputeMaximum* and *ComputeMinimum* components are highlighted because they accept numeric data. The input ports of the *DisplaySoundLevel* and *Counter* components are highlighted since they accept any data. Thus, port typing prevents the creation of a graph in which two components are connected through incompatible port types.

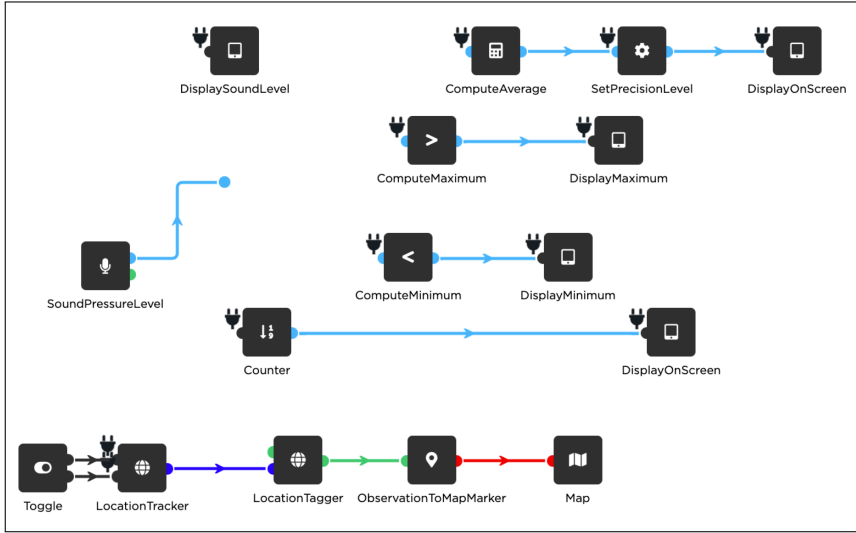


Figure 3.5: Port typing and highlighting compatible component ports in DisCoPar. This example highlights all ports that accept numeric data with the electric plug symbol.

Figure 3.6 shows the supported data types in DisCoPar and the corresponding colours for component ports and connections.

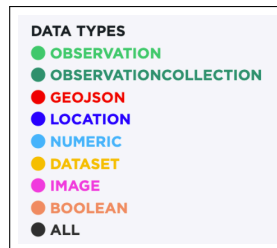


Figure 3.6: Supported data types in DisCoPar. The supported data types are distinguished by colour, i.e., each data type has a distinct colour.



### 3.2.2 Graph Validation

Besides checking port types, DisCoPar features a graph validation mechanism that alerts application designers whenever errors occur in the flow graph during application design. An application graph is considered valid if it satisfies every constraint. Currently, DisCoPar supports three types of constraints: *IncomingData*, *OutgoingData* and *pathExists* constraints. The *IncomingData* verifies whether a certain input port of a process has at least one connection arriving on the port. The constraint ensures that a process with an input port receives some data on the input port. Similarly, the *OutgoingData* constraint checks if an output port of a process has any outgoing connections. The *pathExists* constraint takes an input port of one process and an output port of another and tests whether a path exists between the two ports using the breadth-first search algorithm.

Whenever a constraint is not satisfied, an error message is shown to the application graph designer to indicate that there are still some unresolved issues. For instance, Figure 3.7 shows three error messages indicating the components in an application graph that have issues and need to be resolved on the application flow graph. The constraints are reevaluated whenever the graph is modified, and the visual cues are updated accordingly.

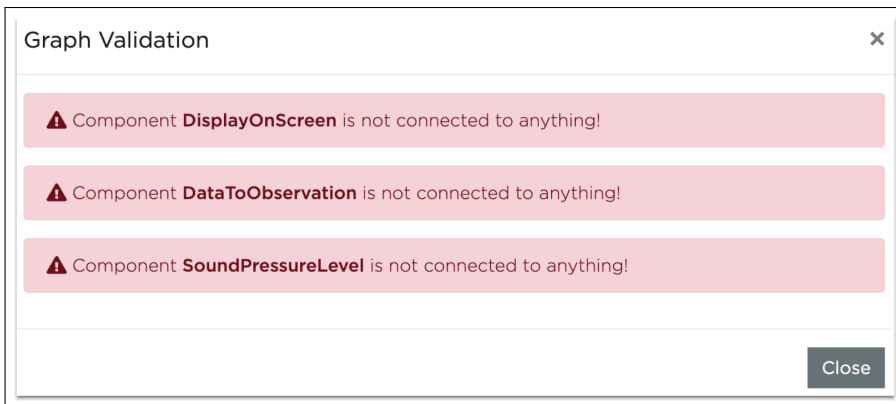


Figure 3.7: Graph validation error message examples in DisCoPar. In this example, the error messages show the components that are not connected.

### 3.2.3 Handling Partial Failures

DisCoPar considers that mobile applications can be deployed in areas that experience network connection issues. As illustrated in Figure 3.1 with the lightning strikes, there is only one point where the network connection issues can be experienced, i.e., between the mobile client and the server dashboard. Network connection issues can cause the mobile application to fail to communicate with the server, resulting in data loss. Hence, DisCoPar provides features for handling partial failures on the mobile side using a specialised component for offline accessibility called *InDatabaseBuffering*. Figure 3.8 shows the specialised component for offline accessibility in action.



Figure 3.8: Implementation of an application showing the use of the *InDatabaseBuffering* component to handle partial failures.

In this example, the application graph is composed of three components. The *Survey* and *InDatabaseBuffering* components execute on the mobile scope while the *ObservationDatabase* component runs on the server. The *Survey* component creates a data collection survey on the mobile phone. The collected data is stored on the mobile phone using the *InDatabaseBuffering* when the network connection becomes unavailable. The *ObservationDatabase* component creates a database on the server and saves data into it. The *InDatabaseBuffering* creates one lightweight database on the mobile phone to keep data for all connected components. Using the database on the mobile phone enables data to be persistently stored until the network connection is restored. The data saved on the mobile phone is automatically sent to the server when the network connection is restored.

## 3.3 DisCoPar by Example

To illustrate how to build applications in DisCoPar, consider the case of a mobile application that measures and monitors noise levels in the

environment. The noise levels are read as noise samples using the mobile phone microphone as a sound sensor. The application runs fully on the mobile device and performs the following tasks.

1. Computes the average noise level.
2. Determines the maximum noise level.
3. Determines the minimum noise level.
4. Display the measurements on the mobile device.
5. Show the noise levels on a map based on their GPS coordinates.

Figure 3.9 shows the example application for monitoring noise levels implemented in DisCoPar. The application graph is read from the left to the right. The arrows on the component links show the direction in which data flows. All components are black, as they all run on the mobile scope.

The application has eight functional requirements:

1. *Reading noise levels.* The application reads the noise levels using the *SoundPressureLevel* component. It is a source component with two output ports emitting numeric values and observations. The numeric values are emitted through the blue port, while the observations are emitted through the green port. An observation is a standard format for citizen science data generated by data-producing processes [Zam18]. The *SoundPressureLevel* component automatically adds a button to the application interface for turning on and off the sound sensor (i.e., phone microphone).
2. *Computing the average noise level.* The average noise level is computed using the *ComputeAverage* component. The average is computed for a data stream coming from *SoundPressureLevel* values.
3. *Determining the maximum noise level* is done using the *ComputeMaximum* component.
4. *Determining the minimum noise level* is done using the *ComputeMinimum* component.
5. *Tagging noise level readings with GPS coordinates.* The application reads the GPS coordinates using the *LocationTracker* component. The noise level measurements from the *SoundPressureLevel* component are tagged with the GPS coordinates using the *LocationTagger* component.

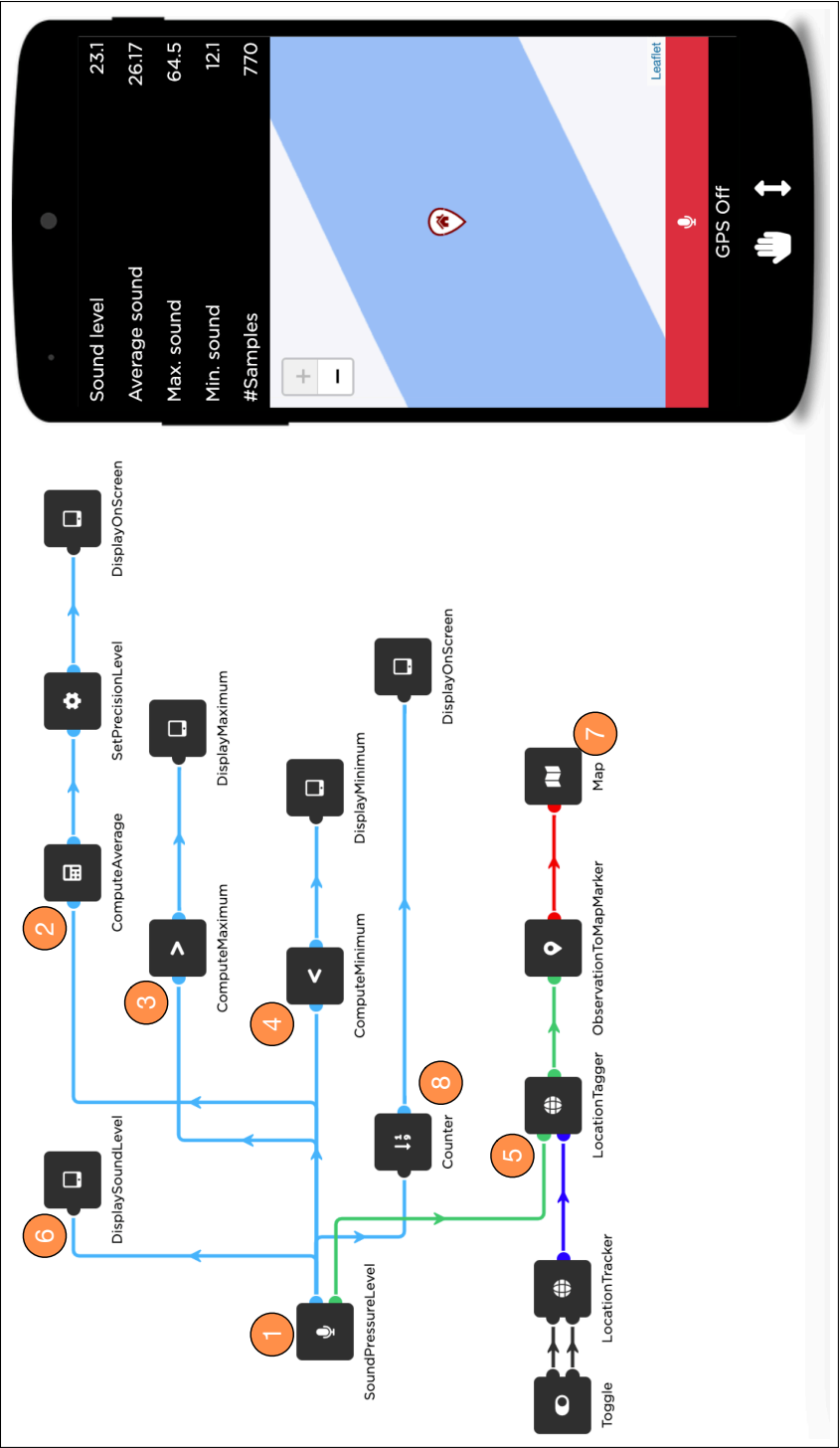


Figure 3.9: An example application in DisCoPar for measuring and monitoring noise levels. All the components in this application execute on the mobile scope.

6. *Map the measurements.* Mapping the measurements uses the *Map* component that creates a map on the mobile device and shows noise levels on the map. The measurements are converted from observations for mapping using the *ObservationToMapMarker* component.
7. *Displaying the measurements on a graphical user interface (mobile phone screen).* The measurements are displayed on the screen using the *DisplaySoundLevel* component. The average measurements are displayed using *DisplayOnScreen* component. The maximum and minimum measurements are displayed using the *DisplayMaximum* and *DisplayMinimum* components, respectively. The above components create labels on the mobile device and display respective measurements against each label.
8. *Count the number of measurements.* The *Counter* component keeps track of the number of noise level samples. The samples are displayed using the *DisplayOnScreen* component.

As mentioned before, some components, such as the *SoundPressureLevel* component, act as *source* components while others, such as the *DisplayOnScreen* component, act as *sink* components. The source and sink components can be used to build the application's graphical user interface. As mentioned before, the application flow graph is executed via DisCoPar's execution engine.

### 3.4 Conclusion

This chapter describes DisCoPar, a low-code development environment based on the flow-based programming paradigm. DisCoPar was initially designed to create citizen science applications with sensing, data processing, and coordination components. Applications implemented in DisCoPar follow a client/server architecture relying on a centralised server for communication and coordination. Lastly, DisCoPar features components for creating data collection surveys and saving the collected data into cloud-hosted databases.

DisCoPar components can be helpful for domain experts, such as agricultural extension workers, who often use farm data to advise farmers. For instance, DisCoPar features sensing components, e.g., for measuring and monitoring noise levels. There is no support for gathering sensing data in

crop fields or farms (i.e., at the edge), such as sensing soil moisture and temperature levels. Therefore, the featured sensing components must be expanded for smart agriculture applications.

DisCoPar already features a specialised component for offline accessibility on mobile phones that keeps data on the mobile phone whenever the network fails. The stored data is automatically sent to the server when the network connection becomes available. However, it does not offer support for handling partial failures at the edge or different strategies for dealing with offline accessibility. Lastly, DisCoPar does not have built-in validation mechanisms to avoid incorrect composition of offline accessibility components.

In summary, to implement smart agriculture applications based on the properties identified in Section 2.3, DisCoPar: (1) features sensing capabilities, albeit not specific to smart agriculture and (2) supports handling partial failures on the mobile phone. However, DisCoPar lacks the capabilities of (1) executing components at the edge, (2) coordinating the edge with mobile components and (3) handling partial failures at the edge. Finally, DisCoPar lacks components to accumulate data from sensors and track sensors when they go offline.



## Chapter 4

# DisCoPar-Kilimo

In Chapter 2, we identified four properties that low-code development environments need to adhere to facilitate smart agriculture application development. Our review of existing low-code development environments in Section 2.5 showed that none of the existing low-code development environments offer support for the properties identified in Chapter 2 as environment sensing, computation at the edge, coordination with the edge and handling partial failures.

This chapter presents DisCoPar-Kilimo, a low-code development environment for constructing smart agriculture applications. DisCoPar-Kilimo provides and supports the properties mentioned above. The chapter begins by describing our approach and the features of DisCoPar-Kilimo. It then describes the DisCoPar-Kilimo architectural overview and the visual programming environment it provides for composing applications. In addition, it describes in detail the features of our approach. Lastly, we end the chapter by describing how to design and implement applications in DisCoPar-Kilimo.

### 4.1 Our Approach in a Nutshell

We now describe the fundamental ideas of our approach to build a low-code development environment specially designed to support the four properties as identified in Section 2.3.

*Environment sensing.* Our approach offers dedicated sensing components for monitoring environmental conditions relevant to smart agricul-



ture, such as soil moisture, temperature, and humidity. The three parameters are essential for the seeding, growth and development of crops, and as such, they should be offered as built-in components. Additionally, our approach introduces components that can filter specific sensor data and enable additional computations to be performed on that data.

*Computation at the edge.* We introduce (1) infrastructure to support computations on devices installed at the edge, (2) infrastructure to support exporting and executing an application graph at the edge and lastly, (3) a set of built-in components to perform computations on devices installed at the edge, e.g., to process and keep data at the edge before being sent to the server or the mobile phone components of an application. The infrastructure also supports the environment sensing components to execute at the edge. The components running on edge devices are collectively referred to as *edge components*. The *edge components* can perform different computations, such as computing the soil moisture and temperature averages.

*Coordination with the edge.* Edge components require coordination to communicate with the mobile components. Recall from Section 2.2 that multiple edge devices can be installed on the farm. Our approach devises a mechanism to allow edge components to communicate with mobile components directly. In addition, our approach proposes components to accumulate data from multiple edge devices on the mobile phone for further processing.

*Handling partial failures.* We introduce components to deal with intermittent network connections. The aim is to keep data whenever network connections become unavailable. Our approach offers offline accessibility at two levels. First, we introduce offline accessibility policies on the mobile, such as time-based and memory-based policies between mobile, server, and dashboard components. Second, we introduce offline accessibility components to keep data when network connections between edge and mobile devices become unavailable, considering that edge devices are often more resource-constrained than mobile devices. Additionally, we added a mechanism for monitoring the connectivity status of edge devices to enable tracking of which devices are online.

In the subsequent sections, we present and explain the architectural overview of our approach, the design of components and its integration into a concrete LCDE, DisCoPar-Kilimo.

## 4.2 Architectural Overview of DisCoPar-Kilimo

Figure 4.1 shows the architectural overview of DisCoPar-Kilimo, the LCDE where we prototyped our approach. The architecture extends DisCoPar’s architecture shown in Figure 3.1 and consists of four different parts labelled 1, 2, 3, and 4 corresponding to devices at the edge, mobile client, a server with storage, and a web dashboard (web client) for visualisation. We assume that sensors at the edge are attached to microcontrollers, e.g., the ESP32<sup>1</sup> family of microcontrollers. The microcontroller can perform computations for processing and storing data at the edge and features networking technology to communicate with the mobile device. As mentioned before, we generally call those microcontrollers with sensors as *edge devices*.

Motivated by the driver scenarios in Section 2.2, mobile devices (e.g., phones) are used for data collection, processing, and storage. The data can also be entered directly into the mobile application by the end users via data collection surveys. Like DisCoPar, the server is purely used for data processing and long-term storage. Lastly, the web dashboard is used for data visualisation. The four parts of Figure 4.1 represent execution scopes for DisCoPar-Kilimo components. Our architecture, thus, has four scopes distinguished by colour: (1) the edge scope (blue), (2) the mobile scope (black), (3) the server scope (light grey), and lastly, (4) the web (dashboard) scope (grey). Data processing can happen at three points, i.e., at the edge, on the mobile client and the server. Data visualisation can happen at two points, i.e., the mobile and web dashboards. The lightning strikes in Figure 4.1 depict two points of failure of DisCoPar-Kilimo applications due to network connection issues. The first point of failure is between the edge and mobile scopes, and the second is between the mobile client and the server. To avoid data losses when the network connections become unavailable, DisCoPar-Kilimo offers dedicated offline accessibility components detailed later in Section 4.3.4.

Table 4.1 shows the overview of new edge and mobile components that

---

<sup>1</sup><https://www.espressif.com/en/products/socs/esp32>

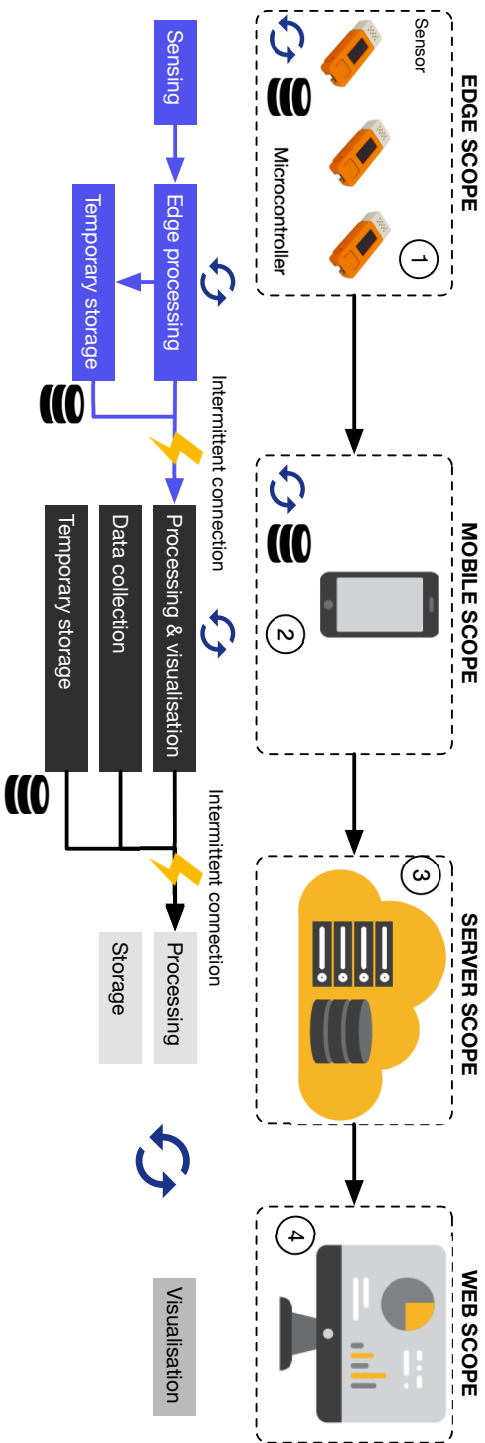


Figure 4.1: Architectural overview of DiSCoPar-Kilimo. The architecture depicts four execution scopes. The edge scope is novel and features edge components. The mobile scope features mobile components. The server scope features server components. Lastly, the web scope features web components.

DisCoPar-Kilimo incorporates into a low-code-development environment. Some components were incorporated in DisCoPar-Kilimo to support the implementation of SAAs.

### 4.3 DisCoPar-Kilimo Visual Programming Environment

We now introduce DisCoPar-Kilimo’s visual programming environment (VPE), in which we integrated novel components for building SAAs. Figure 4.2 provides an overview of DisCoPar-Kilimo VPE and component execution scopes (i.e., mobile, edge, server, and dashboard). The figure shows the edge scope highlighted in yellow in the component menu with the edge components. Components for each scope are distinguished by colour. On the canvas, the edge components are depicted in blue, the mobile components in black, the server in light grey, and the dashboard in grey. The component menu allows developers to search for all components in the different scopes using their names or parts of their names. Unlike the other components, edge components do not provide an execute method and are strictly executed at the edge (i.e., they cannot have multiple execution scopes).

In the remainder of this section, we describe in detail the features introduced by DisCoPar-Kilimo to simplify the development of SAAs.

#### 4.3.1 Ensuring Support for Environment Sensing

DisCoPar-Kilimo introduces components for environment sensing that are hosted at the edge scope and can be deployed in farms and crop fields. The built-in sensing components allow measuring and monitoring *soil moisture*, *temperature*, and *humidity* conditions. Figure 4.3 shows an example application for sensing and sending soil moisture data to the mobile device. The example shows two connected components, i.e., one for reading the soil moisture (*ReadSoilMoisture* component) and another one for displaying the soil moisture data on the mobile device (*DisplayOnScreen* component). The *ReadSoilMoisture* component executes at the edge while the *DisplayOnScreen* component executes at the mobile.

In contrast to the state-of-the-art LCDEs like DisCoPar, DisCoPar-Kilimo offers sensing capabilities at the edge scope. In a smart agriculture

Component	Input port and type	Output port and type	Description
<i>Edge components</i>			
Addition	first: EDGE, second: EDGE	out: EDGE	Performs add operation.
Subtraction	first: EDGE, second: EDGE	out: EDGE	Performs subtract operation.
Multiplication	first: EDGE, second: EDGE	out: EDGE	Performs multiply operation.
Division	first: EDGE, second: EDGE	out: EDGE	Performs divide operation.
Exponentiation	exponent: EDGE, base: EDGE	out: EDGE	Performs exponentiation operation.
ComputeEdgeAverage	in: EDGE	out: EDGE	Computes average.
PreviousValue	in: EDGE	out: EDGE	Keeps previous value.
ReadsSoilMoisture	–	out: EDGE	Reads soil moisture.
ReadTemperature	–	out: EDGE	Reads temperature.
ReadHumidity	–	out: EDGE	Reads humidity.
BufferData	in: EDGE	out: EDGE	Stores data in a buffer in memory.
BufferOnDisk	in: EDGE	out: EDGE	Stores data on disk.
SetConstant	–	out: EDGE	Sets and sends a constant number.
<i>Mobile components</i>			
UnWrap	in: EDGE	out: NUMERIC	Unwraps payload.
UnWrapForSpecificDevice	in: EDGE	out: NUMERIC	Unwraps payload for a specific device.
DeviceAccumulator	in: EDGE	data: EDGE, out: NUMERIC	Accumulates data from multiple devices.
DataArrayToTable	in: ALL	number: NUMERIC, out: DATASET	Converts data into a dataset.
ConnectedDevices	–	out: EDGE	Tracks connected devices.
HybridPolicyBuffering	in: OBSERVATION	out: OBSERVATION	Stores data in a buffer in memory.
TimePolicyBuffering	in: OBSERVATION	out: OBSERVATION	Stores data in a buffer in memory.
RecordPolicyBuffering	in: OBSERVATION	out: OBSERVATION	Stores data in a buffer in memory.
InMemoryBuffering	in: OBSERVATION	out: OBSERVATION	Stores data in a buffer in memory.
GenerateAndShowAlert	in: ALL	out	Generates and displays alerts.
ReadQRCode	–	out: OBSERVATION	Reads QR code.
SetThreshold	–	out: NUMERIC	Sets a threshold value.
Compare	threshold: NUMERIC, input: NUMERIC	out: BOOLEAN	Compares two inputs.
PlotSoilMoisture	in: ALL	–	Plots a line chart.

Table 4.1: Summary of edge and mobile components of DisCoPar-Kilimo. EDGE refers to data coming from edge devices in JSON format. OBSERVATION refers to data created by data-producing processes, NUMERIC refers to numerical data, DATASET refers to a collection of related data sets, and ALL refers to any data.

### 4.3. DISCOPAR-KILIMO VISUAL PROGRAMMING ENVIRONMENT

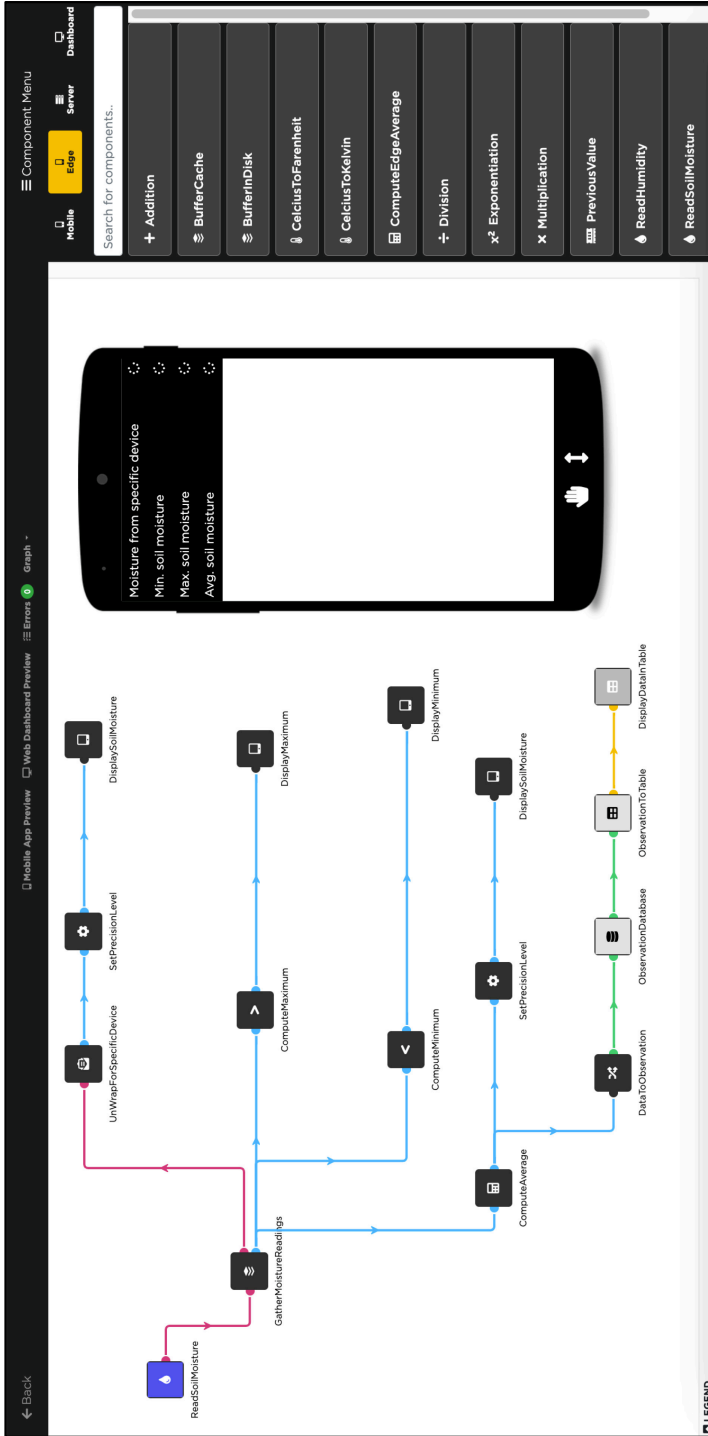


Figure 4.2: Component scopes of DisCoPar-Kilimo on the visual programming environment. The scopes correspond to the architectural model depicted in Figure 4.1.

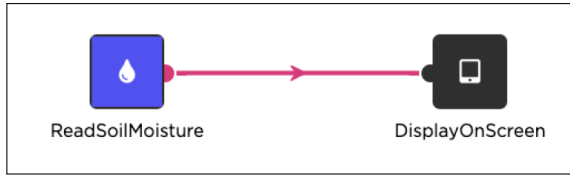


Figure 4.3: Example environment sensing application.

context, the edge computing components introduced by DisCoPar-Kilimo are expected to support environmental sensing and storing data on environmental conditions on the microcontrollers hosting sensors.

### 4.3.2 Ensuring Support for Computation at the Edge

The edge scope hosts components in DisCoPar-Kilimo that support *computations at the edge*. Recall from Chapter 2 that edge devices are resource-constrained and cannot execute entire application graphs. Only the part of the application graph containing the edge components is exported and deployed to the edge devices during deployment. DisCoPar-Kilimo introduces a *stripped down* version of components that can be used in drawing complete application graphs and have corresponding functions executed at the edge to perform the computational tasks of those components. The components are stripped down because the edge devices are resource-constrained and cannot run the infrastructure for executing flow-based application flow graphs. We refer to the functions called to execute at the edge as *companion functions*. The companion functions that require interacting with the hardware of edge devices invoke low-level functions that we call *friend functions* as explained later in Chapter 5. The friend functions perform low-level operations, such as reading data from a soil moisture sensor. The companion functions are implemented to receive information from the friend functions.

#### 4.3.2.1 Extracting edge graph

We devised a process to extract an edge graph from the overall application graph to execute edge components. The process takes an entire application graph and iteratively traverses it to identify edge subgraphs and edge components and how they connect. The process then generates the function calls (i.e., edge function calls) representing the edge application flow

graph. The edge function calls execute on the edge devices to perform the edge application flow graph tasks. To explain how the process works, let us consider an application graph with four edge components as shown in Figure 4.4.

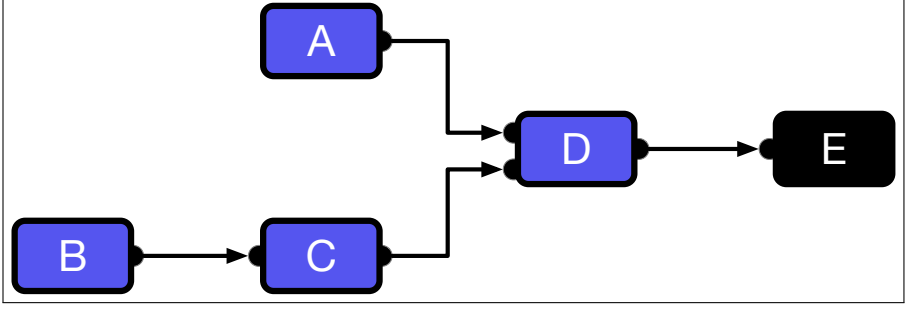


Figure 4.4: Application example with four connected components.

The arrows in the graph indicate the direction of data flow. Components *A* and *B* are source nodes and do not receive any input. Component *C* receives input from component *B*, and component *D* receives input from both components *A* and *C*.

#### 4.3.2.2 Building the edge application

After extracting the edge graph, the devised process performs a topological sorting of the operations represented by the edge components in the graph. Then, the process builds calls to the respective companion functions as follows.

*Let  $x_1$  be  $A()$ .*  
*Let  $x_2$  be  $B()$ .*  
*Let  $x_3$  be  $C(x_2)$ .*  
*Let  $x_4$  be  $D(x_1, x_3)$ .*

$A()$ ,  $B()$ ,  $C()$ , and  $D()$  are the companion functions of components *A*, *B*, *C*, and *D*. The arguments in the calls to companion functions  $C()$  and  $D()$  represent data passed from upstream components, i.e.,  $x_1$  which is the output from component *A*,  $x_2$  which is the output from component *B* and  $x_3$  which is the output from component *C*. The above function calls are packaged within a DisCoPar-Kilimo loop function, which is an entry



point for all executions at the edge devices. We detail later in Chapter 5 how the companion functions call their friend functions.

### 4.3.3 Ensuring Support for Coordination with the Edge

DisCoPar-Kilimo enables direct communication between edge and mobile scope components (e.g., between microcontrollers and mobile phones) without requiring a centralised coordination server, as is needed in existing approaches like DisCoPar or Node-RED. This is necessary to support SAAs deployed to rural areas that experience limited connectivity, as explained in Chapter 1 and Chapter 2, and the farmer periodically visits the farm to collect data from the edge devices.

To provide coordination, the mobile scope portions of the application graph are specialised to act as an access point for edge devices. The mobile scope runs a dedicated networking code to accept connections from the edge devices. The dedicated networking code ensures that the edge devices can send data directly to the components running on the mobile device, which can communicate to the DisCoPar-Kilimo server. During deployment, the edge devices are configured with the credentials to connect to the mobile network eventually. Data from the edge devices can eventually reach the server via mobile phone components for further processing and long-term storage.

### 4.3.4 Ensuring Support for Handling Partial Failures

DisCoPar-Kilimo is the only LCDE that offers offline accessibility facilities for both mobile and edge devices. We now detail the built-in offline accessibility components and policies offered by DisCoPar-Kilimo.

#### 4.3.4.1 Policies for Handling Partial Failures on the Mobile

Based on the scenarios presented in Section 2.2, we distilled four different policies of offline accessibility for SAAs. The introduced policies enable developers to keep data upon a disconnection between the mobile and server or the web based on:

1. The number of data items, i.e., records.
2. The available memory for data storage.

3. The time duration to keep data.
4. A combination of time and available memory, i.e., hybrid policy.

The above policies enable configuring applications for different contexts. In all the above policies, data is kept in a buffer in memory. In what follows, we describe the rationale for each policy.

*Record-based policy:* The first policy allows the application designer to specify the maximum number of records stored during a network failure. When the upper limit is surpassed, the oldest records are removed from the buffer to create space for more recent data. This policy was motivated by the fact that farmers are often interested in the latest measurements for farm conditions [PE08]. The number and data type determines how many records will be kept in memory.

*Memory-based policy:* The second policy sets a maximum amount of memory that can be used to store data. Old records are removed when stored data exceeds the maximum allocated memory space.

*Time-based policy:* The third policy is based on the time that data can be kept in memory that we call the *lease* window and assumes an infinite buffer. The *lease* window is configurable and can vary based on specific application requirements explained when the lease starts. The lease window works globally for all data records stored in the buffer. So, if a connection is restored within the lease window, data records are pushed to the server. Otherwise, all data records are removed from the buffer when the lease window expires.

*Hybrid policy:* The fourth and last policy (i.e., hybrid policy) combines the upper limit of records and the *lease* window to keep data. The policy checks to ensure that memory is not depleted as data records are added to it within the lease window.

#### 4.3.4.2 Components for Handling Failure on the Mobile

DisCoPar-Kilimo implements the offline accessibility policies described in Section 4.3.4 into dedicated components. When data arrives on the input ports of those components, and there is connectivity, the components forward it to their output ports. When data arrives and there is a disconnection, the data is handled based on the specified policy. Application

developers can specify the policy parameters as configuration settings for each offline accessibility component. Specifying the parameters is done by right-clicking the component on the canvas and selecting the *configure* option.

In what follows, we explain the built-in offline accessibility components in DisCoPar-Kilimo.

*Record-based offline accessibility component.* This component specifies an upper limit of records stored during a disconnection. The data is pushed into a buffer in memory as it arrives at the input ports of the component. The developer can specify the number of records by configuring the component. When the internal counter of records exceeds the upper limit, old records are removed to create space for recent data. Figure 4.5 shows the configuration window for specifying the number of records.



Figure 4.5: Window for configuring the *record-based* offline accessibility component. The number of records to keep is specified as a numerical value.

*Memory-based offline accessibility component.* This component specifies an upper limit of memory in bytes that can be used to store data. Records received on its input port are stored in a buffer in memory until the buffer is full. Each time data is pushed into the buffer, its size is computed and determined. When the memory used by the buffer exceeds the upper limit, old records are removed. The developer can specify the upper limit for the memory as shown by Figure 4.6.

*Time-based offline accessibility component.* This component specifies a lease window in minutes to store data in a buffer in memory. The lease window starts when a disconnection occurs. Stored data is removed from the buffer when the lease window expires. If the connection is restored before the lease window expires, data stored in

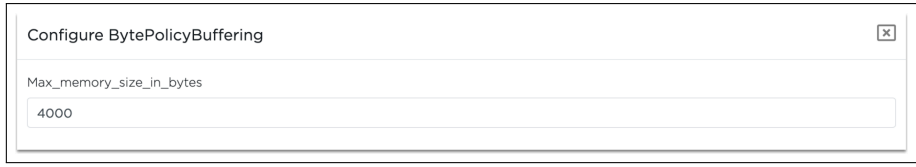


Figure 4.6: Window for configuring the *memory-based* offline accessibility component for specifying the maximum memory for data storage. The memory size is specified in bytes.

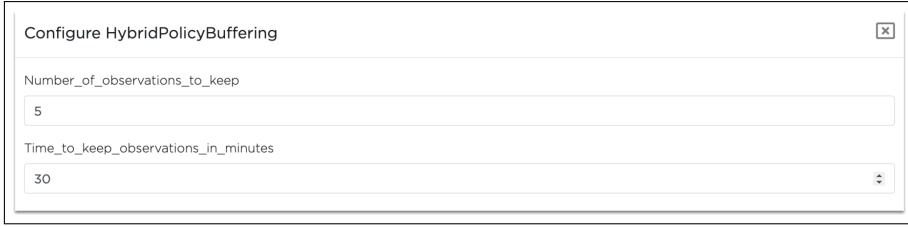
the buffer is pushed to the server. The developer can specify the lease window as a configuration for the component, as shown by Figure 4.7.



Figure 4.7: Window for configuring the *time-based* offline accessibility component. The configuration specifies the maximum time to keep records in memory, i.e., the lease window. The time is specified in minutes.

*Hybrid offline accessibility component.* This component combines the lease window and the number of records to keep when the network fails. The developer can configure both parameters as shown by Figure 4.8. When data arrives on its input port, and there is a connection, it is immediately forwarded to the output port. When a disconnection happens, data arriving on the input port is stored in a buffer in memory. When the buffer gets full, old records are removed, leaving only the specified number of most recent records. The records are kept in the buffer until the lease window expires, and all records are removed from the buffer.

Previous work, e.g., DisCoPar, has already explored on-disk data storage and database policies for offline accessibility. Therefore, the above components are based on in-memory storage policies for offline accessibility. The components give application developers more flexibility in specifying different policies to store data when network connections fail. In



The screenshot shows a window titled "Configure HybridPolicyBuffering" with a close button in the top right corner. Inside the window, there are two text input fields. The first field is labeled "Number\_of\_observations\_to\_keep" and contains the number "5". The second field is labeled "Time\_to\_keep\_observations\_in\_minutes" and contains the number "30".

Figure 4.8: Window for configuring the *hybrid-based* offline accessibility component. Time is specified in minutes, and the number of records to keep is numerical.

contrast to the state-of-the-art LCDEs, DisCoPar-Kilimo specifies more policies and offers a broader portfolio of offline accessibility components.

#### 4.3.4.3 Handling Partial Failures at the Edge

The policies offered at the edge are different than those for mobile phones since the networking or device malfunctioning assumptions for microcontrollers are different than those for mobile devices, such as phones. First, we still assume that the network connections, as with mobile devices, can become unavailable due to the quality of service in developing areas. Second, we assume that edge devices can fail due to drained batteries or mechanical malfunction than mobile phones. Hence, storing data in long-term memory (i.e., on disk) and restoring it after a device failure should also be possible. To account for those failures, we propose two policies that DisCoPar-Kilimo uses to offer offline accessibility at the edge, i.e., (1) in-memory buffering and (2) on-disk storage.

DisCoPar-Kilimo implements the above two policies into two concrete edge components to store data at the edge when the network becomes unavailable: the *BufferData* and *BufferOnDisk* components.

*BufferData component:* This component works analogously to its mobile counterpart, the record-based offline accessibility component. The *BufferData* component specifies the upper limit of records to be kept. The number of records can be specified as a configuration setting for the component as shown in Figure 4.9. When the network connection becomes unavailable, input records are stored in a buffer in memory until the buffer reaches its upper limit. At this point,

old records are removed to create space for recent data.

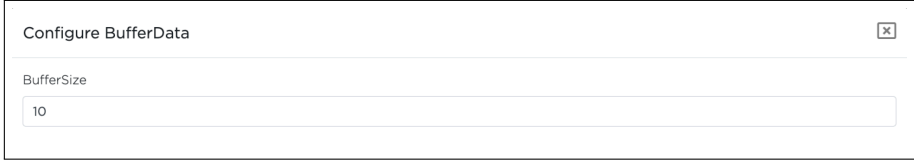


Figure 4.9: Configuration window for the *BufferData* component to specify the number of records to store in a buffer in memory.

Figure 4.10 shows an example application flow graph using the *BufferData* component to keep the latest readings of a soil moisture sensor during disconnections. The role of the *UnWrap* component is crucial to picking the soil moisture’s numerical values from the messages from the edge to the mobile device. The component sends the numerical soil moisture values to the *DisplayOnScreen* component for display on the mobile.

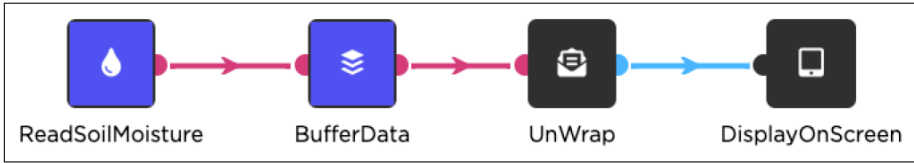


Figure 4.10: Flow-graph for *in-memory* buffering at the edge when the network connection becomes unavailable.

*BufferOnDisk component:* This component is based on the on-disk storage policy. A buffer is stored on a disk to keep records when networks fail. Therefore, old records are removed when the buffer becomes full to create space for recent data. When the network becomes available, data is read from the buffer and sent to the mobile phone. Figure 4.11 shows an example application flow-graph using the *BufferOnDisk* component.

#### 4.3.4.4 Validating Graphs with Offline Accessibility Components

Remember from Section 3.2.3 that in DisCoPar, it is assumed that the application developers know where the offline accessibility component should

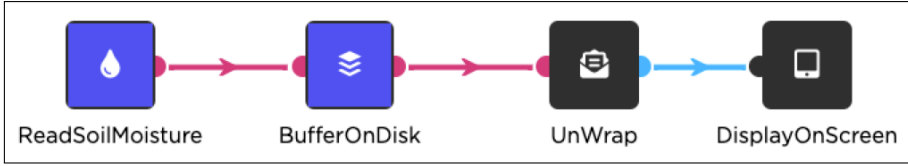


Figure 4.11: Flow-graph for *on-disk* data buffering at the edge when the network connection becomes unavailable.

be placed when composing application flow graphs. Also, recall that components emit and accept specific data types, e.g., the offline accessibility components that execute on the mobile scope emit and receive *observations*. However, developers can compose incorrect application graphs by connecting two or more successive downstream offline accessibility components or to other mobile components that accept *observations* consecutively.

To avoid incorrect application flow graphs, DisCoPar-Kilimo ensures that the offline accessibility components that execute on the mobile device are the last ones in the chain of successive downstream components to connect to the server side whenever they are used in an application flow graph. This ensures 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, i.e., another mobile component cannot be placed between the offline accessibility component and the server-side component. For instance, consider Figure 4.12 in which *InDatabaseBuffering* component that supports offline accessibility in the application can only be connected to the *Observation-Database* component that runs on the server side. The connection marked with a red colour between the *InDatabaseBuffering* component and the *BytePolicyBuffering* component cannot happen since both components are offline accessibility components. The *BytePolicyBuffering* component is an instance of the memory-based offline accessibility component presented in Section 4.3.4.2. Similarly, while implementing edge applications, the buffering component on the edge should be the last to connect to the downstream mobile side.

To ensure correct usage of offline components in DisCoPar-Kilimo applications, offline accessibility components internally specify a *type* property that is assigned a value “*Offline*”. We use the *type* property and the

### 4.3. DISCOPAR-KILIMO VISUAL PROGRAMMING ENVIRONMENT

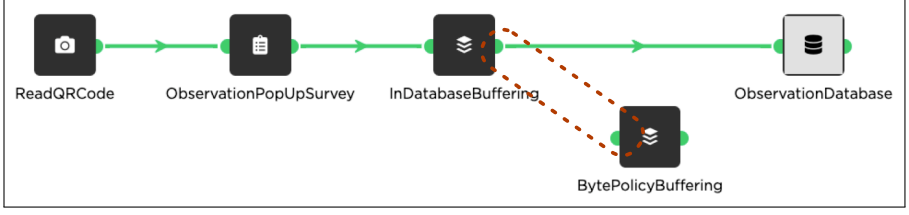


Figure 4.12: Flow-graph for connecting an offline accessibility component to the server side. The offline accessibility component cannot connect to another downstream offline accessibility or mobile component successively.

component scope to check and validate each connection to a downstream component. The connection is not allowed during application design if the downstream component has either the *type* property with a value “*Offline*” or the mobile scope. This deters application designers from making errors when composing application flow graphs on the canvas. DisCoPar-Kilimo uses the *type* property and mobile scope to validate application flow graphs because some mobile scope components accept *any* data type on their input ports. The offline accessibility components execute on the mobile scope; hence, DisCoPar-Kilimo uses the *offline* property to distinguish them from the other mobile scope components.

#### 4.3.5 Tracking Connected Edge Devices

As motivated by the driver scenarios in Section 2.2, it is sometimes necessary to have a global overview of the environment (e.g., a farm) by installing multiple microcontrollers (edge devices) with sensors attached to them. The installed edge devices can go offline due to battery drains or mechanical damage. Therefore, it is essential to monitor and track the connected edge devices. The monitoring and tracking of connected edge devices can give farmers general information about the sensing status. We refer to this general information on the connectivity of devices as “tracking edge devices”.

DisCoPar-Kilimo incorporates a connectivity tracking mechanism as illustrated by the flow chart in Figure 4.13 to track the edge devices. The input to the connectivity tracking mechanism is a uniform resource locator (URL) and event from the edge devices sent via conventional networking



facilities (e.g., SocketIO<sup>2</sup>). The URL contains information that shows the type of device from which it originates, e.g., ESP32 and the socket event. Components on mobile devices then use this information to subscribe and listen to events on the sockets from edge devices.

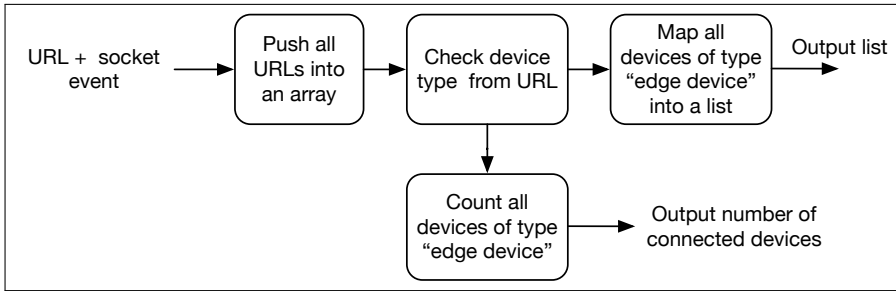


Figure 4.13: Flow chart for tracking the number of connected edge devices. The input to the flow chart is an event with details for the type of edge devices.

DisCoPar-Kilimo implements a concrete *ConnectedDevices* component for tracking the connected devices as illustrated by Figure 4.14. The component subscribes to the socket event mentioned previously and sends the information it receives on connected devices to the *DataArrayToTable* component. The role of the *DataArrayToTable* component is to convert the received data into a dataset for display as a list using the *ListConnectedDevices* component. The component also outputs the numerical count of the connected devices for display using the *NumberOfConnectedDevices* component. The numerical count is helpful to give a quick indication of the number of connected devices. The *ListConnectedDevices* component is an instance of the existing *DisplayInTable* component, while the *NumberOfConnectedDevices* component is an instance of the existing *DisplaOnScreen* component.

The *ConnectedDevices* component allows developers to track connected edge devices that we explain in Section 5.2.7.

### 4.3.6 Accumulating Data from Multiple Edge Devices

Recall from the scenarios in Section 2.2 that multiple edge devices can be installed to give a better overview of the farm conditions. DisCoPar-

<sup>2</sup><https://socket.io/>

### 4.3. DISCOPAR-KILIMO VISUAL PROGRAMMING ENVIRONMENT

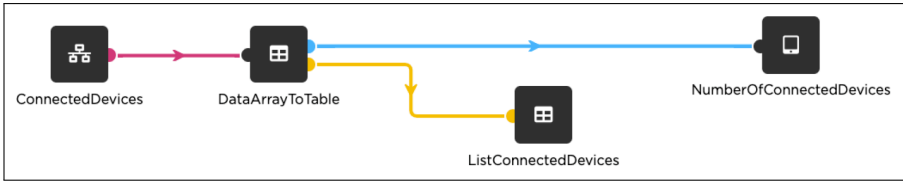


Figure 4.14: Example application for tracking connected devices. The application is composed of four mobile components.

Kilimo offers a dedicated component to accumulate data from multiple edge devices on the phone. Developers can configure this component to specify the number of devices to accumulate data as shown in Figure 4.15.

Configure GatherMoistureReadings

Number

5

Figure 4.15: Configuration window for specifying the number of edge devices accumulating soil moisture in the *GatherMoistureReadings* component. The number of devices is specified as a numerical value.

Figure 4.16 illustrates how the accumulation component works. The accumulator component receives the number of devices to accumulate data from and the actual data from the sensing component as input. Each data received is mapped to the identifier (ID) of the sending device. This accumulation mechanism outputs a map of device IDs, their payloads, and a numerical array of the latest values from the edge devices. The accumulated data can be filtered on the mobile phone to get the specific payloads for each edge device.

Figure 4.17 shows an example application for accumulating soil moisture from multiple devices. This example accumulates data using the *GatherMoistureReadings* component on the mobile phone. The number of devices to accumulate data is specified as a configuration for the *GatherMoistureReadings* component as shown in Figure 4.15. The configured number of devices must be met before the data accumulation. The *GatherMoistureReadings* component accumulates the data internally. Then, it sends a map of device IDs, their payloads, and an array of numerical values on its output ports to the subsequent components. The aver-

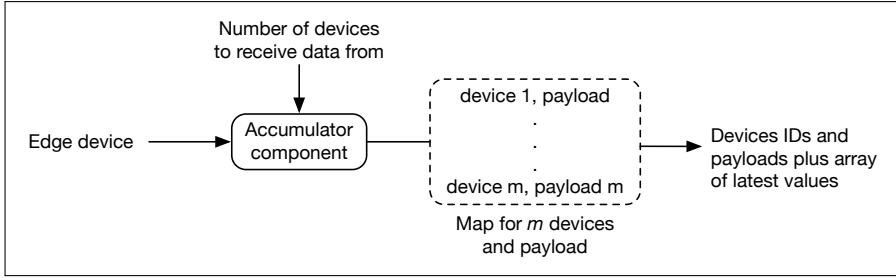


Figure 4.16: Flow chart for accumulating data from multiple edge devices. The input to the process is a message from each edge device.

age for the accumulated readings is computed using the *ComputeAverage* component using the data it receives from the *GatherMoistureReadings* component. The *ComputeMaximum* and *ComputeMinimum* components determine the maximum and minimum values of the accumulated data, respectively.

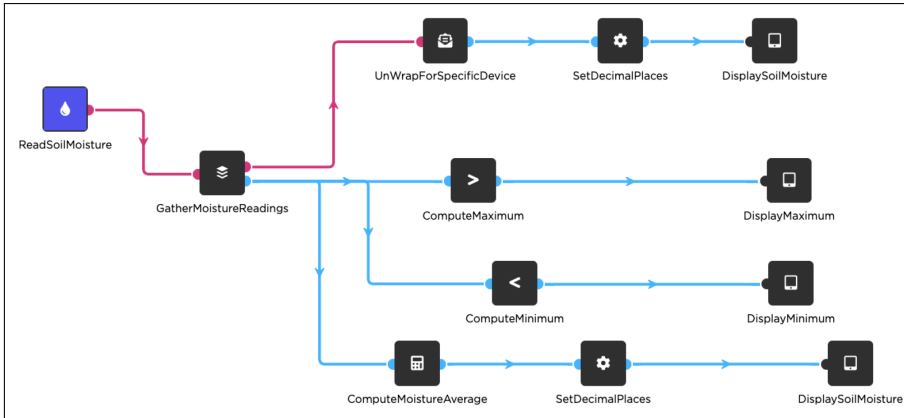


Figure 4.17: Flow-graph for an example application for accumulating soil moisture data. The application is composed of one edge component and several mobile components.

DisCoPar-Kilimo incorporates two components for filtering data from the edge devices, i.e., the *UnWrap* and *UnWrapForSpecificDevice* components. Previously, we have explained how the *UnWrap* component filters data from edge devices to get the general payload in Section 4.3.4.3. The *UnWrap* component filters the payload for all messages from the edge

devices. Therefore, it cannot serve in the context of accumulating data from multiple edge devices, and the payload for each edge device needs to be filtered. In this example, we use and explain how the *UnWrapForSpecificDevice* component filters data for specific edge devices. For the filtering to happen, the identity of the specific device has to be specified by the developer in the configuration of the component as illustrated by Figure 4.18. Multiple *UnWrapForSpecificDevice* components can be used depending on the number of edge devices considered for filtering. In such a case, each *UnWrapForSpecificDevice* component is used to filter data for one edge device.



Figure 4.18: Window for configuring the *UnWrapForSpecificDevice* component to specify the edge device for which to filter data.

## 4.4 Developing and Deploying Applications by Example

We now detail how to design and deploy an application in DisCoPar-Kilimo. Figure 4.19 illustrates the different stages of creating an application in DisCoPar-Kilimo. Recall that DisCoPar-Kilimo’s VPE is hosted on a server. Hence, it can be accessed on a web interface. Therefore, everything begins by accessing the installed and running instance of DisCoPar-Kilimo on a browser and clicking on the part labelled 1 (platform) in Figure 4.19a to access the platform. A new application can be created in the platform by clicking the part labelled 2 in Figure 4.19b and specifying the general application details in the fields shown in Figure 4.19c. Lastly, the application is created by clicking the “*Create Application*” button. After creating the application, one can access the application builder interface by clicking the button labelled ‘4’ in Figure 4.19d. The application builder interface (shown in Figure 4.19e) allows dragging, dropping and connecting components on the VPE to compose application flow graphs. Figure 4.19e depicts a sample application graph that we explain in the

following subsection.

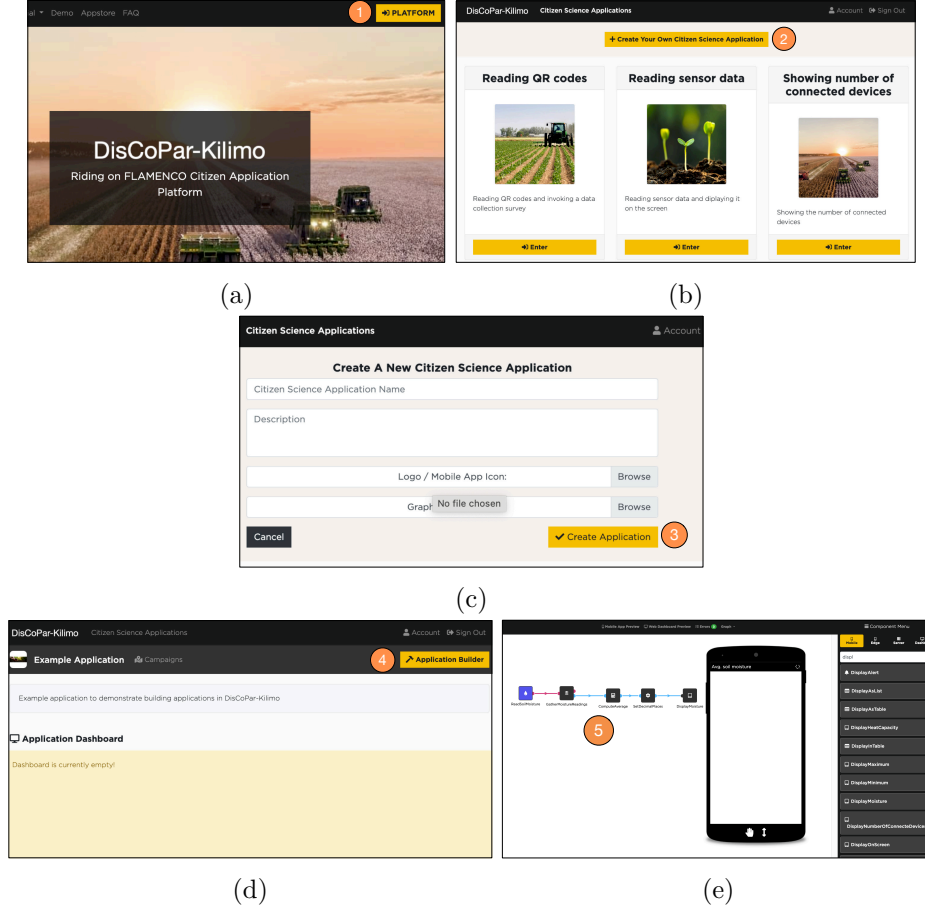


Figure 4.19: Steps for creating an application in DisCoPar-Kilimo. (a) Accessing DisCoPar-Kilimo platform, (b) Creating an application, (c) Specifying general application details, (d) Accessing the application builder VPE and (e) Complete application flow-graph on the VPE.

#### 4.4.1 Example application

To illustrate how to build and deploy applications in DisCoPar-Kilimo, consider an application that measures and monitors soil moisture levels in the farm as explained in the driver smart agriculture scenarios in Chapter 2. The application accumulates the readings from several devices and

#### 4.4. DEVELOPING AND DEPLOYING APPLICATIONS BY EXAMPLE

---

computes the average on the mobile client. The moisture levels are read using a soil moisture sensor attached via serial connection to an edge device, e.g., M5StickC device, as shown in Figure 4.20.



Figure 4.20: Deployed edge devices for a DisCoPar-Kilimo application for monitoring soil moisture.

When the readings are received on the mobile phone, the application displays the soil moisture values on the screen. The above application has four functional requirements:

1. Read soil moisture levels.
2. Accumulate the soil moisture readings on the mobile phone.
3. Compute the average soil moisture on the mobile phone.
4. Display the average soil moisture on a mobile phone.

Computationally, the application executes on two scopes: the edge scope and the mobile scope. The flow graph for this application is represented in the visual programming environment as shown in Figure 4.21.

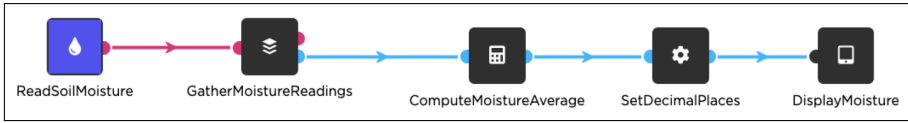


Figure 4.21: Example flow-graph of an application for monitoring soil moisture in DisCoPar-Kilimo. The application is composed of one edge component and four mobile components.

This application reads the soil moisture using the *ReadSoilMoisture* component and accumulates the readings on the mobile phone using the *GatherMoistureReadings* component. The application computes the average soil moisture using the *ComputeMoistureAverage* component. Lastly, the application displays the average soil moisture on the mobile phone using the *DisplayMoisture* component. The *SetDecimalPlaces* component is used to specify the number of decimal places for the soil moisture values displayed on the mobile phone.

#### 4.4.2 Deploying the example application

Now that we have the composed application flow graph, we need to deploy it to the mobile phone and the edge device with the soil moisture sensor attached. First, executing the flow graph on the mobile phone requires the base and application graphs. The base application for the mobile is built, exported in JavaScript format, and bundled with the client application. The base application contains the infrastructure and machinery for all DisCoPar-Kilimo components. The mobile application is exported in JSON format and packaged using Cordova<sup>3</sup> for the target device, e.g., an Android device. Second, executing the flow graph on edge devices requires the edge graph. The edge graph contains the calls to functions invoked when the edge components execute. Both base applications can be provided via web link(s) where they can be downloaded. The application graphs can be downloaded from the canvas as shown by Figure 4.22. Deployment details are explained in Chapter 5.

---

<sup>3</sup><https://cordova.apache.org/>

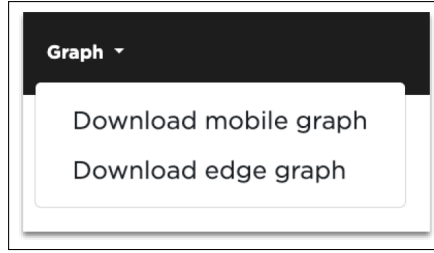


Figure 4.22: Exporting the mobile and edge application flow graphs in DisCoPar-Kilimo.

## 4.5 Extensions to DisCoPar

As previously mentioned in Section 4.2, DisCoPar-Kilimo builds on DisCoPar. In this section, we detail how DisCoPar-Kilimo extends DisCoPar as follows.

*New execution scope:* DisCoPar-Kilimo extends the architecture of DisCoPar with a new execution scope. The new scope hosts components that support computation at the edge and environment sensing. As a result, DisCoPar-Kilimo features infrastructure for extracting the edge graph from an entire application graph and deploying and executing it at the edge devices.

*Policies to handle partial failures:* DisCoPar-Kilimo introduces different policies to address partial failures geared to smart agriculture applications. This entailed extending the existing DisCoPar policies for handling partial failures on mobile devices and adding new ones for edge devices.

*Tracking connected edge devices:* An essential aspect of handling partial failures is knowing when devices go online and offline. In this regard, DisCoPar-Kilimo introduces a mechanism and component for tracking connected devices. This is important to provide information on devices requiring repair and maintenance.

*Accumulating data from edge devices:* DisCoPar-Kilimo provides an accumulator component for accumulating data from several edge devices on the mobile device. Remember from Section 2.2 that several edge devices can be deployed in large farms.

*Coordination with the edge:* DisCoPar-Kilimo introduces a coordination



mechanism to allow edge components to communicate directly with the mobile components. Hence, it introduces the infrastructure that transparently provides coordination to application developers.

## 4.6 Conclusion

In this chapter, we introduce our approach for a low-code development environment to simplify the development of SAAs. We design and incorporate components in a concept LCDE, DisCoPar-Kilimo, that supports (1) dedicated environment sensing, (2) computation at the edge, (3) coordination with the edge and lastly, (4) handling partial failures. Computation at the edge is essential for microcontrollers to process data near the source before sending it to mobile devices. Keeping data at the edge is essential to avoid losing it when the network connection becomes unavailable. Lastly, support for coordination with the edge is essential to enable edge devices to communicate directly with the components on a mobile device without going through a centralised server. The above features have been designed to simplify the development of SAAs.

The features are implemented into DisCoPar-Kilimo's architectural model, which has four execution scopes: the edge, mobile, server, and web scopes. Each category hosts components that can be used to construct smart agriculture applications. In the next chapter, we present and discuss the implementation of DisCoPar-Kilimo.

## Chapter 5

# DisCoPar-Kilimo Implementation

In the previous chapter, we introduced DisCoPar-Kilimo, the low-code development environment presented in this dissertation. This chapter aims to describe the implementation of DisCoPar-Kilimo to make it reproducible. In particular, the chapter focuses on the implementation details of each feature that DisCoPar-Kilimo devised to satisfy the properties identified in Chapter 2. Lastly, the chapter discusses the design choices adopted and challenges faced during the implementation.

### 5.1 Basic Building Blocks

DisCoPar-Kilimo is built on DisCoPar (cfr. Chapter 3). DisCoPar is implemented using Node.js<sup>1</sup> and JavaScript. This section outlines the internal details of DisCoPar to explain the implementations of our contributions.

#### 5.1.1 Application Graphs

Recall from Section 3.2 that in DisCoPar, applications are directed acyclic graphs composed of connected components. For execution, the application flow graph (network of dependent computations) is represented as a JSON file (see Appendix A). The JSON file stores the application graph in a

---

<sup>1</sup><https://nodejs.org/en/>

text format for transporting data. Therefore, the JSON file represents the application graph as an object. Components and their connections are nested objects of the application graph object. Each nested *component object* denotes a node in the application graph. Similarly, each nested *connection object* denotes a link in the application graph. The connection objects show the source output ports and the destination input ports, channels through which data can flow from one component to another. The *component object* contains the state and scope of the node.

### 5.1.2 Executing Application Graphs

The JSON file representing the application graph is loaded onto the execution engine of DisCoPar for execution. Execution is accomplished by initialising the components, loading their source code from the component layer, and establishing the real-time data streams between the newly created processes. Due to DisCoPar’s reactive flow-based nature, each component process activates automatically based on data availability in its incoming data streams. State changes are automatically and efficiently propagated across the application flow graph by the underlying execution engine.

### 5.1.3 Basic Application Example

To explain the basic building blocks in DisCoPar, let’s consider a simple application consisting of a component that transmits data received on its input port and sends a message to an upstream component each time it receives data. Figure 5.1 illustrates the *TransmitData* component in use.

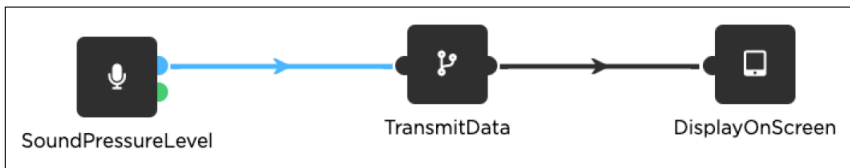


Figure 5.1: Simple application showing a component named *TransmitData* that transmits data received on its output port.

In the example above, the *TransmitData* component receives sound pressure measurements from the *SoundPressureLevel* component and sends

them on its output port to the *DisplayOnScreen* component. The *TransmitData* component then sends back an acknowledgement message to the *SoundPressureLevel* component for each measurement received.

#### 5.1.4 Implementing Components in DisCoPar

Conventionally, DisCoPar components are implemented using JavaScript classes before making them available as visual blocks. Therefore, Listing 5.1 illustrates the conventional implementation of the *TransmitData* component in Figure 5.1. The component is called *TransmitData* and extends the *Component* class generated by DisCoPar. The *Component* class provides the blueprint for components.

```

1 class TransmitData extends Component {
2   constructor() {
3     super();
4     this.description = "Transmit data received on its output port";
5     this.icon = "fas fa-code-branch";
6     this.scope = Globals.Scope.Mobile;
7   }
8   initialise() {
9     this.inPorts.add("in", {type: Globals.DataType.ALL});
10    this.outPorts.add("out", {type: Globals.DataType.ALL});
11  }
12  execute(component) {
13    component.inPorts.in.on("data", (input, connection) => {
14      component.outPorts.out.send(input);
15      connection.acknowledge(input)
16    });
17  }
18 }

```

Listing 5.1: Implementation of the basic building block of applications in DisCoPar.

Each component in DisCoPar has a *name* picked by the end user, which is expected to describe the computation task it performs. DisCoPar uses the component name for a JavaScript class representing the component and is stored in the component palette of DisCoPar. Internally, each component has three main parts that include the *constructor* method, *initialise* method and lastly, the *execute* method.

*Constructor method:* The constructor is a unique method for creating an instance of the component and initialising the default state. The default state of a component includes (1) the description of the component given by the user in the canvas, (2) the component icon to display on the canvas and (3) the execution scope of the component.

*Initialise method:* Recall from Section 3.2.1.3 that components have input and output ports to exchange data and messages. Component

ports are specified within the *initialise()* method (lines 9 and 10 of Listing 5.1) using the *add()* method. The *add()* method takes the component's name and its type as arguments. In this example, the ports are named “*in*” and “*out*” respectively. Remember from Section 3.2.1.4 that the component *port type* specifies the kind of data the port accepts (for input ports) or emits (for output ports). The *port type* is specified using a JavaScript object `{type: Globals.DataType.«specific_data_type»}`. Both ports in Listing 5.1 are of type *ALL*, which represents the *Any* type. Lastly, the *initialise* method is used to specify the concrete state of a component, which can be given by “configure”. In this case, no state is passed.

*Execute method:* DisCoPar asks a component to perform some computation using the *execute()* method. The method receives as input an object containing meta-data about the component and a function containing instructions on performing the intended computation. The method adds a listener for events on the input port. Line 13 of Listing 5.1 listens for events on the input port of the component using the *on()* method. When data arrives on the input port, callback (*input*, *connection*) is applied. The callback receives as arguments (1) a channel to receive data and send it to the next *downstream component* and (2) an optional channel to send back acknowledgement messages to *upstream components* e.g., confirmatory messages when data is saved into a database successfully. In this example callback, *input* is a channel to receive data and send it to the next *downstream component* while *connection* is the channel to send back acknowledgement messages to *upstream components*. In Listing 5.1, lines 13 – 16 of the component simply forward to its output port the data received on its input port. DisCoPar uses web sockets (i.e., Socket-IO<sup>2</sup>) for communication via component ports. In this example, the component listens on its input port for incoming data as the *payload*. Line 14 sends the data received on the input port to the output port using the *send()* method. Line 15 emits an acknowledgement event to upstream components. Lastly, component output ports emit events to which the input ports of connected components can subscribe. In this example, the *TransmitData* subscribes and listens to a “*data*” event emitted by the upstream component on

---

<sup>2</sup><https://socket.io/>

line 13.

In DisCoPar, there are two kinds of components: (1) components that perform computational tasks and (2) that visualise and display the results of the computational tasks. The components that visualise and display computation can be used to build graphical user interfaces (GUIs) for applications. Each component has a unique *identifier* generated when the component is published to a component library in the canvas.

### 5.1.5 Distributed Connections

As mentioned in Chapter 3, components can belong to three different scopes as shown in Listing 5.2, i.e., server, mobile and web scope.

```
1 Globals.Scope = {  
2   Server: new ComponentScope.Server(),  
3   Mobile: new ComponentScope.Mobile(),  
4   Web: new ComponentScope.Web(),  
5 };
```

Listing 5.2: Component execution scopes in DisCoPar. It features the server, mobile and web component execution scopes.

Connecting the output ports of mobile components to the input ports of server or web components creates a distributed connection. Distributed connections are implemented using Socket.IO. The Socket.IO library includes additional features, such as auto-reconnection after network connectivity has been restored and network disconnection detection. The library provides server-side and client-side components with similar APIs such that both the client and the server can emit events and subscribe to events in a *publish-subscribe* manner.

## 5.2 DisCoPar-Kilimo

In this section, we explain in detail the implementation of the DisCoPar-Kilimo extensions. We focus our explanations on the features DisCoPar-Kilimo offers to support the properties for smart agriculture applications identified in Chapter 2.

### 5.2.1 Computation at the Edge

As explained in Section 4.2, DisCoPar-Kilimo introduces components that can run computations at the edge. Those components form the *edge scope*.

To provide a new scope, two changes were added to DisCoPar. First, a new class was added for the edge scope as shown in Listing 5.3.

```

1 class Edge extends Client {
2   constructor() {
3     super();
4   }
5 }

```

Listing 5.3: Defining the edge scope. The edge scope is defined as a class that extends the *Client* class.

The new class is modelled as a subclass of the *Client* class since the components that execute at the edge scope eventually communicate with the DisCoPar-Kilimo server. The *Client* class provides the functionality to specify where components can be deployed. Second, as shown in Listing 5.4, the new edge scope needs to be published in the global object to make it appear as a category on the component menu in the visual programming environment of DisCoPar-Kilimo. Publishing the new scope globally allows it to specify component categories while defining components that will eventually be hosted on it. Hence, all components that execute at the edge are hosted within this scope.

```

1 Globals.Scope = {
2   Server: new ComponentScope.Server(),
3   Mobile: new ComponentScope.Mobile(),
4   Web: new ComponentScope.Web(),
5   Edge: new ComponentScope.Edge(),
6 };

```

Listing 5.4: Adding the *edge* scope to the global object. This allows it to be globally accessible in DisCoPar-Kilimo.

### 5.2.1.1 Runtime Environment at the Edge

At the edge, we use Arduino<sup>3</sup> and Duktape<sup>4</sup> to support executing the parts of an application graph portion running on edge devices, e.g., microcontrollers. We refer to this application running on edge devices as the ‘edge application’. The edge application consists of a base application implemented in C/C++/Arduino and a JavaScript application generated from the edge components connected in the DisCoPar-Kilimo visual programming environment. We use Duktape to run JavaScript applications on microcontrollers, e.g., the ESP32 family. Duktape is a lightweight

---

<sup>3</sup><https://www.arduino.cc/en/software>

<sup>4</sup><https://duktape.org/>

JavaScript engine that can run on microcontrollers. The engine allows JavaScript programs to call functions implemented in C/C++. This enables deploying and executing JavaScript code on the edge devices which interact with sensors and peripheral devices using Arduino.

We now describe the key ideas about implementing the built-in edge components. The edge components fall into three main categories: (1) environment sensing components, (2) edge computation components and (3) offline accessibility components to handle partial failures.

### 5.2.2 Environment Sensing

DisCoPar-Kilimo provides three built-in sensing components named *ReadSoilMoisture*, *ReadTemperature* and *ReadHumidity* components to read soil moisture, air temperature and humidity, respectively. All the sensing components at the edge have no input ports – they have only the output port(s). They are source components in the application graphs that can be used to produce sensor data.

Listing 5.5 shows the implementation of the *ReadSoilMoisture* component. Its scope is as defined in line 6. The component has one output port named ‘out’ and emits data of type “EDGE”. The “EDGE” data type refers to data coming from edge devices in JSON format.

```

1 class ReadSoilMoisture extends Component {
2   constructor() {
3     super();
4     this.description = 'Reads soil moisture at the edge.';
5     this.icon = 'fas fa-tint';
6     this.scope = Globals.Scope.Edge;
7   }
8   initialise(graph) {
9     this.outPorts.add('out', {type: Globals.DataType.EDGE});
10  }
11 }

```

Listing 5.5: Implementation of the component for reading soil moisture at the edge.

#### 5.2.2.1 Executing edge components

Recall from Chapter 4 that DisCoPar-Kilimo incorporates companion and friend functions. Therefore, the edge components do not define an execute method as other regular components. Instead, each edge component has a companion function invoked to perform the component’s task at the edge. Remember from Section 5.1 that an application is represented as a



JSON file. The JSON file is processed to extract the companion function calls representing the edge application and executed at the edge devices. Companion functions that require interaction with the hardware, in turn, call friend functions.

Listing 5.6 shows the function calls for the application example in Section 4.3.1 (Figure 4.3). The code defines a *discoparKilimoLoop* that creates a variable that calls the *ReadSoilMoisture* companion function in line 2. The *discoparKilimoLoop* function is the entry point for all executions at the edge devices. In this code example, the variable assigned the call to the *ReadSoilMoisture* companion function is passed as an argument to the *SendData* function. The *discoparKilimoLoop* is executed within the Arduino loop in the edge devices. The *SendData* is a special function defined to send data to the network as shown by Listing 5.7. The function takes the connection between the edge and mobile component as an argument and the data to send over the network. When executed, it invokes its friend function, the *sendDataToNetwork* in line 6.

```
1 function discoparKilimoLoop(){
2   var _92e4f2d1cc5b465bac81b93a3690f37a = ReadSoilMoisture();
3   SendData("data:92e4f2d1-cc5b-465b-ac81-b93a3690f37a_out_0eccc2b0-c0ec-49ac-d273-17e6a55e5677_in",
4     _92e4f2d1cc5b465bac81b93a3690f37a);
5 }
```

Listing 5.6: Example exported edge graph showing function calls. This example calls the *ReadSoilMoisture* companion function and passes its computation results to the *SendData* function.

```
1 function SendData(conn, payload) {
2   var send_to_network = "";
3   send_to_network += conn;
4   send_to_network += "#";
5   send_to_network += payload;
6   sendDataToNetwork(send_to_network);
7 }
```

Listing 5.7: Implementation of the edge function to interact with the network. The function calls a low-level function *sendDataToNetwork*.

**Example:** In the application shown in Figure 4.3 (Chapter 4), the *ReadSoilMoisture* companion function shown in Listing 5.8 is invoked to sense soil moisture. In this example, the companion function requires interaction with the hardware to read soil moisture levels. Therefore, the companion function further invokes the friend function, *readSoilMoistureFromSensor()*, in line 2 of Listing 5.8.

```

1 function ReadSoilMoisture() {
2     var soilM = readSoilMoistureFromSensor();
3     return soilM;
4 }

```

Listing 5.8: Implementation of the companion function of the *ReadSoilMoisture* component. The function assumes the same name as the component.

The friend functions are low-level functions implemented in C/C++ that interact directly with the hardware. They perform low-level operations, such as reading values using a sensor and returning those values to the calling companion function. The *readSoilMoistureFromSensor* friend function to the *ReadSoilMoisture* companion function is shown in Listing 5.9. The function reads soil moisture in line 6, adds it into an array and computes the average of several measurements in line 9. The number of measurements pushed into the array is specified as the array size in line 2. The average handles erroneous sensor readings [GTTSBR<sup>+</sup>19]. In line 10, the average soil moisture is spread over a 0 – 100 range. The spread helps limit the soil moisture values within the 0 – 100 range. The soil moisture readings are then pushed to the Duktape stack in line 11.

```

1 static duk_ret_t native_readSoilMoistureFromSensor(duk_context *ctx) {
2     float sm_array[WINDOW_SIZE];
3     float sum = 0.0;
4     int i;
5     for (i = 0; i < (sizeof(sm_array)/sizeof(sm_array[0])); i++) {
6         sm_array[i] = analogRead(33);
7         sum = sum + sm_array[i];
8     }
9     float avg_sm = (float)sum/(sizeof(sm_array)/sizeof(sm_array[0]));
10    avg_sm = map(avg_sm, 4095, 0, 0.00, 100);
11    duk_push_number(ctx, avg_sm);
12    memset(sm_array, 0, sizeof(sm_array));
13    return 1; /* one return value */
14 }

```

Listing 5.9: Implementation of the friend function of the *ReadSoilMoisture* companion function.

The rest of the sensing components, such as the *ReadTemperature* and *ReadHumidity* components, follow a similar implementation strategy to the *ReadSoilMoisture* component. Appendix B lists all the implemented companion functions.

### 5.2.3 Components for Computation at the Edge

To support computation using sensing data at the edge, DisCoPar-Kilimo implements several components that can be used to perform arithmetic op-

erations, e.g., addition, subtraction, multiplication and division. To illustrate the implementation of those components, let us consider computing temperature differences ( $\Delta T = T_2 - T_1$ ). This computation requires the current temperature ( $T_2$ ) at time  $t$  and previous temperatures ( $T_1$ ) at time  $t - 1$ , where  $t$  is a logical time. Therefore, DisCoPar-Kilimo implemented a *PreviousValue* component shown in Listing 5.10. The component has one input port and one output port. This component can be connected to a *ReadTemperature* component to get the previous temperature reading value at  $t - 1$ . The temperature change is computed using the *Subtraction* component shown in Listing 5.11. The component has two input ports and one output port.

```
1 class PreviousValue extends Component {
2   /* more code */
3   initialise(graph) {
4     this.inPorts.add('in', {type: Globals.DataType.EDGE});
5     this.outPorts.add('out', {type: Globals.DataType.EDGE});
6   }
7 }
```

Listing 5.10: Implementation of the component for keeping previous values.

```
1 class Subtraction extends Component {
2   /* more code */
3   initialise(graph) {
4     this.inPorts.add('first', {type: Globals.DataType.EDGE});
5     this.inPorts.add('second', {type: Globals.DataType.EDGE});
6     this.outPorts.add('out', {type: Globals.DataType.EDGE});
7   }
8 }
```

Listing 5.11: Implementation of the *Subtraction* component that finds the difference between two numerical values.

Recall that the edge components do not provide the `execute` method because their companion and friend functions are invoked on the edge devices instead. Listing 5.12 shows the companion function of the *PreviousValue* component. The edge flow graph implements the temperature difference using the *Subtraction* arithmetic operator components. The *Subtract* companion function receives two input values and computes the difference in line 2 of Listing 5.13.

## 5.2.4 Handling Partial Failures on the Mobile Scope

DisCoPar-Kilimo introduces more policies for handling partial failures and implements them into concrete components. In addition, DisCoPar-Kilimo implements policies for the application flow graph to ensure that

```

1 var preValue = null;
2 function PreviousValue(current){
3     var temp = preValue;
4     preValue = current;
5     return temp;
6 }

```

Listing 5.12: Implementation of the companion function of the *PreviousValue* component.

```

1 function Subtract(a, b){
2     return a - b;
3 }

```

Listing 5.13: Implementation of the companion function of the *Subtraction* edge component.

offline accessibility components cannot be connected successively or to another downstream mobile component. DisCoPar-Kilimo proposes and implements the *OfflineComponent* blueprint to generalise handling partial failures. Therefore, each offline accessibility component extends the *OfflineComponent* class (Listing 5.14). The *OfflineComponent* class defines the “\$type” property with a value “*Offline*” that can be used as a flag to validate application graphs as we explain later in Section 5.2.5.

```

1 class OfflineComponent extends Component {
2     constructor() {
3         this.$type = "Offline";
4     }
5 }

```

Listing 5.14: Implementation of the blue-print for the offline accessibility components.

#### 5.2.4.1 Mobile Components for Handling Partial Failures

As mentioned previously in Section 4.3.4, DisCoPar-Kilimo proposes different offline accessibility policies. Each of the policies is implemented into a concrete component that we describe below.

**Record-based offline accessibility component:** Listing 5.15 shows the implementation of the record-based offline accessibility component. The number of records to keep is specified as a component configuration in line 12. The component defines a flag (i.e., «*this.isConnected*») that

it initially sets to *true* in line 7 and uses it to determine when to store data in a buffer. The buffer to store data records is defined in line 6. On disconnection (line 29), the component sets the «*this.isConnected*» flag to *false* in line 30. This makes the component execute lines 21 – 28, checking whether the buffer is full in line 23, removing some records from the buffer if it's full in line 24, and pushing data to the buffer in line 26. On reconnection (line 32), all records stored in the buffer are sent in bulk in line 33, the buffer is cleared in line 34, and «*this.isConnected*» flag is set to *true* in line 35.

```
1 class RecordPolicyBuffering extends OfflineComponent {
2   constructor() {
3     super();
4     this.scope = Globals.Scope.Mobile;
5     this.$type = "Offline";
6     this.buffer = [];
7     this.isConnected = true;
8   }
9   initialise(graph) {
10    this.inPorts.add("in", {type: Globals.DataType.OBSERVATION});
11    this.outPorts.add("out", {type: Globals.DataType.OBSERVATION});
12    this.settings.add("records_to_keep", 5, {
13      type: Globals.SettingType.Numerical
14    });
15  }
16  execute(component) {
17    const {settings: {records_to_keep},} = component;
18    component.inPorts.in.on('data', (data, connection) => {
19      if(this.isConnected){
20        component.outPorts.out.send(data);
21      } else {
22        let required_records = records_to_keep.value();
23        if (this.buffer.length > records_to_keep.value()) {
24          this.buffer.splice(0, this.buffer.length - required_records);
25        }
26        this.buffer.push(data);
27      }
28    });
29    component.outPorts.out.on("disconnect", () => {
30      this.isConnected = false;
31    });
32    component.outPorts.out.on("reconnect", () => {
33      component.outPorts.out.sendBulk(this.buffer);
34      this.buffer.splice(0, this.buffer.length);
35      this.isConnected = true;
36    });
37  }
38 }
```

Listing 5.15: Implementation of the record-based offline accessibility component.

**Memory-based offline accessibility component:** Listing 5.16 shows the implementation of the memory-based offline accessibility component. In line 12, the application designer can specify the maximum memory size

configured for data storage. The component defines a buffer to store data records in line 6. Also, the component defines «*this.isConnected*» flag in line 7 and uses it to determine in line 21 whether to send data when there is a connection in line 22 or store data in a buffer when there is a disconnection in lines 27.

```

1 class BytePolicyBuffering extends OfflineComponent {
2   constructor() {
3     super();
4     this.scope = Globals.Scope.Mobile;
5     this.$type = "Offline";
6     this.buffer = [];
7     this.isConnected = true;
8   }
9   initialise(graph) {
10    this.inPorts.add("in", {type: Globals.DataType.OBSERVATION});
11    this.outPorts.add("out", {type: Globals.DataType.OBSERVATION});
12    this.settings.add("max_memory_size", 4000, {
13      type: Globals.SettingType.Numerical
14    });
15  }
16  execute(component) {
17    const {settings: {max_memory_size},} = component;
18    let available_memory = memory_size_units(max_memory_size.value());
19    let used_memory = memory_size_units(calc_buffer_size(this.buffer));
20    component.inPorts.in.on("data", (data, connection) => {
21      if(this.isConnected){
22        component.outPorts.out.send(data);
23      } else {
24        if (used_memory > available_memory) {
25          this.buffer.splice(0, this.buffer.length);
26        }
27        this.buffer.push(data);
28      }
29    });
30    component.outPorts.out.on("disconnect", () => {
31      this.isConnected = false;
32    });
33    component.outPorts.out.on("reconnect", () => {
34      component.outPorts.out.sendBulk(this.buffer);
35      this.buffer.splice(0, this.buffer.length);
36      this.isConnected = true;
37    });
38
39    function calc_buffer_size(input) { ... }
40    function memory_size_units(input) { ... }
41  }
42 }

```

Listing 5.16: Implementation of the memory-based offline accessibility component.

Before storing data records in the buffer, the component checks the available memory in line 24 and removes data from the buffer in line 25 if the allocated memory is full. On disconnection (line 30), the component sets «*this.isConnected*» flag to *false* in line 31. Setting the flag to *false* prompts the execution to jump to line 23. On reconnection (line 33), all data records in the buffer are sent in bulk in line 34, the buffer is cleared

in line 35, and «*this.isConnected*» flag is set to *true* in line 36. Setting the flag to *true* prompts the execution to jump to line 21.

**Time-based offline accessibility component:** Listing 5.17 shows the implementation of the time-based offline accessibility component. The lease window in minutes to keep the data is specified as a component configuration in line 13. The component defines a time interval in line 6 that starts to count when a disconnection occurs in line 28, a buffer in line 7 and «*this.isConnected*» flag in line 8. The time counter is global and applies to all incoming data.

```
1 class TimePolicyBuffering extends OfflineComponent {
2   constructor() {
3     super();
4     this.scope = Globals.Scope.Mobile;
5     this.$type = "Offline";
6     this.time_interval = null;
7     this.buffer = [];
8     this.isConnected = true;
9   }
10  initialise(graph) {
11    this.inPorts.add("in", {type: Globals.DataType.OBSERVATION});
12    this.outPorts.add("out", {type: Globals.DataType.OBSERVATION});
13    this.settings.add("time_to_keep_data", 1, {
14      type: Globals.SettingType.Numerical
15    });
16  }
17  execute(component) {
18    const {settings: {time_to_keep_data},} = component;
19    component.inPorts.in.on("data", (data, connection) => {
20      if(this.isConnected){
21        component.outPorts.out.send(data);
22      } else {
23        this.buffer.push(data);
24      }
25    });
26    component.outPorts.out.on("disconnect", () => {
27      this.isConnected = false;
28      this.time_interval = setInterval(() => {
29        this.buffer.splice(0, this.buffer.length);
30      }, time_to_keep_data.value()*60000);
31    });
32    component.outPorts.out.on("reconnect", () => {
33      component.outPorts.out.sendBulk(this.buffer);
34      this.buffer.splice(0, this.buffer.length);
35      this.isConnected = true;
36    });
37  }
38 }
```

Listing 5.17: Implementation of the time-based offline accessibility component.

When data arrives on the input port in line 19, the component checks connectivity in line 20. When there is a connection, i.e., the «*this.isConnected*» flag is *true*, the component sends data in line 21, and otherwise, it pushes

the data records into a buffer in line 23. As mentioned previously in Section 4.3.4, the component assumes an infinite buffer to keep data until the lease window expires. On disconnection (line 26), the component sets «*this.isConnected*» flag to *false* in line 27, waits for the lease window to expire and removes data records from the buffer in line 29. On reconnection (line 32), all stored data records are sent in bulk in line 33, the buffer is cleared in line 34, and «*this.isConnected*» flag is set to *true* in line 35 to prompt the execution to jump to line 21.

**Hybrid-based offline accessibility component:** Listing 5.18 shows the implementation of the hybrid-based offline accessibility component. The number of data records and lease window to keep them in a buffer are specified as the component configuration in lines 13 and 16, respectively. The component defines a timer in line 6 activated when a disconnection happens. Also, the component defines the «*this.isConnected*» flag in line 8 to determine when to send or store data records in lines 24 and 30, respectively. When the connection is available, i.e., *this.isConnected* flag is *true*, the component sends data in line 25. On disconnection (line 33), the component sets *this.isConnected* flag to *false* in line 34. It then checks whether the buffer is full in line 36 and, when the lease window expires, removes data records from the buffer in line 37.

When disconnected, if data arrives on its input port, the component checks if the buffer is full in line 27 and removes old data records, leaving only the most recent data records in the buffer in line 28. Otherwise, it stores the data arriving at its input port in the buffer in line 30. On reconnection (line 41), the stored data is sent on the output port in bulk in line 42, the buffer is cleared in line 43 and *this.connected* flag is set to *false*. Setting the *this.connected* flag to *false* prompts the execution of the component to jump to line 24.

### 5.2.5 Validating Flow-Graphs to Handle Partial Failures

Recall from Section 5.2.4 that the blueprint for the offline accessibility components provides a *\$type* property assigned with a value “*Offline*”. We use the *\$type* property and the component scope to validate each connection to a downstream component in application flow graphs. When connecting offline accessibility components to others successively, DisCoPar-Kilimo checks if the downstream component has either the *\$type* property



```
1 class HybridPolicyBuffering extends OfflineComponent {
2   constructor() {
3     super();
4     this.scope = Globals.Scope.Mobile;
5     this.$type = "Offline";
6     this.time_interval = null;
7     this.buffer = [];
8     this.isConnected = true;
9   }
10  initialise(graph) {
11    this.inPorts.add("in", {type: Globals.DataType.OBSERVATION});
12    this.outPorts.add("out", {type: Globals.DataType.OBSERVATION});
13    this.settings.add("records_to_keep", 5, {
14      type: Globals.SettingType.Numerical
15    });
16    this.settings.add("time_to_keep_data", 30, {
17      type: Globals.SettingType.Numerical
18    });
19  }
20  execute(component) {
21    const {settings: {records_to_keep, time_to_keep_data},} = component;
22    let required_records = records_to_keep.value();
23    component.inPorts.in.on("data", (data, connection) => {
24      if (this.isConnected){
25        component.outPorts.out.send(data);
26      } else {
27        if (this.buffer.length > records_to_keep.value()) {
28          this.buffer.splice(0, this.buffer.length - required_records);
29        }
30        this.buffer.push(data);
31      }
32    });
33    component.outPorts.out.on("disconnect", () => {
34      this.isConnected = false;
35      this.time_interval = setInterval(() => {
36        if (this.buffer.length > records_to_keep.value()) {
37          this.buffer.splice(0, this.buffer.length);
38        }
39      }, time_to_keep_data.value()*60000);
40    });
41    component.outPorts.out.on("reconnect", () => {
42      component.outPorts.out.sendBulk(this.buffer);
43      this.buffer.splice(0, this.buffer.length);
44      this.isConnected = true;
45    });
46  }
47 }
```

Listing 5.18: Implementation of the hybrid offline accessibility component.

with an “*Offline*” value or the *Mobile* scope. The connection is automatically dropped if the downstream component has either. These checks stop application designers from creating incorrect application flow graphs. The implementation of the above validation checks is summarised in Listing 5.19 and Listing 5.20.

Connecting offline accessibility components to mobile components successively is not permitted either, as shown in lines 4 – 9 in Listing 5.19. The code snippet checks if the source and the target components being connected successively have the *\$type* property with a value “*Offline*” and belong to the “*Mobile*” scope. The connection is dropped if the code evaluates to *true* when composing the application flow graphs.

```

1 drawConnection(connection) {
2   let outPortID = this._getOutPortUuid(connection.outPort);
3   let inPortID = this._getInPortUuid(connection.inPort);
4   if((connection.sourceComponent.graph ===
5       connection.targetComponent.graph &&
6       connection.sourceComponent.$type === "Offline") ||
7       (connection.sourceComponent.$type === "Offline" &&
8         connection.targetComponent.scope === "Mobile")) {
9     return undefined;
10  } else {
11    this._drawConnectionBetweenEndpoints(connection, outPortID, inPortID);
12  }
13 }

```

Listing 5.19: Implementaion of the validation to drop connections between successive offline accessibility components.

Lines 6 – 15 of Listing 5.20 generate an error message to alert the developer of incorrect component connections. This validation ensures that an offline accessibility component can only be the last to connect to the server side in a flow graph. Therefore, successive offline accessibility components cannot be connected. The validation happens live while composing applications in the visual programming environment.

### 5.2.6 Handling Partial Failures at the Edge

DisCoPar-Kilimo implements two components to handle partial failures at the edge, i.e., *BufferOnDisk* and *BufferData* components. The *BufferData* component follows a similar approach to the memory-based offline accessibility component explained in Section 5.2.4.1 to store data records in a buffer in memory. In what follows, we explain the implementation of the *BufferOnDisk* component.

```

1  var GraphValidationFunctions = {
2  /*more code*/
3      NoConnectedOfflineComponents: (graph) => {
4          let msg = [];
5          for (let connectionID in graph.connections) {
6              let connection = graph.connections[connectionID];
7              if (connection.sourceComponent.$type === "Offline" &&
8                  connection.targetComponent.scope ===
9                  connection.sourceComponent.scope) {
10                 msg.push(new GraphValidationMessage(false,
11                     /*error message */('));
12             }
13         }
14         return msg;
15     }
16 };

```

Listing 5.20: Implementation for validating flow-graphs that connect successive offline accessibility components.

### 5.2.6.1 BufferOnDisk component

Listing 5.21 shows the implementation of the *BufferOnDisk* edge component. The component has one input port and one output port. When the application graph is executed, the *BufferOnDisk* companion function is invoked. Listing 5.22 shows the companion function of the *BufferOnDisk* component. The function takes input data to store in memory and calls the *storeOnDisk()* friend function in line 2.

```

1  class BufferOnDisk extends Component {
2  /*more code*/
3      initialise(graph) {
4          this.inPorts.add("in", {type: Globals.DataType.EDGE});
5          this.outPorts.add("out", {type: Globals.DataType.EDGE});
6      }
7  }
8

```

Listing 5.21: Implementation of the on-disk buffering edge component.

```

1  function BufferOnDisk(data){
2      var data_to_store = storeOnDisk(data);
3      return data_to_store;
4  }
5

```

Listing 5.22: Companion edge function for the *BufferOnDisk* component.

Listing 5.23 shows the *storeOnDisk* friend function of the *BufferOnDisk*

companion function. The companion function invokes the friend function to store data on disk. It defines two counters in line 2 that are used to keep track of the memory addresses (i.e., *aCounter*) and the entries in the Duktape stack (i.e., *i*). Line 6 fetches data from the stack and keeps it in an array. Line 8 checks for network connectivity, and if there is a connection, data is fetched from memory in line 11 and pushed to the Duktape stack in line 13. The data is written into a disk on lines 16 and 17 when the network fails. In line 20, data is always pushed to the top of the stack and returned to the calling companion function.

```

1 static duk_ret_t native_storeOnDisk(duk_context *ctx) {
2     int aCounter, i;
3     int n = duk_get_top(ctx); /* #args */
4     char* storedData = "no data";
5     for (i = 0; i < n; i++) {
6         storedData = (char*)duk_to_string(ctx, i);
7     }
8     if(socketIO.isConnected()){
9         char* readData = "";
10        for(aCounter = 0; aCounter < EEPROM.length(); aCounter++){
11            readData = (char*)EEPROM.read(aCounter);
12        }
13        duk_push_string(ctx, readData);
14    } else {
15        for(aCounter = 0; aCounter < EEPROM.length(); aCounter++){
16            EEPROM.write(aCounter, duk_get_top(ctx));
17            EEPROM.commit();
18        }
19    }
20    duk_push_string(ctx, storedData);
21    return 1;
22 }

```

Listing 5.23: *Friend function* of the on-disk storage companion function.

## 5.2.7 Tracking Connected Edge Devices

Remember from scenarios three and four in Section 2.2 that more than one edge device can be installed on a farm. Therefore, the farmer needs to know when the devices go off by obtaining general information for all the connected and active devices. This requires the farmer to track all the installed, connected, and active edge devices. To accomplish this goal of tracking the connected devices, DisCoPar-Kilimo incorporates a tracking mechanism for connected edge devices and implements two components, i.e., the *ConnectedDevices* and *DataArrayToTable* components.

### 5.2.7.1 Mechanism for tracking connected devices

To track connected devices, DisCoPar-Kilimo implements a networking code that runs on the mobile device (Listing 5.24) as infrastructure for

monitoring those devices. The networking code uses socket information to track connected devices. All connected sockets are pushed into an array in line 11. The networking code keeps a global counter of all connected devices in line 7. The sockets contain information on the type of connected device, e.g., a phone or an ESP32, as defined by the developer. In our implementations, the edge devices are flagged with a type “esp32” property, as shown in line 14. We use the type property to map only the edge devices and emit the number of connected devices as a “data” event (line 18). The event is broadcast to all subscribed components (clients).

```

1  /*more code */
2  let mobileSocket = [];
3  let dServerSocket = ioc.connect("<<DisCoPar-Kilimo server address>>");
4  io.on("connection", (socket) => {
5      let connectedDevices = null;
6      if(socket.connected){
7          connectedDevices = Array.from(io.sockets.sockets).map(socket => (socket[1].handshake.address).
8              substring(7, (socket[1].handshake.address).length));
9      }
10     /* more code */
11     if(getParam(socket.client.request.url,"device_type") === "phone"){
12         mobileSocket.push(socket);
13     }
14     if(getParam(socket.client.request.url,"device_type") === "esp32"){
15         socket.on("discopar_k_event", (data) => {
16             if(mobileSocket && mobileSocket.length){
17                 mobileSocket.map((s)=> {
18                     s.emit("data", connectedDevices);
19                     s.emit(data.connection, {data: data});
20                 });
21             }
22         });
23     }
24 });

```

Listing 5.24: Implementation for tracking connected devices on the minimalist server.

### 5.2.7.2 Component for tracking connected devices

On the mobile scope of its architecture, DisCoPar-Kilimo incorporates and implements the *ConnectedDevices* component that registers to listen to the events emitted by the networking code infrastructure (line 9 of Listing 5.25). The component receives an array of connected devices as input and sends the array values on its output port in line 11.

### 5.2.7.3 Component for listing connected edge devices

Listing 5.26 shows the implementation of the *DataArrayToTable* component. The component has one input port (line 8) and two output ports

```

1 class ConnectedDevices extends Component {
2   constructor() {
3     /*more code */
4   }
5   initialise() {
6     this.outPorts.add("out", {type: Globals.DataType.EDGE});
7   }
8   execute(component) {
9     Globals.connDevicesSubject.subscribe({
10       next: (connectedDevices) => {
11         component.outPorts.out.send(connectedDevices.values());
12       }
13     });
14   }
15 }

```

Listing 5.25: Implementation of the *ConnectedDevices* component.

(lines 9 and 10). As input, the component receives a list of connected devices from the *ConnectedDevices* component. Line 12 specifies the time interval within which data is sent on the component's output port. The received input message is saved into an array in line 22. The array is processed in lines 25 – 30 to create observations in line 26, update the counter of connected devices in line 28 and add the observations to the observations dataset in line 29. The observation data is defined in line 22. The counter for the number of connected devices is defined in line 24. Line 32 sends data on the component output ports based on the specified time interval. Lines 34 and 35 send the number of connected devices and the dataset to the component's output ports.

## 5.2.8 Accumulating Data from Multiple Edge Devices

Recall from scenarios three and four in Section 2.2 that farmers can install multiple edge devices to monitor the farm conditions. This means that data comes from multiple sources and needs to be accumulated on the mobile phone. DisCoPar-Kilimo implemented the *DeviceAccumulator* component that can accumulate data from several edge devices to accomplish this goal. The application designer can specify the devices from which to accumulate data by indicating the number of devices as a component configuration setting.

### 5.2.8.1 Data accumulation component

Listing 5.27 shows the implementation of the *DeviceAccumulator* component. The component has one input port (line 6) and two output ports

```

1  class DataArrayToTable extends Component {
2      constructor() {
3          /* more code */
4          this.interval = null;
5      }
6
7      initialise() {
8          this.inPorts.add("in", {type: Globals.DataType.ALL});
9          this.outPorts.add("number", {type: Globals.DataType.NUMERIC});
10         this.outPorts.add("out", {type: Globals.DataType.DATASET});
11         this.settings.add("fieldName", 'field${Math.floor(Math.random() * 1000)}');
12         this.settings.add("sendDataInterval", 10000, {
13             type: Globals.SettingType.Numerical
14         });
15     }
16
17     execute(component) {
18         const {settings: {sendDataInterval},} = component;
19         component.inPorts.in.on("data", (input, connection) => {
20             clearInterval(this.interval);
21             var observation = null;
22             var observationDataset = new DataSet();
23             var uniqueConnectedDevices = [...new Set(input)];
24             var number_of_devices = 0;
25             uniqueConnectedDevices.forEach((observationItem) => {
26                 observation = new Observation(component.graph.context,
27                     {[component.settings.fieldName.value(): observationItem]});
28                 number_of_devices += 1;
29                 observationDataset.addObservation(observation);
30             });
31             let send_data = sendDataInterval.value();
32             this.interval = setInterval(sendConnectedDevices, send_data);
33             function sendConnectedDevices() {
34                 component.outPorts.number.send(number_of_devices);
35                 component.outPorts.out.send(observationDataset);
36             }
37         });
38     }
39 }

```

Listing 5.26: Implementation of the *DataArrayToTable* component.

(lines 7 and 8). It defines a map in line 12 and adds data to the map in line 17. The component uses the device identifier as the key and the payload as the value to store data on the map. As a configuration setting, the component takes the number of connected devices from which to accumulate data in line 9. The number of connected devices corresponds to the number of measurements that must be accumulated before they are sent to the output port in line 16. On its output port, the component sends only the data values stored on the map (line 19) and clears the map in line 20.

The output from the *DeviceAccumulator* component is a map of device identities and their corresponding payload. The payload for each specific device can be filtered. To filter the specific payload for particular devices, we implemented the *UnWrapForSpecificDevice* component (Listing 5.30).

```

1 class DeviceAccumulator extends Component {
2   constructor() {
3     /* more code */
4   }
5   initialise() {
6     this.inPorts.add("in", {type: Globals.DataType.EDGE});
7     this.outPorts.add("data", {type: Globals.DataType.EDGE});
8     this.outPorts.add("out", {type: Globals.DataType.NUMERIC});
9     this.settings.add("number", "#devices to accumulate data.");
10  }
11  execute(component) {
12    let newMap = new Map();
13    const {settings: {number}} = component;
14    let number_of_readings = parseInt(number.value(), 10);
15    component.inPorts.in.on("data", (data, connection) => {
16      component.outPorts.data.send(data);
17      newMap.set(data.device_id, data.payload);
18      if(newMap.size === number_of_readings){
19        component.outPorts.out.send(Array.from(newMap.values()));
20        newMap.clear();
21      }
22    });
23  }
24 }

```

Listing 5.27: Implementation of the *DeviceAccumulator* component.

### 5.2.8.2 Filtering edge data

Messages from the edge come in the format depicted in Listing 5.28 i.e., as a JSON object. The data has the identity of the emitting device, the link connecting the edge, mobile components, and the payload. Therefore, this message can be filtered to get the general payload or one for specific edge devices. DisCoPar-Kilimo implemented two filter components: one for filtering the general payload and another for filtering the payload for specific sensor devices.

```

1 {
2   "device_id": "192.168.1.104",
3   "connection": "data:df9643a-0279-4a03-9ebc-d310ab196388_out_bc78d71e-7dbd-4a38-ea18-740561501949_in",
4   "payload": "92"
5 }

```

Listing 5.28: Format of data coming from the edge.

*General payload:* DisCoPar-Kilimo implemented the *UnWrap* component that executes on the mobile phone to get the general payload. Listing 5.29 shows the implementation of the *UnWrap* component. The component has one input port (line 6) and one output port (line 7). On receiving data on its input port, the payload is filtered and sent out on its output port (line 11).



```
1 class UnWrap extends Component {  
2   constructor() {  
3     /* more code */  
4   }  
5   initialise() {  
6     this.inPorts.add("in", {type: Globals.DataType.EDGE});  
7     this.outPorts.add("out", {type: Globals.DataType.NUMERIC});  
8   }  
9   execute(component) {  
10    component.inPorts.in.on("data", (data, connection) => {  
11      component.outPorts.out.send(data.payload);  
12    });  
13  }  
14 }
```

Listing 5.29: Filtering general payload of data coming from edge devices.

*Specific payload:* To get the payload for a specific edge device, DisCoPar-Kilimo implemented the *UnWrapForSpecificDevice* component (Listing 5.30). As mentioned in Chapter 4, the motivation for this component is derived from the fact that in large farms, multiple edge devices can be installed, and the farmer may want to monitor the measurements from specific edge devices. The component takes as a configuration setting the identity of the device for which to filter data. The user must set the device identifier during application design in line 8 of Listing 5.28. This configuration setting is used to filter the payload for a specific device. The component has one input port (line 6) and one output port (line 5). When data is received on its input port, the component checks the device identity (line 13), filters the payload for that device, and sends it to its output port (line 14).

Now that we have explained how DisCoPar-Kilimo features are implemented, we will next describe the process of designing and deploying DisCoPar-Kilimo applications in the subsequent section.

## 5.3 Deploying Applications

This section describes and explains the process followed to deploy applications in DisCoPar-Kilimo. In the subsequent section, we describe the general design of applications in DisCoPar-Kilimo before detailing its deployment strategies.

```

1 class UnWrapForSpecificDevice extends Component {
2   constructor() {
3     /* more code */
4   }
5   initialise() {
6     this.inPorts.add("in", {type: Globals.DataType.EDGE});
7     this.outPorts.add("out", {type: Globals.DataType.NUMERIC});
8     this.settings.add("device", "Device ID");
9   }
10  execute(component) {
11    const {settings: {device}} = component;
12    component.inPorts.in.on("data", (data, connection) => {
13      if(device.value() === data.device_id){
14        component.outPorts.out.send(data.payload);
15      }
16    });
17  }
18 }

```

Listing 5.30: Filtering payload of specific edge devices.

### 5.3.1 Designing Applications in DisCoPar-Kilimo

Figure 5.2 shows the application design interface of DisCoPar-Kilimo. The interface features a canvas, a component menu, a mobile phone and a web emulation preview. The emulation previews are necessary to give the developer the look and feel of the final application. The interface also has an error message section to inform the application designer when errors are detected while composing application flow graphs (e.g., it shows type validation errors).

To design an application, the developer must draw the required components from the component menu, drop them on the canvas, and connect them to compose the application flow graph. Components can be drawn from any of the categories in the component menu. Typical applications constructed using DisCoPar-Kilimo have four parts: the edge application, the mobile application, the server, and the dashboard. The graph of the composed components is validated for correctness in terms of component connections. The validation happens live and gives immediate feedback to the application designer. The flow chart in Figure 5.3 summarises the entire process of building mobile applications.

#### 5.3.1.1 Building the edge application graph

Recall from Section 4.2 that DisCoPar-Kilimo introduces the edge execution scope. This means some parts of DisCoPar-Kilimo applications must be executed on the edge. The deployment of DisCoPar-Kilimo deviates

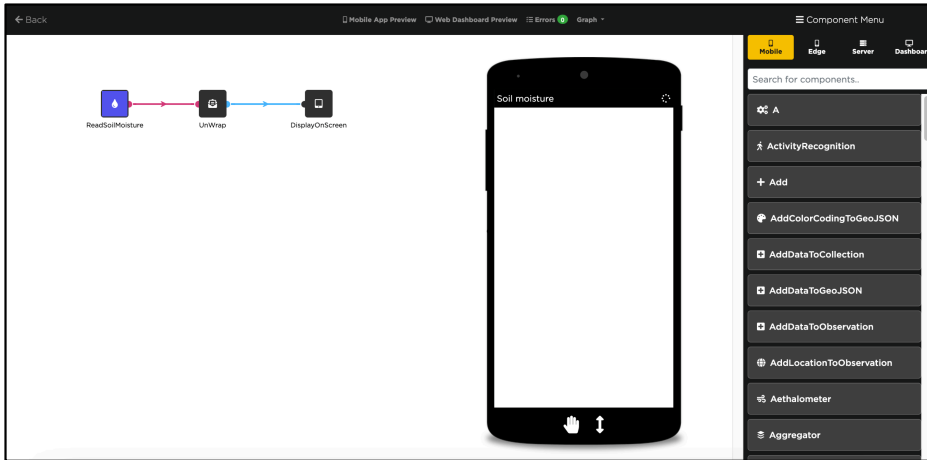


Figure 5.2: Application design interface of DisCoPar-Kilimo showing the VPE (canvas and component menu). The VPE also shows the emulation for mobile applications.

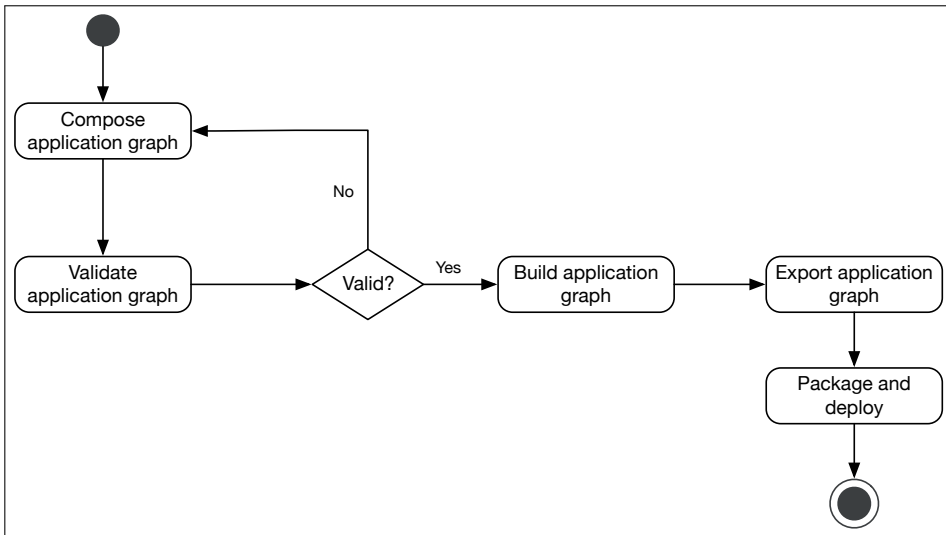


Figure 5.3: Flow chart for building the entire application graph in DisCoPar-Kilimo.

from DisCoPar as two applications may need to be generated: one for mobile and another for edge devices. Only the edge graph (part of the graph with blue components) is exported and deployed to the edge devices. The

edge graph is extracted from the entire application graph represented in JSON format as explained in Section 5.1. The edge graph's implementation and exporting are shown in Listing 5.31. More detailed code is shown in Appendix C.

```

1 function processGraph(ids, graph) {
2   const workList = []; // the first item does not have any parent
3   for (const startNodeId of ids) {
4     workList.push({ nodeId: startNodeId, parentNodesIDs: [] });
5   }
6   const visitedNodes = new Set();
7   const orderedNodes = [];
8   while (workList.length !== 0) {
9     // dequeue the first node from the list
10    const currentNode = workList.shift();
11    if (visitedNodes.has(currentNode.nodeId)) continue;
12    if (hasAllDependencies(visitedNodes, currentNode.parentNodesIDs)) {
13      //the dependencies of the node have been resolved.
14      visitedNodes.add(currentNode.nodeId);
15      //Now its children can be added to the queue.
16      const children = getChildrenOf(currentNode.nodeId, graph.connections);
17      children.forEach(child => workList.push({
18        nodeId: child,
19        parentNodesIDs: getParentsOf(child, graph.connections)
20      }));
21      orderedNodes.push(currentNode);
22    } else {
23      //node cannot be processed since dependencies still need to be resolved.
24      workList.push(currentNode);
25    }
26  }
27  return orderedNodes;
28 }
29
30 function hasAllDependencies(available, myParents) { ... }
31 function getChildrenOf(outPort, edgesSet) { ... }
32 function getParentsOf(nodeId, edgesSet) { ... }
33 function getInitialGraphNodes(graph) { ... }
34
35 /*application*/
36 var ids = getInitialGraphNodes(edgeGraph);
37 var orderedNodes = processGraph(ids, edgeGraph);
38 var edge_graph = buildEdgeGraph(orderedNodes, edgeGraph);

```

Listing 5.31: Implementation for building the edge graph.

The *processGraph* function takes the entire application graph composed on the DisCoPar-Kilimo visual programming environment as an argument. In line 2, the function creates a working array (work list) and pushes the graph nodes to it in line 4. It keeps track of the visited nodes in line 14 to create an ordered list of nodes in line 21 for the edge graph. The function iterates through the work list until all graph nodes are processed. Line 38 shows the application of the *buildEdgeGraph* function to generate the function calls for deployment to the edge device. The *buildEdgeGraph* function receives as input the list of ordered edge nodes (line 37) from the *processGraph* function.

Let us demonstrate the extraction of the edge graph using a concrete example of an application consisting of four nodes: A, B, C, and D. All of these nodes execute at the edge. Listing 5.31 extracts the graph as illustrated in Figure 5.4. Initially, the work list (stack), visited nodes and ordered nodes arrays are empty (Figure 5.4a). Node A is determined to be a root node since it does not have any predecessor node connected to it. Node A is visited and pushed to the ordered nodes array, and its non-visited adjacent (neighbours) nodes B and C are pushed into the work list (Figure 5.4b). In the next iteration, node B is at the top of the work list. Node B is visited, removed from the work list and put into the array of visited nodes and ordered nodes (Figure 5.4c). Node B has no unvisited adjacent nodes. In the next iteration, node C is at the top of the work list. Node C is visited, removed from the work list and put into the array of visited nodes and ordered nodes (Figure 5.4d). Lastly, node D at the top of the work list is visited, removed from the stack and added to the array of visited nodes and ordered nodes (Figure 5.4e). The work list becomes empty, meaning all the nodes have been visited and ordered. The ordered nodes are then used to generate the edge graph.

### 5.3.2 Deploying DisCoPar-Kilimo Applications

We now detail how deployment happens per scope. Each scope generates a respective application, i.e., for the web, mobile or edge. When the user clicks deploy, three applications will be deployed as follows.

#### 5.3.2.1 Web application

The web (dashboard) application is automatically deployed to the cloud or server hosting DisCoPar-Kilimo. The components used to build the web application are rendered to the web client and are implemented in HTML5, JavaScript and CSS. The web components have access to the document object model (DOM) that comprises the structure and content of documents on the web, i.e., the DOM contains fields that hold the HTML code. When loading the pages for the web client, the underlying dashboard components automatically connect themselves to the corresponding server-side output ports. From that point on, the dashboard receives real-time status updates from the server. When initialising their connection, the server-side components will send their entire state

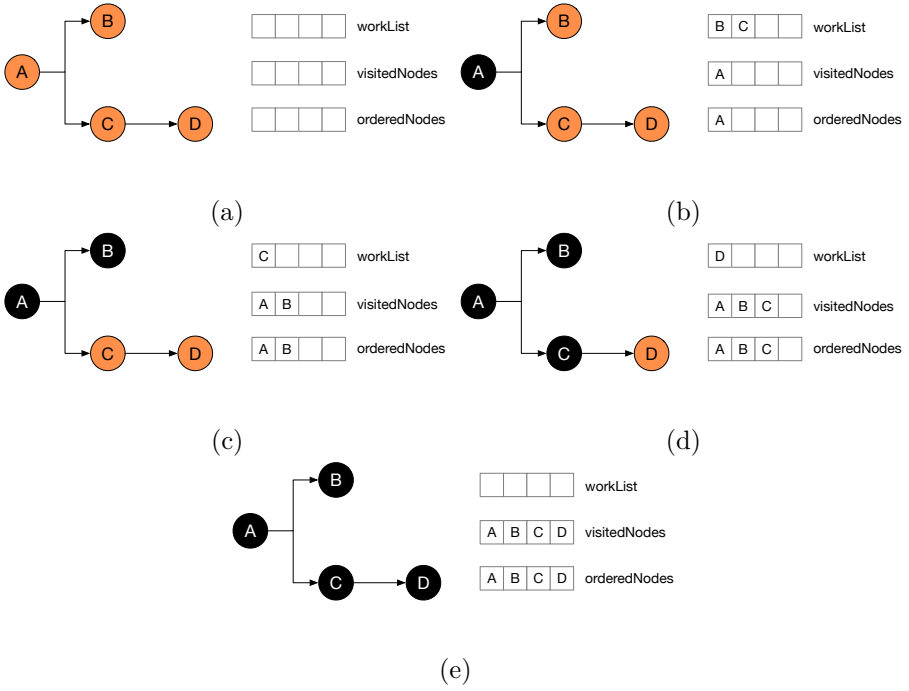


Figure 5.4: Process for extracting the edge application graph.

to update the dashboard. Figure 6.13 in Section 6.2.6 shows a preview of a web application.

### 5.3.2.2 Mobile application

The application developer deploys the mobile application to the mobile phone by installing the corresponding distribution file, e.g., the *.apk* for Android devices. This is a two-step process. In the first step, the developer must download the mobile application graph in JSON format. Then, application developers need to use *Cordova*<sup>5</sup>, a cross-platform application deployment tool, to package the application graph together with the base DisCoPar-Kilimo into an executable file following the steps described below. The application developer must also ensure the tool is installed and working. Then, the developer downloads and sets up a Cordova project for DisCoPar-Kilimo mobile applications. The base DisCoPar-Kilimo (*disco-*

<sup>5</sup><https://cordova.apache.org/>

*par.min.js*) should be copied into the */www/js/* directory of the *Cordova* project. The base application for DisCoPar-Kilimo contains the core and user-defined components necessary for the mobile application graph to execute.

In the second step, the developer exports and downloads the mobile application graph from the DisCoPar-Kilimo canvas by clicking the *Download mobile graph* button as shown in (Figure 4.22). This will download the mobile application graph as a JSON file. The JSON file should be copied into the root directory of the *Cordova* project. The last step is opening a command terminal, navigating to the root directory of the *Cordova* project and executing the *cordova run «target-platform»* command, e.g., *cordova run android*. The above command will package and deploy the application to the Android phone connected to the computer via a USB port. The developer should install any command line tool on the mobile phone and use it to deploy the networking code to the mobile phone. The networking code (Appendix D) must be up and running to communicate between the edge and mobile components.

### 5.3.2.3 Edge application

Deploying applications to the edge requires a base edge application and the edge application graph. The base edge application talks directly to the hardware of the edge device and runs a lightweight JavaScript engine that can execute the companion functions of the edge components. The edge application graph is designed on the VPE of DisCoPar-Kilimo and exported as a JavaScript file (as illustrated in Figure 4.22). The exported file contains the companion functions for the edge components. The companion functions are invoked when the graph is executed as explained in Section 5.2.2.

Edge applications are deployed to ESP32<sup>6</sup> supported hardware or any other hardware with similar configurations. Deploying the application to the edge device follows the following steps. First, the developer should download the edge graph and copy the file into the */data/* directory of the DisCoPar-Kilimo edge application setup as a PlatformIO project. The file contains the edge application graph and the functions that are invoked when the edge application executes. Second, the developer specifies the

---

<sup>6</sup><https://www.espressif.com/en/products/socs/esp32>

network credentials and the mobile device’s address. In the next step, the developer should run a series of commands as follows:

1. “*platformio run -target buildfs -environment m5stick-c*” command to compile the file system,
2. “*platformio run -environment m5stick-c*” to compile the edge application,
3. *platformio run -target uploadfs -environment m5stick-c* command to install the file system on the edge device and lastly,
4. “*platformio run -target upload -environment m5stick-c*” command to install the edge application.

Appendix E lists the resources used for the applications implemented in this work.

## 5.4 Conclusion

This chapter discusses the implementation of core concepts and components of DisCoPar-Kilimo. DisCoPar-Kilimo proposes and implements a new design choice to support a new breed of components that can execute at the edge. At the edge, the components are implemented as companion functions that may further invoke friend functions. The friend functions can talk directly to the hardware of edge devices hosting sensors. This enabled *stripping down* the implementation of the edge components in DisCoPar-Kilimo and transferring the business logic for those components to the edge. We use the *stripped-down* edge components on the visual programming environment to ensure composing complete application flow graphs and configure the state for the edge components. The state of the edge components is passed as a configuration for those components.

The chapter presents the mechanisms introduced to handle partial failures at the edge. The chapter presents and discusses the policies introduced to handle partial failures on the mobile side. The policies are implemented as concrete components of DisCoPar-Kilimo. Though the implementations we present in this chapter advance the work done on the existing low-code development environments, there is still room to add more components that can execute at the edge for advanced real-world



applications. In the next chapter, we present the validation of DisCoPar-Kilimo by implementing the driver scenarios from Chapter 2 on it.

# Chapter 6

## Validation

In this chapter, we validate the approach we present in this dissertation and its implementation in DisCoPar-Kilimo. The validation aims to examine the applicability of DisCoPar-Kilimo in implementing smart agriculture applications. To this end, we implement seven smart agriculture scenarios on DisCoPar-Kilimo. The implemented scenarios were introduced in Section 2.2.

We begin the chapter by describing the validation approach adopted in this dissertation in Section 6.1. Then, we present and describe the implementation of each scenario in Section 6.2. Lastly, the chapter ends with a discussion and a conclusion on the implemented scenarios based on the four properties for smart agriculture applications as explained in Section 2.3.

### 6.1 Validation Approach

In this chapter, we assess the applicability of our approach by answering the following questions.

1. *Can DisCoPar-Kilimo simplify the development of smart agriculture applications?*
2. *How flexible is DisCoPar-Kilimo in implementing smart agriculture applications?*

In this dissertation, we use the scenario-based approach to answer those questions. We define scenarios which are then implemented using

DisCoPar-Kilimo. In a scenario-based validation approach [ACS10], a scenario describes the set of interactions between different actors in a system and can comprise a concrete sequence of interaction steps or a set of possible interaction steps. The interaction steps represent the behaviour of the application. During validation, the requirements of the scenario are verified against the expected behaviour of the application.

Scenarios provide an intuitive approach to describe complex behaviours of software applications [MMCR13]. Scenarios can be derived from the requirements analysis and used later to verify whether the implementation satisfies the specifications described by the scenario for validation. Reusing scenarios can help in economising the development of validation use cases.

In the subsequent sections, we describe the implementation of the scenarios we use to evaluate DisCoPar-Kilimo.

## 6.2 Validation Scenarios

For our evaluation, we start from the driver scenarios presented in Section 2.2 using DisCoPar-Kilimo. We used the scenarios to identify the features that low-code development environments need to support implementing smart agriculture applications. In addition, we implement three additional scenarios derived from the literature [SNN18] and our experiences with agricultural extension workers in Kenya. In the following sections, we describe each scenario and explain its implementation.

### 6.2.1 Scenario 1: Monitoring Soil Moisture

As mentioned in Section 2.2, soil moisture is critical in maintaining the ecological stability of the soil-plant-atmosphere system [ZPP<sup>+</sup>20, SPKB<sup>+</sup>22]. For instance, changes in soil moisture directly influence the hydrological processes on the land surface. Hence, accurate soil moisture estimation is essential for ecological assessment and predicting crop growth and development.

Farmers in developing regions rely on physical observations (e.g., when it rained last) to determine whether the available soil moisture is sufficient to support the growth and development of crops [SPKB<sup>+</sup>22]. Therefore, access to soil moisture data can significantly benefit agricultural extension

workers in making appropriate decisions concerning their advice to farmers, e.g., when to plant or even when to water the crops. Although weather information services have been promoted in several developing countries, soil moisture measurements in situ are required to address needs at the farm level [OCC<sup>+</sup>13, MMC19].

In this scenario, we build a smart agriculture application which monitors soil moisture in the following conditions.

1. A small farm (e.g., a research garden in a university environment) that always has a reliable internet connection.
2. The farmer has an up-to-date view of the soil moisture conditions on the farm.

Figure 6.1 shows the flow-graph of the application to monitor soil moisture. Recall that application flow graphs are read from left to right, and the arrows on the component links show the direction in which data flows through the graph. The application comprises three components to read and display soil moisture on the mobile phone. Recall from Section 5.1.5 that connecting components with different execution scopes automatically yields a distributed application. In this case, the application runs on an edge device and mobile phone.



Figure 6.1: Flow-graph of the application for monitoring soil moisture.

The application uses the *ReadSoilMoisture* component to read soil moisture every 30 minutes. Recall from Section 4.3.3 that components that execute at the edge scope can directly be connected to those that run on the mobile scope in DisCoPar-Kilimo’s VPE. As shown in Figure 6.1, the *ReadSoilMoisture* component is directly connected to the *UnWrapMoisture* component, avoiding a centralised server for communication. Figure 6.2 shows the deployed soil moisture sensor (the part labelled A) and the application preview to display the soil moisture readings (the part labelled B). The soil moisture sensor (circled in orange) is attached to an M5StickC microcontroller (circled in white). The label and measurements

for the soil moisture are shown on the user interface thanks to the *DisplayOnScreen* component. The application's GUI is built based on the flow graph composed by either the application designer or domain expert.

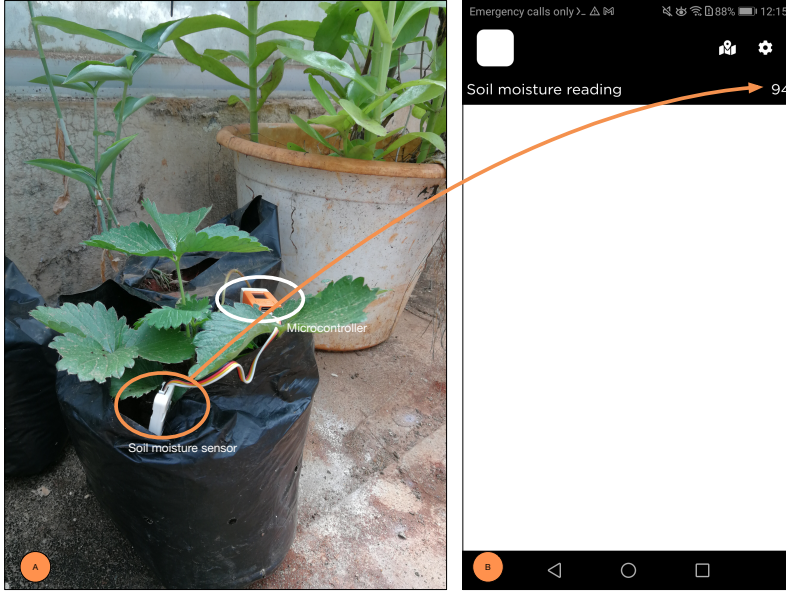


Figure 6.2: Preview of the application for monitoring soil moisture deployed to crops in a greenhouse.

### 6.2.2 Scenario 2: Computing Average Soil Moisture and Keeping Data at the Edge

As mentioned in Chapter 2, this scenario considers a small farm, and the farmer periodically visits the farm. Hence, the scenario seeks to fulfil the following two goals.

1. Perform computations for the average soil moisture at the edge device.
2. Store sensor data at the edge and send it to the farmer's mobile device when it becomes available.

Figure 6.3 shows the implemented application in DisCoPar-Kilimo. The flow graph comprises five components – three edge components and

two mobile components. The *ReadSoilMoisture* component reads soil moisture values and sends them to the *ComputeEdgeAverage* component that computes the average. The average soil moisture values from the *ComputeEdgeAverage* component are sent to the *BufferOnDisk* component for storage when the network fails. Recall from Section 4.3.2 that edge components have companion functions invoked to execute at the edge devices. When the network connection is available, data is sent from the *BufferOnDisk* component to the *UnWrapMoisture* component on the mobile phone in JSON format. The *UnWrapMoisture* filters the soil moisture values and sends numerical values to the *DisplayOnScreen* component for display on the mobile device. Again, the *BufferOnDisk* edge component is directly connected to the *UnWrapMoisture* component that runs on the mobile device.

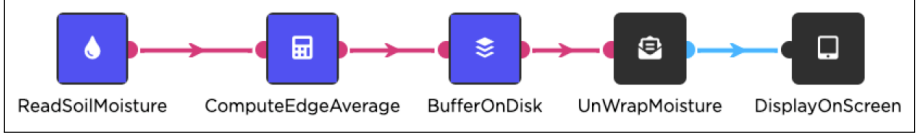


Figure 6.3: Flow-graph of an application for computing average soil moisture and keeping data at the edge. The application is composed of three edge components and two mobile components.

### 6.2.3 Scenario 3: Monitoring Soil Moisture Using more than one Edge Device

As explained in Chapter 2, this scenario considers a large farm with many edge devices deployed. The scenario seeks to fulfil the following two goals.

1. Accumulate soil moisture sensor data from multiple edge devices on the mobile phone.
2. Compute the average soil moisture of the data accumulated and the minimum and maximum soil moisture of the accumulated data.

Figure 6.4 shows the application flow-graph composed of edge and mobile components. Similar to the previous scenario, the application reads soil moisture using the *ReadSoilMoisture* component and stores data when networks fail at the edge using the *BufferOnDisk* component. In this scenario, however, computing the average soil moisture does not happen at

the edge. Instead, soil moisture data is accumulated on the mobile device using the *GatherMoistureReadings* component. Recall from Section 4.3.6 that each edge device represents one measurement. Hence, the configured number of devices also represents the number of soil moisture measurements to accumulate. The accumulated data is sent to the output port as an array of numeric values.

Once data is at the mobile phone, the average soil moisture is computed for the accumulated data using the *ComputeMoistureAverage* component. The component receives an array of numeric values from the *GatherMoistureReadings* component. The minimum and maximum soil moisture values are computed using the *ComputeMinimum* and *ComputeMaximum* components, and the values are displayed on the mobile device using the *DisplayMinimum* and *DisplayMaximum* components, respectively. We use the existing *SetDecimalPlaces* component to round off the soil moisture average.

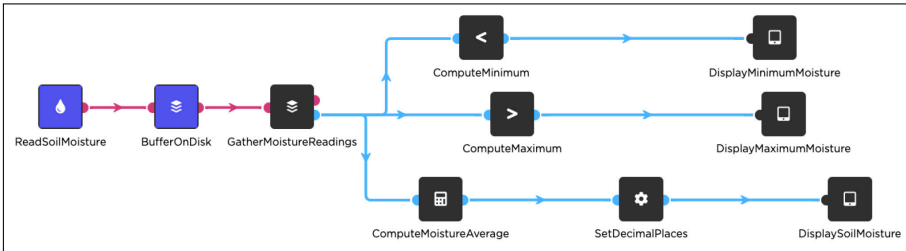


Figure 6.4: Flow-graph of an application for monitoring soil moisture using more than one edge device.

Figure 6.5 shows the deployment of the application for monitoring soil moisture using more than one edge device. In the figure, three soil moisture sensors were deployed to three plant pots in a greenhouse to simulate a large crop field. Each of the soil moisture sensors was attached to a microcontroller. Recall that the microcontroller receives and sends the sensor measurements to the mobile phone. Part B of Figure 6.5 shows the minimum, maximum and average soil moisture displayed on the mobile application.

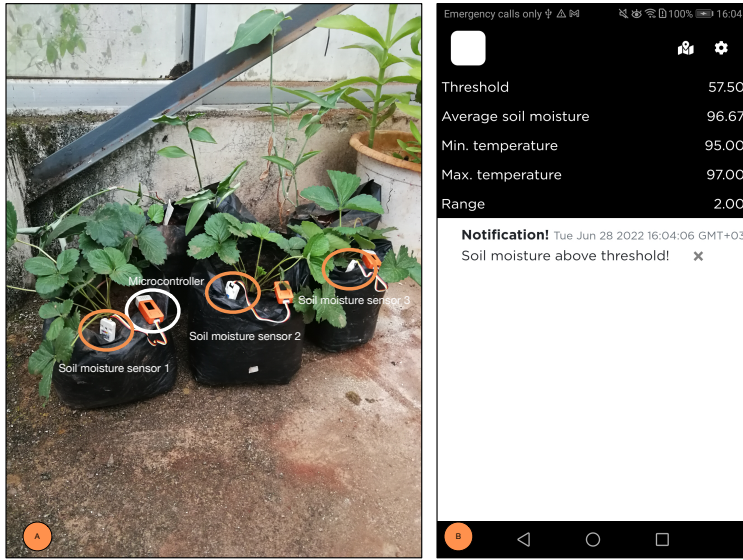


Figure 6.5: Preview of the application for monitoring soil moisture using more than one edge device.

#### 6.2.4 Scenario 4: Tracking Connected Edge Devices

As mentioned in Chapter 2, some devices can fail due to drained batteries or mechanical malfunction. This scenario seeks to fulfil the following goals.

1. Track connected edge devices.
2. Display the number of connected edge devices as a numerical value.
3. Display connected edge devices as a list on the farmer's mobile phone.

Figure 6.6 shows the application flow-graph added to the graph of Figure 6.3 to track devices. The tracking happens only on the mobile side; messages for connected edge devices are broadcast to all mobile-scope application components. The *ConnectedDevices* component in this application subscribes to receive messages on all connected edge devices and sends on its output port a list of connected devices. The list is received by the *DataArrayToTable* component, which processes it to (1) count the number of connected devices for display using the *NumberOfConnectedDevices* component and (2) display as a table using the *ListConnectedDevices*



component. The *NumberOfConnectedDevices* and *ListConnectedDevices* components are reused from the existing components in DisCoPar.

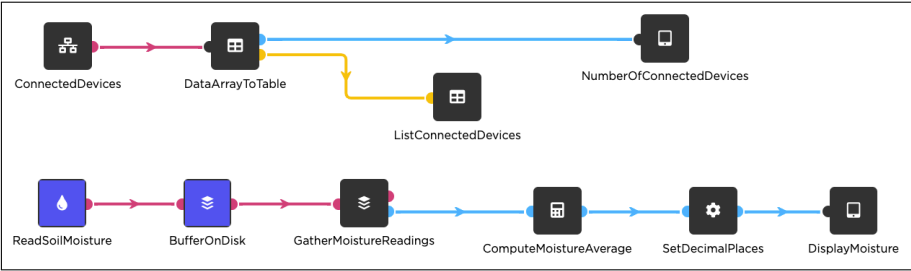


Figure 6.6: Flow-graph of an application for tracking general information of connected edge devices.

Figure 6.7 shows the preview of the implemented application with the number and listing of the connected edge devices. When edge devices go offline or return online, the number and list of connected edge devices are updated automatically in real-time.

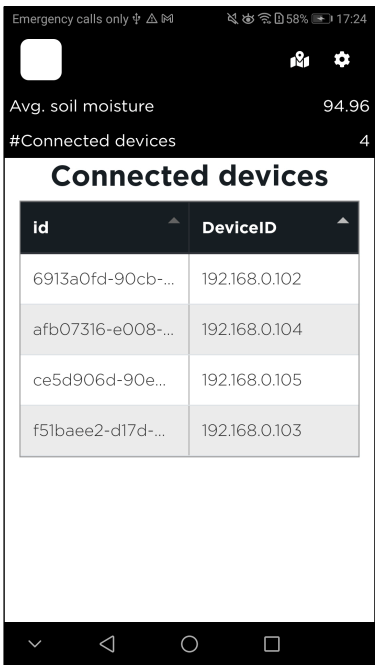


Figure 6.7: Preview of the application for tracking connected edge devices.

### 6.2.5 Scenario 5: Tracking and Monitoring Paddy Rice Storage Conditions

The scenario was derived from the literature as explained in Section 2.2. The goals of the scenario are as follows.

1. Measure humidity level in paddy rice storage locations. Humidity sensors attached to an edge device can be installed at several points within the storage area to take measurements. We assume that several paddy bags can be piled on each other to allow measuring one pile at a time. The measurements are sent to the farmer's mobile phone for further processing.
2. Accumulate humidity data from all the installed edge devices. Remember that rice storage areas can have several relative humidity measurement points with edge devices installed. Hence, humidity measurements for all the edge devices must be collected on the mobile phone for further processing.
3. Show humidity levels on a gauge and line chart.
4. Set a threshold for generating and showing notifications. The farmer must be alerted when the humidity level exceeds a certain threshold.
5. Store humidity measurements in a database for future retrieval and, at the same time, display them on a web dashboard.
6. Generate CSV report for humidity data. According to Serikul *et al.* [SNN18], the CSV report allows further processing of humidity data by third-party applications.

Compared to the application presented in [SNN18], our implementation adds the following. First, it accumulates humidity data from edge devices and processes it to get a general overview of rice storage areas. Second, it generates notifications when the humidity level exceeds the required limit.

Figure 6.8 shows the flow graph of the application for tracking the humidity levels in paddy rice storage areas.

The application uses the *ReadHumidity* component to measure the humidity levels. The humidity data is stored at the edge device when there is a network disconnection using the *BufferOnDisk* component. Humidity

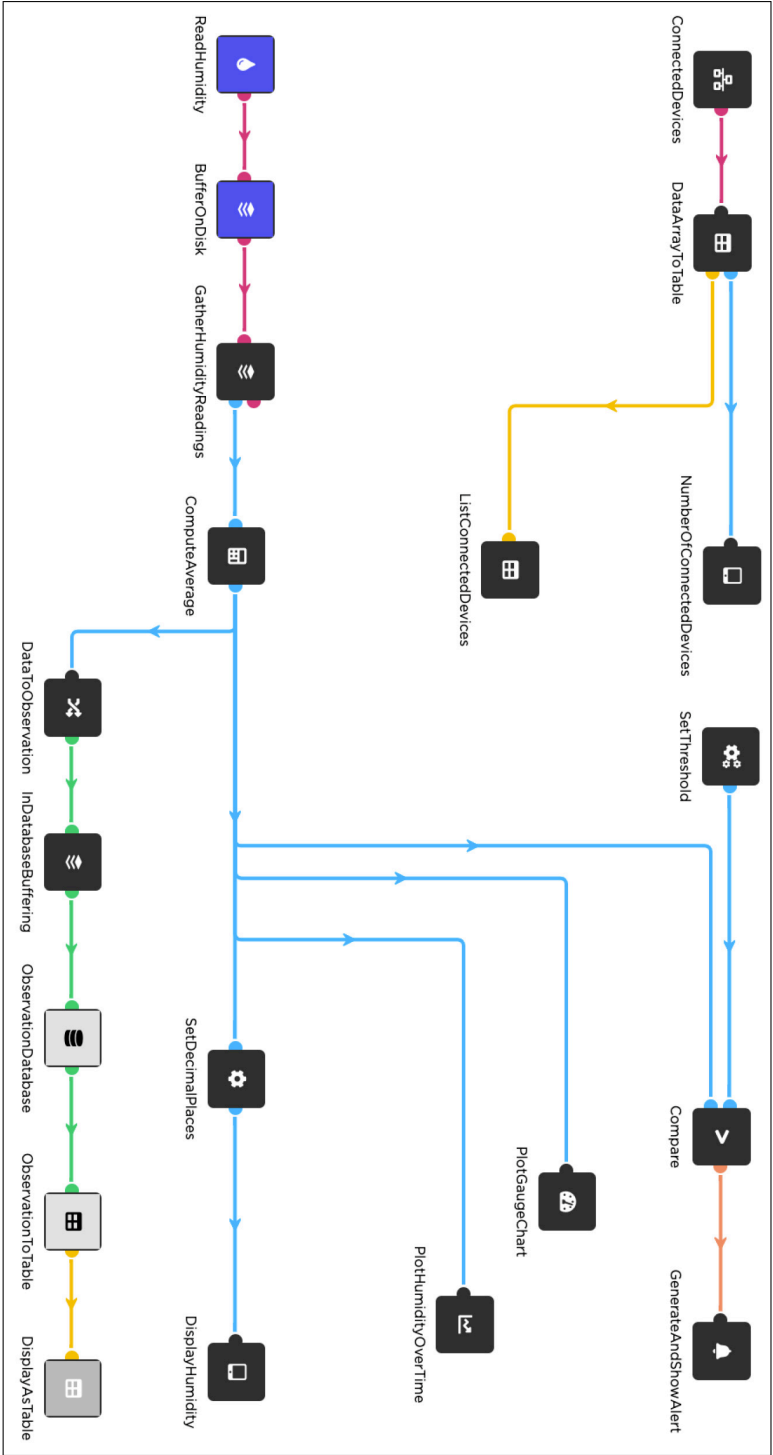


Figure 6.8: Flow-graph of an application for tracking and monitoring humidity levels in storage areas for paddy rice.

measurements are accumulated using the *GatherHumidityReadings* component on the mobile device. The accumulated data is processed further using the *ComputeAverage* component to get the average humidity level.

Tracking humidity on a gauge chart uses the existing DisCoPar *PlotGaugeChart* component. Tracking humidity levels over time (i.e., on a line chart) uses the existing DisCoPar *PlotHumidityOverTime* component. In addition, the component was modified and adapted to plot the maximum and minimum humidity levels. Both the *PlotGaugeChart* and *PlotHumidityOverTime* components receive as input the average humidity from the *ComputeAverage* component. To generate notifications, the application starts by setting the threshold using the *SetThreshold* component configured with 75% for a tropical environment [CFM<sup>+</sup>19]. The threshold can be changed by configuring the component as shown in Figure 6.9. By default, the component emits the set threshold on its output port every 10 seconds. The application uses the *GenerateAndShowAlert* component to generate humidity notifications.




Figure 6.9: Configuring the *SetThreshold* component to specify the time interval for sending the set value on the output port.

Collecting humidity data into a database for long-term storage and future retrieval uses the existing DisCoPar *ObservationDatabase* component. This component accepts observations as input data. As such, the humidity data is first converted into an observation by the *DataToObservation* component. Humidity data is exported to CSV report via the *DisplayAsTable* component. The component adds a “Download CSV Report” button to the application’s web dashboard. The *DisplayAsTable* component also displays the humidity data as a list on a web dashboard.

To track connected devices, the application uses the *ConnectedDevices* component that sends an array of connected devices. The *DataArrayToTable* component converts the received array of connected devices and sends two outputs. The first output it sends is a numerical number of connected devices to the *NumberOfConnectedDevices* component for display.

The second output it sends is a list of connected devices for display using the *DisplayAsTable* component.

Figure 6.10 and Figure 6.11 show the preview of the implemented application. Figure 6.11 shows the results of the CSV report when the “Download CSV Report” button is clicked.

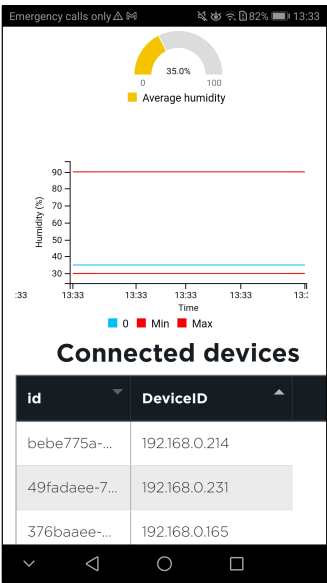


Figure 6.10: Mobile application preview for tracking humidity levels in storage areas for paddy rice. Humidity is below the 75% threshold, and no alert is generated.

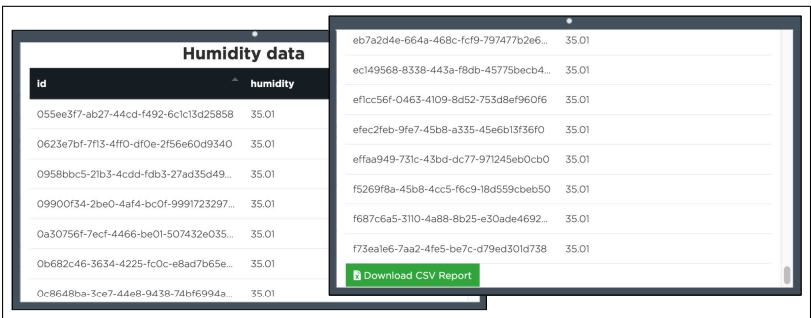


Figure 6.11: Dashboard application preview for tracking and monitoring humidity levels in storage areas for paddy rice.

### 6.2.6 Scenario 6: Collecting Data Using Mobile Applications

This scenario was introduced in Section 2.2. We rely on plant labels to identify individual plants in the field. We assume the plant labels are QR codes generated using an external application. This scenario seeks to fulfil the following goals.

1. Scan plant labels and present a survey to the farmer to enter plant growth and status data.
2. Store the survey data on the mobile phone when the network fails.
3. Send the data to a server for storage and display it on a dashboard.

Figure 6.12 shows the flow graph of the application for tracking plant-specific data. The application scans plant labels using the *ReadQRCode* component. The component implements a QR code scanner that uses the mobile phone's camera where the application is running. On successfully reading the plant label, a survey is generated using the *ObservationPopUpSurvey* component already existing in DisCoPar. The survey entries we defined for this application are the number of leaves, the colour of leaves and the plant height.

Data is stored on the mobile device when the network fails using the *InDatabaseBuffering* component already in DisCoPar. The component creates a small database on the mobile to store data, which it synchronises to the server when the network is restored. Recall from Section 4.3.4 that the *InDatabaseBuffering* component is the last one to connect to the server-side component following the offline accessibility policies for DisCoPar-Kilimo.

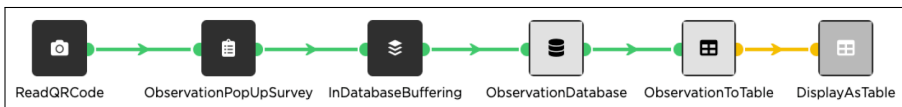


Figure 6.12: Flow-graph of an application for tracking application data.

Part 1 of Figure 6.13 shows the preview of the data collection survey on a mobile phone. Part 2 of Figure 6.13 shows the web dashboard of the application. The data can be exported from the web dashboard as a CSV file

for further analysis. The application uses the existing DisCoPar *ObservationDatabase* component to store the data in a database. The component transparently establishes a connection and inserts data into the database running on the server. The data saved into a database is displayed on a dashboard using the existing DisCoPar *DisplayAsTable* component. The web dashboard of the application allows users to download collected data as a CSV report. Domain experts from other fields, such as statistical science experts, can use such reports to gain more insights.

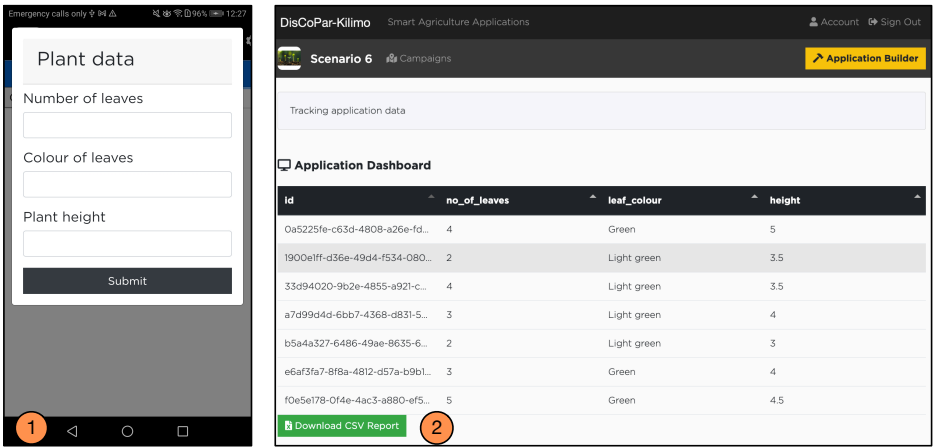


Figure 6.13: Preview for tracking application data. The application features a survey for data entry and a web dashboard for data display.

6.2.7 Scenario 7: Monitoring Soil Moisture and Temperature in Corn Seeding and Sprouting

This scenario was introduced in Section 2.2. The goals of this scenario are as follows.

- 1. Read soil moisture and temperature.
- 2. Compute soil heat capacity to determine soil moisture and temperature interaction.
- 3. Accumulate soil moisture and temperature data on the mobile phone.
- 4. Generate and show notifications on soil moisture.
- 5. Visualise soil moisture data over time.

Figure 6.14 shows the application flow graph for monitoring corn seeding and soil heat capacity computation. The application uses the *ReadSoilMoisture* and *ReadTemperature* components to measure soil moisture and temperature. Accumulating data uses the *GatherMoistureReadings* and *GatherTemperatureReadings* components. The accumulated data is processed to get the soil moisture and temperature averages using the *ComputeAverage* components.

Computing the soil heat capacity is done using mobile components that perform arithmetic operations (such as addition, subtraction, multiplication and exponentiation) according to the following formula:  $Q = 4.2 \times 10^3 \times V \times (0.2 + W) \times \Delta T$ , where  $V$  is the soil volume,  $W$  is the soil moisture, and  $\Delta T$  is the temperature change [ZPP<sup>+</sup>20]. The soil heat capacity is displayed on the mobile device using the *DisplayHeatCapacity* component.

The application also uses the *SetThreshold* component to specify the threshold for generating notifications when soil moisture exceeds the set threshold. The threshold value was set at 57.5% for a tropical setting [STCR13]. The threshold value is compared with the incoming average soil moisture data using the *Compare* component. The *Compare* component receives two inputs, i.e., the threshold and the data value. The output from the component triggers the *GenerateAndShowAlert* component to generate and show soil moisture notifications as popup messages on the mobile phone. Lastly, the application uses the *PlotSoilMoistureOverTime* component to visualise the average soil moisture on a line chart. This component plots the average soil moisture and the maximum (98%) and minimum (55%) thresholds. During application design, the maximum and minimum thresholds are configured in the component setting. The component receives, as input, the average soil moisture from the *ComputeAverage* component.

Figure 6.15 shows the preview of the resulting application. The line chart shows the average soil moisture over time. The notifications displayed as pop-up messages on the mobile phone were generated at 57.5% soil moisture.



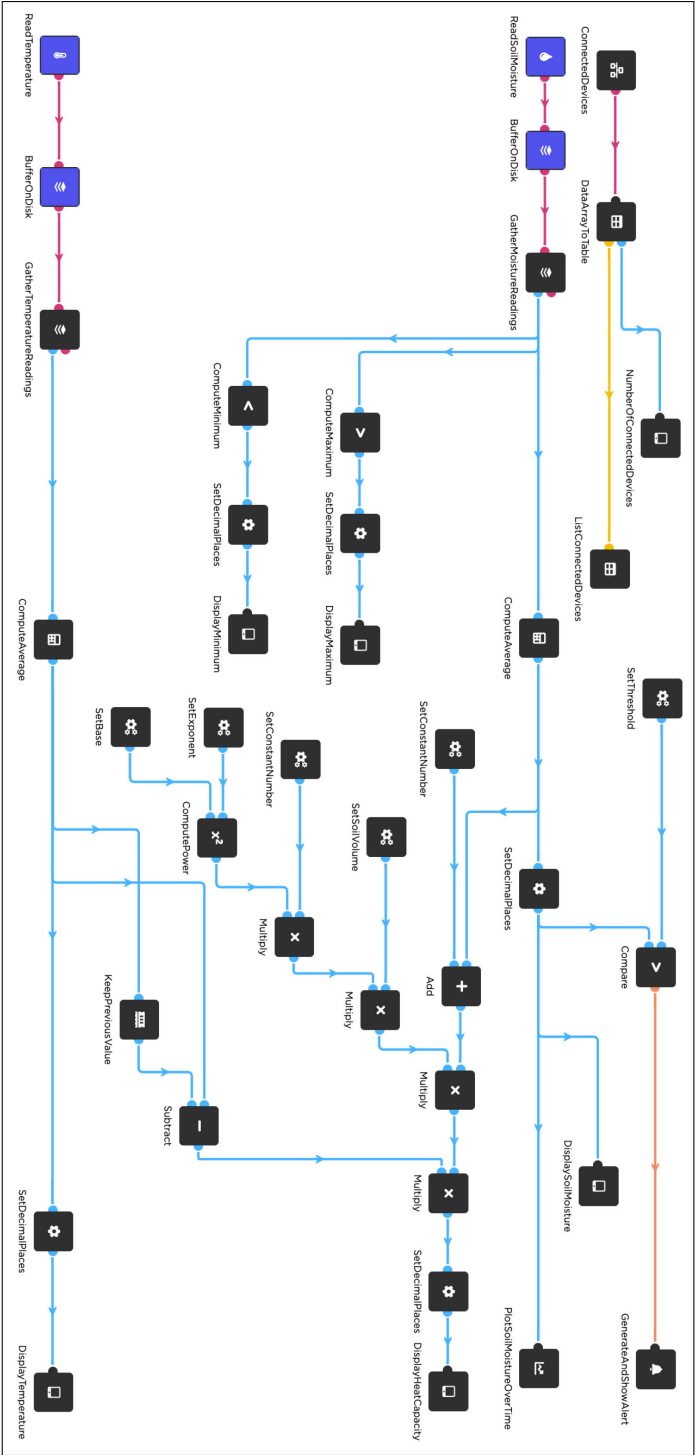


Figure 6.14: Flow-graph of an application for monitoring soil moisture and temperature in corn seeding and sprouting.

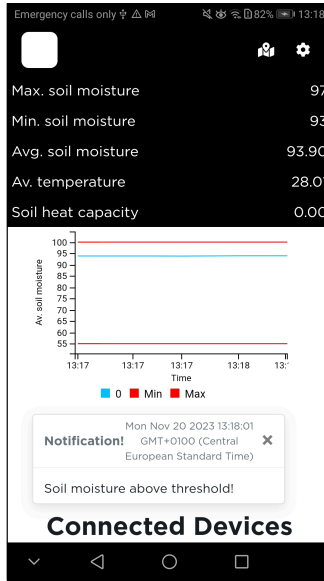


Figure 6.15: Application preview for monitoring farm conditions in corn seeding and sprouting. The preview shows the maximum, minimum and average soil moisture.

## 6.3 Discussion

We now discuss how DisCoPar-Kilimo adheres to the properties presented in Chapter 2.

*Environment sensing:* DisCoPar-Kilimo introduces new and dedicated components for smart agriculture applications requiring environmental sensing. The above components are collectively called *environment sensing* components. The environment sensing components form a sound basis for providing a rich portfolio of future sensing components. Our approach allows sensor data to accumulate on mobile devices, where it can be processed to give a general overview of environmental conditions on the farm.

*Computation at the edge:* DisCoPar-Kilimo provides infrastructure and dedicated components for performing computations at the edge. As explained in Chapter 4 and Chapter 5, DisCoPar-Kilimo implements companion and friend functions to interact with the sensor hardware. In contrast to the existing LCDEs, our approach provides

mechanisms to export and execute application graphs at the edge devices.

*Coordination with the edge:* DisCoPar-Kilimo devises a communication mechanism between components executing at the edge and those running on mobile devices. Then, DisCoPar-Kilimo implements the mechanism into a networking code to support the implementation of smart agriculture applications. The mechanism enables it to support coordination with the edge and track connected edge devices. It also provides components to accumulate and filter data coming from edge devices. In contrast to existing LCDEs, our work enables communication between edge and mobile devices without requiring a centralised server.

*Handling partial failures:* First, DisCoPar-Kilimo devises policies for handling partial failures at the edge device and on the mobile device. Second, DisCoPar-Kilimo provides dedicated edge and mobile components based on the devised policies that can handle partial failures in smart agriculture applications when networks fail. In contrast to existing LCDEs, our work extends offline accessibility to edge devices.

The implemented scenarios demonstrate that DisCoPar-Kilimo provides “ready-to-go” components for implementing different SAAs. The components hide away application development issues like coordinating communication between different components that compose SAAs.

## 6.4 Conclusion

In this chapter, we implemented different scenarios to show how the LCDE presented in this dissertation meets our research vision for constructing smart agriculture applications. By implementing different scenarios, we demonstrate how DisCoPar-Kilimo supports environment sensing, computation at the edge, coordination with the edge, handling partial failures, tracking connected edge devices and accumulating data from multiple edge devices. Supporting the above properties means that DisCoPar-Kilimo can be considered, to the best of our knowledge, the first of a kind in new and future generations of LCDEs for implementing SAAs.

# Chapter 7

## Conclusion

This dissertation explored a low-code development environment for implementing smart agriculture applications. In this final chapter, we revisit the problem statement, research approach, contributions, and limitations and provide directions for possible future work.

### 7.1 Problem Statement Revisited

Smart agriculture applications are geared towards improving modern farm operations to obtain optimal yields. The applications orchestrate components deployed in microcontrollers, smartphones, and cloud services. Implementing these applications requires skilled engineers, careful handling of distribution, dealing with potential network and device failures, and integration with resource-constrained devices. Smart agriculture applications are often constructed using textual programming languages requiring software development knowledge that domain experts usually need to have. Low-code development environments have emerged as an alternative for users needing more technical expertise. The low-code development environments offer visual programming environments with “ready-to-use” components, making software development more accessible to all technical skill levels.

In this work, we identified four properties that a low-code development environment should have to simplify the development of smart agriculture applications. We summarise them below.

1. *Environment sensing* to monitor prevailing farm conditions. Pre-

vailing farm conditions are crucial in the growth and development of crops. Hence, low-code development environments should allow sensing data to be collected to monitor environmental conditions.

2. *Computation at the edge* to process data near the source. Farmers, especially in rural areas, do not have uninterrupted access to cloud services. Hence, collecting data and performing computations on edge devices instead of sending all data to a centralised server for processing is essential.
3. *Coordination with the edge* to ensure that data is sent from the edge devices to the mobile devices. Implementing distribution and computation at the edge requires supporting communication between edge and mobile devices. This communication requires some form of coordination between the edge and mobile devices.
4. *Handling partial failures* to keep data when the networks become unavailable. LCDEs should provide mechanisms for dealing with partial failures. The failures can lead to data losses, which can affect decision-making processes. Partial failures can happen between edge and mobile devices and between the mobile device and the server.

We performed a state-of-the-art analysis of low-code development environments using the identified properties. The findings showed that none of the existing low-code development environments supported all the identified properties.

## 7.2 Research Approach Revisited

Recall that we introduced and stated our research approach in Chapter 1. In Chapter 2, we presented two studies to motivate the work in this dissertation. The two studies presented state-of-the-art solutions for smart agriculture applications and low-code development environments. In Chapter 3, we described in detail a low-code development environment that we extended in this dissertation with novel components for constructing smart agriculture applications. We explained our low-code development environment's features and implementation details in Chapter 4 and Chapter 5. As mentioned earlier, the components were geared to support the properties of smart agriculture applications.

In this dissertation, we adopted a scenario-based validation approach. We implemented the seven driver scenarios introduced in Chapter 2 in DisCoPar-Kilimo.

## 7.3 Contributions

Our work on developing a low-code development environment for smart agriculture applications led to this dissertation’s main contributions.

- As our first contribution, the dissertation proposes a set of properties that a domain-specific low-code development environment needs to support implementing smart agriculture applications. The proposed properties include support for environment sensing, computation at the edge, coordination with the edge and handling partial failures when networks fail.
- Our second and main contribution is DisCoPar-Kilimo, a domain-specific low-code development environment for smart agriculture applications. DisCoPar-Kilimo is based on flow-based programming, where application tasks are represented as graphs of interconnected components that stream data to each other. The implementation of DisCoPar-Kilimo builds on DisCoPar and adds new extensions as explained in Section 4.5.

## 7.4 Shortcomings and Future Work

In this section, we state shortcomings and possible avenues of future work.

*Deployment.* In the current implementation, the deployment of applications is partially done by executing a series of commands. This requires specific technical skills, which domain experts may still need to gain. In future work, we plan to improve the deployment process so that domain experts can easily deploy DisCoPar-Kilimo applications.

*Dynamically adding and labelling new sensors.* Large farms require installing many edge devices and adding new ones after deployment. Dynamically adding devices to the application is not possible in our current implementation. Additionally, each edge device installed

must be labelled correctly to make it easy to identify and track. The number of edge devices to install can grow considerably, and naming and labelling each edge device can take considerable time. For future work, we suggest investigating how to dynamically add devices to an application and an automatic approach for labelling edge devices with the data they generate.

*Applying DisCoPar-Kilimo components to other domains.* The components that DisCoPar-Kilimo features can be applied to other domains, e.g., in citizen science applications. For example, the offline accessibility policies and components that DisCoPar-Kilimo provides can be used to enrich data collection for different participatory campaigning citizen science applications. However, some domains, like irrigation management, may require domain experts to add new components. Adding new components by domain experts is not supported in our current implementation. Hence, we plan to explore how domain experts can add new components to our low-code development environment in future work.

*Supporting actuators:* In this dissertation, we have focused on supporting sensing environmental conditions. However, some smart agriculture applications may also require actuating on the environment. For example, when the soil moisture content falls below a set threshold for a particular edge device, a smart agriculture application can go beyond notifying the farmer and activate the irrigation system to open water valves and irrigate the part of the farm where the particular edge device is deployed. From the scenarios presented in Section 2.2, this can help reduce water wastage, thus conserving it for other farming activities. This requires implementing actuator components to the edge scope of DisCoPar-Kilimo since they are not supported in the current implementation. Hence, adding the actuator components requires implementing (1) companion functions that will execute on edge devices and (2) friend functions that can interact with the actuation hardware.

*Throttling (i.e., handling back pressure).* In flow-based programming, data is consumed by the receiving components as soon as it arrives. However, this can be problematic when the data arrival rate becomes higher than the rate at which the receiving components consume the data. To support back pressure, existing literature proposes using

bounded buffers (i.e., communication channels)[Ngu15]. This issue is not addressed in the current implementation of DisCoPar-Kilimo. For future work, we aim to explore how the application developer can explicitly specify the buffer capacity for those communication channels.

*Receiving data from two different nodes.* In our current implementation, we assume that data from two different nodes arrives at the same rate and order to the receiving component. However, data from different nodes can arrive at different rates and orders to the receiving component. Existing literature proposes using logical conjunction operators that are commutative in time [VVANDM22]. This is not featured in the current implementation of DisCoPar-Kilimo. For future work, we aim to explore how to handle and coordinate data arriving at the receiving components at different rates and orders from two different nodes.

*Further evaluation and user study.* The implemented scenarios only focused on demonstrating the properties supported by DisCoPar-Kilimo. Further and more elaborate user studies are necessary to evaluate how DisCoPar-Kilimo can be used by domain experts, such as agricultural extension workers, to implement applications.

## 7.5 Concluding Remarks

Computers and software applications are changing our world at a fast pace. A growing demand for software applications has increasingly pressured the market for skilled software developers. LCDEs are an alternative for implementing software applications. In this work, we use LCDEs to simplify the development of smart agriculture applications. Our findings show that the techniques presented in LCDEs for software construction can be intuitive. Domain engineering, which incorporates domain concepts into the target LCDEs, is necessary for the LCDE techniques to be effective and for the software development goals to be met. This requires close collaboration with the experts in the target domain. Although our tool is a research prototype, we consider our work the foundation for the new and future generation of LCDEs.





## Appendix A

# DisCoPar Application Graph

Listing A.1 shows the JSON representation of an application flow-graph in DisCoPar. Components and their connections are represented as nested JSON objects. Each nested component object denotes a node in the application graph. Similarly, each nested connection object denotes a link between components in the application graph.

```
1 {
2   "graph": {
3     "version": 3,
4     "constraints": [],
5     "_id": "6496aabe4017577a81f0ed48",
6     "id": "9bf0d69f-90ea-4885-92ec-5ed53e2a2a2e",
7     "observatory_id": "6496aabe4017577a81f0ed4d",
8     "components": {
9       "05f1f2d8-e73d-4d19-a8b5-5102e5d62d59": {
10         "id": "05f1f2d8-e73d-4d19-a8b5-5102e5d62d59",
11         "name": "SoundPressureLevel",
12         "icon": "fas fa-microphone",
13         "description": "Records sound samples using the microphone and calculates the sound pressure level
14         in decibel",
15         "settings": {},
16         "position": {
17           "left": 361,
18           "top": 215
19         },
20         "clone": false,
21         "parent": "SoundPressureLevel",
22         "scope": {},
23         "platform": "Mobile",
24         "options": {
25           "componentColor": "#333",
26           "iconColor": "#FFF"
27         },
28         "components": null
29       },
30       "d2d3e429-8537-4ec0-9dfc-cf87b24c76de": {
31         "id": "d2d3e429-8537-4ec0-9dfc-cf87b24c76de",
32         "name": "DisplaySoundLevel",
33         "icon": "fas fa-tablet-alt",
```

```

33     "description": "Displays input on a label on the screen.",
34     "settings": {
35       "heading": {
36         "name": "heading",
37         "value": "someText"
38       },
39       "unit": {
40         "name": "unit",
41         "value": ""
42       },
43       "backgroundColor": {
44         "name": "backgroundColor",
45         "value": "#000000"
46       },
47       "textColor": {
48         "name": "textColor",
49         "value": "#FFFFFF"
50       },
51       "fontWeight": {
52         "name": "fontWeight",
53         "value": "400"
54       }
55     },
56     "position": {
57       "left": 582,
58       "top": 205.76953125
59     },
60     "clone": false,
61     "parent": "DisplaySoundLevel",
62     "scope": {
63       "scopes": [
64         {},
65         {}
66       ]
67     },
68     "platform": "Mobile",
69     "options": {
70       "componentColor": "#333",
71       "iconColor": "#FFF"
72     },
73     "components": null
74   }
75 },
76 "connections": {
77   "8918fd84-1c68-43d7-bab8-8822ac74cd1f": {
78     "id": "8918fd84-1c68-43d7-bab8-8822ac74cd1f",
79     "outPort": {
80       "name": "decibel",
81       "component": "05f1f2d8-e73d-4d19-a8b5-5102e5d62d59",
82       "graph": "9bf0d69f-90ea-4885-92ec-5ed53e2a2a2e"
83     },
84     "inPort": {
85       "name": "in",
86       "component": "d2d3e429-8537-4ec0-9dfc-cf87b24c76de",
87       "graph": "9bf0d69f-90ea-4885-92ec-5ed53e2a2a2e"
88     }
89   }
90 },
91 "socketConnections": {
92   "inbound": {},
93   "outbound": {}
94 },
95 "DOM": {},
96 "__v": 0
97 },
98 "processingGraph": {
99   "version": 0,
100   "constraints": [],
101   "_id": "6496aabe4017577a81f0ed49",
102   "id": "93672c25-c946-4247-a5db-29b24fd83ee",
103   "observatory_id": "6496aabe4017577a81f0ed4d",
104   "components": {},
105   "connections": {},

```

```

106   "socketConnections": {
107     "inbound": {},
108     "outbound": {}
109   },
110   "DOM": {},
111   "--v": 0
112 },
113 "dashboardGraph": {
114   "version": 0,
115   "constraints": [],
116   "_id": "6496aabe4017577a81f0ed4a",
117   "id": "88e03b7f-80a4-4f9a-c392-ae72a419ed6",
118   "observatory_id": "6496aabe4017577a81f0ed4d",
119   "components": {},
120   "connections": {},
121   "socketConnections": {
122     "inbound": {},
123     "outbound": {}
124   },
125   "DOM": {},
126   "--v": 0
127 },
128 "edgeGraph": {
129   "version": 0,
130   "constraints": [],
131   "_id": "6496aabe4017577a81f0ed4b",
132   "id": "954c38b9-f7d4-4719-c373-bbfe39f31310",
133   "observatory_id": "6496aabe4017577a81f0ed4d",
134   "components": {},
135   "connections": {},
136   "socketConnections": {
137     "inbound": {},
138     "outbound": {}
139   },
140   "DOM": {},
141   "--v": 0
142 },
143 "mobileApp": {
144   "_id": "6496aabe4017577a81f0ed4c",
145   "name": "Simple application App",
146   "implementation": "9bf0d69f-90ea-4885-92ec-5ed53e2a2a2e",
147   "configuration": {
148     "layout": {
149       "mode": "SINGLE"
150     }
151   },
152   "--v": 0,
153   "id": "6496aabe4017577a81f0ed4c"
154 }
155 }

```

Listing A.1: JSON representation of an application flow-graph in DisCoPar.



## Appendix B

# Companion Functions for Edge Components

Listing B.1 shows the portfolio of companion functions invoked to perform edge components' computation tasks.

```
1 function ReadTemperature() {  
2     var temperature = readTemperatureFromSensor();  
3     return temperature;  
4 }  
5  
6 function CelciusToKelvin(temp) {  
7     var kelvinTemp = parseInt(temp) + 273;  
8     return kelvinTemp;  
9 }  
10  
11 function CelciusToFarenheit(temp) {  
12     var farenheitTemp = (parseInt(temp)*9/5) + 32;  
13     return farenheitTemp;  
14 }  
15  
16 function ReadHumidity() {  
17     var humidity = readHumidityFromSensor();  
18     return humidity;  
19 }  
20  
21 function ReadSoilMoisture() {  
22     var soilM = readSoilMoistureFromSensor();  
23     return soilM;  
24 }  
25  
26 function ComputeEdgeAverage(data) {  
27     var count = 0;  
28     var sum = 0;  
29     sum += parseFloat(data);  
30     count++;  
31     return (sum/count);  
32 }  
33  
34 function BufferOnDisk(data){  
35     var storedData = storeOnDisk(data);  
36     return storedData;  
37 }  
38  
39 function Addition(a, b){
```

## APPENDIX B. COMPANION FUNCTIONS FOR EDGE COMPONENTS

---

```
40|     return a + b;
41| }
42|
43| function Subtraction(a, b){
44|     return a - b;
45| }
46|
47| function Multiplication(a, b){
48|     return a * b;
49| }
50|
51| function Exponentiation(a, b){
52|     return Math.pow(a, b);
53| }
54|
55| function Division(a, b){
56|     return a/b;
57| }
58|
59| function SetConstant(x){
60|     return x;
61| }
62|
63| function SetExponent(x){
64|     return x;
65| }
66|
67| function SetBase(x){
68|     return x;
69| }
70|
71| var preValue = null;
72| function PreviousValue(current){
73|     var temp = preValue;
74|     preValue = current;
75|     return temp;
76| }
77|
78| function SendData(conn, payload) {
79|     var send_to_network = "";
80|     send_to_network += conn;
81|     send_to_network += "#";
82|     send_to_network += payload;
83|     sendDataToNetwork(send_to_network);
84| }
```

Listing B.1: Library of companion edge functions that are invoked when edge components execute.

## Appendix C

# Extracting and Exporting Edge Graph

Listing C.1 shows the DisCoPar-Kilimo implementation to extract the edge application graph and build function calls that can be executed on edge devices.

```
1 export function getInitialGraphNodes(graph) {
2   let eData = graph;
3   let cmpSet = new Set();
4   for (let cmps in eData.components) {
5     cmpSet.add(eData.components[cmps].id);
6   }
7
8   for (let cns in eData.connections) {
9     cmpSet.delete(eData.connections[cns].inPort.component);
10  }
11  return cmpSet;
12 }
13
14 export function processGraph(id, graph) {
15   const workList = [{ nodeId: id, parentNodesIDs: [] }]; // the first item does not have any parent
16   const visitedNodes = new Set();
17   const orderedNodes = [];
18   while (workList.length !== 0) {
19     //Dequeue the first node from the list
20     const currentNode = workList.shift();
21     if (visitedNodes.has(currentNode.nodeId)) continue;
22
23     if (hasAllDependencies(visitedNodes, currentNode.parentNodesIDs)) {
24       //the dependencies of the node has been resolved.
25       visitedNodes.add(currentNode.nodeId);
26
27       //Now its children can be add to the queue.
28       const children = getChildrenOf(currentNode.nodeId, graph.connections);
29       children.forEach(child => workList.push({
30         nodeId: child,
31         parentNodesIDs: getParentsOf(child, graph.connections)
32       }));
33       orderedNodes.push(currentNode);
34     } else {
35       //the node cannot be processed cause it dependencies are not yet resolved.
36       workList.push(currentNode);
37     }
38   }
39 }
```



## APPENDIX C. EXTRACTING AND EXPORTING EDGE GRAPH

```
38 | }
39 | return orderedNodes;
40 | }
41 |
42 | function hasAllDependencies(available, myParents) {
43 |   for (const parent of myParents) {
44 |     if (!available.has(parent)) {
45 |       return false;
46 |     }
47 |   }
48 |   return true;
49 | }
50 |
51 | function getChildrenOf(outPort, edgesSet) {
52 |   const res = [];
53 |   for (const edge in edgesSet) {
54 |     if (edgesSet[edge].outPort.component === outPort) {
55 |       res.push(edgesSet[edge].inPort.component);
56 |     }
57 |   }
58 |   return res;
59 | }
60 |
61 | /** Compute the nodes' id of the incoming arrows to a node */
62 | function getParentsOf(nodeID, edgesSet) {
63 |   let parentsIDs = [];
64 |   for (const edge in edgesSet) {
65 |     if (edgesSet[edge].inPort.component === nodeID) {
66 |       parentsIDs.push(edgesSet[edge].outPort.component);
67 |     }
68 |   }
69 |   return parentsIDs;
70 | }
71 |
72 | export function buildEdgeGraph(computedNodes, edgeGraph) {
73 |   let str = "";
74 |
75 |   var edgeComponent = fs.readFileSync('REFERENCE_BASEFILE', 'utf8');
76 |   str += edgeComponent + "\n ";
77 |
78 |   str += '\n function discoparLoop(){\n';
79 |   computedNodes.forEach((node, index) => {
80 |     const args = node.parentNodesIDs;
81 |     try {
82 |       const varName = `_${node.nodeId.replace(/-/g, "")}`;
83 |       const edgeNode = edgeGraph.components[node.nodeId];
84 |       if (edgeNode) {
85 |         const edgeNodeName = edgeGraph.components[node.nodeId].name;
86 |         if (edgeNodeName && edgeGraph.components[node.nodeId].name !== "SetConstant" && edgeGraph.
87 | components[node.nodeId].name !== "BufferData") {
88 |           const string = `var ${varName} = ${edgeNodeName}(${args.map(id => "_" + id.replace(/-/g, ""))});`;
89 |           str += string + "\n ";
90 |         } else if (edgeNodeName && edgeGraph.components[node.nodeId].name === "SetConstant") {
91 |           const string = `var ${varName} = ${edgeNodeName}(${edgeGraph.components[node.nodeId].
92 | settings.constant.value});`;
93 |           str += string + "\n ";
94 |         } else if (edgeNodeName && edgeGraph.components[node.nodeId].name === "BufferData") {
95 |           const string = `var ${varName} = ${edgeNodeName}(${edgeGraph.components[node.nodeId].
96 | settings.bufferSize.value}, ${args.map(id => "_" + id.replace(/-/g, ""))});`;
97 |           str += string + "\n ";
98 |         }
99 |       }
100 |       if (Object.getOwnPropertyNames(edgeGraph.socketConnections.outbound).some(name =>
101 | edgeGraph.socketConnections.outbound[name].inPort.graph !== edgeGraph.socketConnections.
102 | outbound[name].outPort.graph)) {
103 |         let conIDs = Object.getOwnPropertyNames(edgeGraph.socketConnections.outbound).filter(name
104 | => name.startsWith(node.nodeId));
105 |         if (conIDs[0]){
106 |           str += ` SendData("data:${conIDs[0]}", ${varName});\n`;
107 |         }
108 |       }
109 |     }
110 |   })
111 | }
```

---

```
105         } catch (err) {  
106             console.error('Error when printingFunctionNames: ', err);  
107         }  
108     });  
109     str += "}"  
110     return str;  
111 }
```

Listing C.1: Extracting and exporting the edge graph.



## Appendix D

# Implementation of the Networking Code for Coordination with the Edge

Listing D.1 shows the implementation of the networking code that coordinates communication between edge and mobile devices.

```
1 var express = require("express");
2 var http = require("http");
3 var app = express();
4 var server = http.Server(app);
5 var PORT = 4000;
6
7 server.listen(PORT, function () {
8   console.log("Communication bridge started on port: " + PORT);
9 });
10
11 function getParam(url, param){
12   var paramParts = url.split("?")[1].split("&");
13   for (const entry of paramParts){
14     var split = entry.split("=");
15     if(param === split[0]){
16       return split[1]
17     }
18   }
19   return undefined;
20 }
21
22 var io = require("socket.io")(server);
23 var ioc = require('socket.io-client');
24 var middleware = require("socketio-wildcard")();
25 io.use(middleware);
26 var mobileSocket = [];
27 var discoparServerSocket = ioc.connect('address of discopar-k server');
28
29 io.on("connection", (socket) => {
30   var connectedDevices = null;
31   if(socket.connected && getParam(socket.client.request.url, "device_type") === "esp32"){
32     connectedDevices = Array.from(io.sockets.sockets).map(socket => (socket[1].handshake.address).
33       substring(7, (socket[1].handshake.address).length));
34   }
```

## APPENDIX D. IMPLEMENTATION OF THE NETWORKING CODE FOR COORDINATION WITH THE EDGE

---

```
35 | socket.on("disconnect", () => {
36 |     connectedDevices = Array.from(io.sockets.sockets).map(socket => socket[1].handshake.address);
37 | });
38 |
39 | if(getParam(socket.client.request.url,"device_type") === "phone"){
40 |     mobileSocket.push(socket);
41 |     socket.on(".*", (packet) => {
42 |         var event = packet.data[0];
43 |         var data = packet.data[1];
44 |         if(event.startsWith("data:")){
45 |             discoparServerSocket.emit(event, data);
46 |         }
47 |     });
48 | }
49 |
50 | if(getParam(socket.client.request.url,"device_type") === "esp32"){
51 |     socket.on("discopar_k_event", (data) => {
52 |         if(mobileSocket && mobileSocket.length){
53 |             mobileSocket.map((s)=> {
54 |                 s.emit("onmessage", connectedDevices);
55 |                 s.emit(data.connection, {data: data});
56 |             });
57 |         }
58 |     });
59 | }
60 | });
```

Listing D.1: Communication bridge running on the mobile phone

# Appendix E

## Deployment

### E.1 Resources Required

Deploying DisCoPar-Kilimo applications requires the following resources.

1. **Webpack**<sup>1</sup> is a module bundler. Its primary purpose is to bundle JavaScript files.
2. **Cordova**<sup>2</sup> packages HTML/JavaScript source code into a native application that can run on Android or iOS. It is a platform for building hybrid mobile applications using HTML, CSS and JavaScript.
3. **Visual studio code**<sup>3</sup> is an integrated development environment (IDE) for code editing and debugging applications.
4. **PlatformIO**<sup>4</sup> is a platform and IDE for developing embedded applications, e.g., Arduino applications.
5. **M5StickC microcontroller** is a portable IoT development board. ESP32-PICO-D4 powers the board with Bluetooth 4.2 and WiFi capabilities. The device is integrated with rich hardware resources, such as infrared, real-time clock (RTC), microphone, and light emitting diodes (LED).

---

<sup>1</sup><https://webpack.js.org/>

<sup>2</sup><https://cordova.apache.org/>

<sup>3</sup><https://code.visualstudio.com/>

<sup>4</sup><https://platformio.org/>

6. **Smartphone** is a portable computing device that combines the mobile telephone and computing functions into one unit.

### E.2 Mobile Application

1. Install a command line tool, e.g., Termux.
2. Specify the remote address for the DisCoPar-Kilimo server.
3. Copy the networking code file into the phone. The networking code is a JavaScript file that creates a minimalist server instance on the mobile phone when executed.
4. Navigate into the folder containing the networking code using the installed command line tool.
5. Start the the networking code using “node «communication-bridge-file-name».js” command.
6. Export the mobile application graph and copy it into the “www” directory in the Cordova project.
7. Export the base DisCoPar-Kilimo application and copy it into the js directory in the Cordova project.
8. Connect the mobile phone and run the command (*cordova run “«target-platform»”*) to deploy app to the phone.

### E.3 Edge Application

1. Download the edge graph and copy the file into */data/* directory of the DisCoPar-Kilimo edge application project. The file contains the edge application graph and the functions that are invoked when the edge application executes.
2. Specify the network SSID and password.
3. Specify the address to the mobile device.

4. Compile the file system using “*platformio run -target buildfs -environment m5stick-c*” command and the edge application using “*platformio run -environment m5stick-c*”. Both commands are supported by the PlatformIO tool.
5. Install the file system to the device using the *platformio run -target uploadfs -environment m5stick-c* command.
6. Install the edge application to the device using “*platformio run -target upload -environment m5stick-c*” command.





# Bibliography

- [AAUS<sup>+</sup>19] M. Ayaz, M. Ammad-Uddin, Z. Sharif, A. Mansour, and E. M. Aggoune. Internet-of-Things (IoT)-Based Smart Agriculture: Toward Making the Fields Talk. *IEEE Access*, 7:129551–129583, 2019. ISSN: 2169-3536, DOI: 10.1109/ACCESS.2019.2932609.
- [ACS10] Dave Arnold, Jean-Pierre Corriveau, and Wei Shi. Scenario-Based Validation: Beyond the User Requirements Notation. In *2010 21st Australian Software Engineering Conference*, pages 75–84, Auckland, New Zealand, 2010. IEEE. ISSN: 2377-5408, DOI: 10.1109/ASWEC.2010.29.
- [ADCB13] Matthew Aitkenhead, David Donnelly, Malcolm Coull, and Helaina Black. E-SMART: Environmental Sensing for Monitoring and Advising in Real-Time. In Jiří Hřebíček, Gerald Schimak, Miroslav Kubásek, and Andrea E. Rizzoli, editors, *Environmental Software Systems. Fostering Information Sharing*, pages 129–142, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN: 978-3-642-41151-9, DOI: 10.1007/978-3-642-41151-9\_13.
- [AIMP<sup>+</sup>18] Arturo Aquino, Barrio Ignacio, Diago María-Paz, Milan Borja, and Tardaguila Javier. vitisBerry: An Android-smartphone application to early evaluate the number of grapevine berries by means of image analysis. *Computers and Electronics in Agriculture*, 148:19–28, 2018. ISSN: 0168-1699, DOI: 10.1016/j.compag.2018.02.021.

- [AKKM19] Griffiths G. Atungulu, R.E. Kolb, J. Karcher, and Z. Mohammadi Shad. Postharvest technology: Rice storage and cooling conservation. In Jin-song Bao, editor, *Rice (Fourth Edition)*, pages 517–555. AACC International Press, fourth edition edition, 2019. ISBN: 978-0-12-811508-4, DOI: 10.1016/B978-0-12-811508-4.00016-2.
- [AMG<sup>+</sup>15] Arturo Aquino, Borja Millan, Daniel Gaston, María-Paz Diago, and Javier Tardaguila. vitisFlower<sup>®</sup>: Development and Testing of a Novel Android-Smartphone Application for Assessing the Number of Grapevine Flowers per Inflorescence Using Artificial Vision Techniques. *Sensors*, 15(9):21204–21218, 2015. ISSN: 1424-8220, DOI: 10.3390/s150921204.
- [AUS20] Tewodros W. Ayalew, Jordan R. Ubbens, and Ian Stavness. Unsupervised Domain Adaptation for Plant Organ Counting. In Adrien Bartoli and Andrea Fusiello, editors, *Computer Vision – ECCV 2020 Workshops*, pages 330–346, Cham, 2020. Springer International Publishing. ISBN: 978-3-030-65414-6, DOI: 10.1007/978-3-030-65414-6\_23.
- [BAAB15] A. C. Bartlett, A. A. Andales, M. Arabi, and T. A. Bauder. A smartphone app to extend use of a cloud-based irrigation scheduling tool. *Computers and Electronics in Agriculture*, 111:127–130, 2015. ISSN: 0168-1699, DOI: 10.1016/j.compag.2014.12.021.
- [BBF<sup>+</sup>19] Manlio Bacco, Paolo Barsocchi, Erina Ferro, Alberto Gotta, and Massimiliano Ruggeri. The Digitisation of Agriculture: a Survey of Research Activities on Smart Farming. *Array*, 3–4:1–11, 2019. ISSN: 2590-0056, DOI: 10.1016/j.array.2019.100009.
- [BBKR23] Hamza Benyezza, Mounir Bouhedda, Reda Kara, and Samia Rebouh. Smart platform based on IoT and WSN for monitoring and control of a greenhouse in the context of precision agriculture. *Internet of Things*,

- 23:100830, 2023. ISSN: 2542-6605, DOI: 10.1016/j.iot.2023.100830.
- [BCIR22] Cheick Tidiane Ba, Chloé Choquet, Roberto Interdonato, and Mathieu Roche. Explaining Food Security Warning Signals with YouTube Transcriptions and Local News Articles. In *Proceedings of the 2022 ACM Conference on Information Technology for Social Good, GoodIT’22*, pages 315–322, New York, NY, USA, 2022. Association for Computing Machinery. ISBN: 9781450392846, DOI: 10.1145/3524458.3547240.
- [BGS20] M. Bexiga, S. Garbatov, and J. C. Seco. Closing the gap between designers and developers in a low code ecosystem. In *Proceedings - 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS-C 2020 - Companion Proceedings*, Virtual Event Canada, 2020. ISBN: 978-1-4503-8135-2, DOI: 10.1145/3417990.3420195.
- [BL12] Michael Blackstock and Rodger Lea. IoT mashups with the WoTKit. In *Proceedings of 2012 International Conference on the Internet of Things, IOT 2012*, pages 159–166, Wuxi, China, 2012. ISBN: 978-1-4673-1345-2, DOI: 10.1109/IOT.2012.6402318.
- [BL14] Michael Blackstock and Rodger Lea. Toward a Distributed Data Flow Platform for the Web of Things (Distributed Node-RED). In *ACM International Conference Proceeding Series*, pages 34–39, New York, NY, USA, 2014. ISBN: 978-1-4503-3066-4, DOI: 10.1145/2684432.2684439.
- [BM21] Travis Breaux and Jennifer Moritz. The 2021 Software Developer Shortage is Coming. *Communications of the ACM*, 64(7):39–41, June 2021. ISSN: 0001-0782, DOI: 10.1145/3440753.
- [BSDN19] Cheikh Saliou Mbacke Babou, Bernard Ousmane Sane, Ibrahima Diane, and Ibrahima Niang. Home

- edge computing architecture for smart and sustainable agriculture and breeding. In *Proceedings of the 2nd International Conference on Networking, Information Systems & Security*, NISS19, pages 1–7, New York, NY, USA, 2019. Association for Computing Machinery. ISBN: 978-1-4503-6645-8, DOI: 10.1145/3320326.3320377.
- [BSJA16] J. Bauer, B. Siegmann, T. Jarmer, and N. Aschenbruck. Smart fLAIr: A smartphone application for fast LAI retrieval using Ambient Light Sensors. In *2016 IEEE Sensors Applications Symposium (SAS)*, pages 1–6, Catania, Italy, 2016. ISBN: 978-1-4799-7250-0, DOI: 10.1109/SAS.2016.7479880.
- [BSS<sup>+</sup>15] M. Busse, W. Schwerdtner, R. Siebert, A. Doernberg, A. Kuntosch, B. König, and W. Bokelmann. Analysis of animal monitoring technologies in Germany from an innovation system perspective. *Agricultural Systems*, 138:55–65, 2015. ISSN:0308-521X, DOI: 10.1016/j.agsy.2015.05.009.
- [BSZ<sup>+</sup>21] Heinz Bernhardt, Leon Schumacher, Jianfeng Zhou, Maximilian Treiber, and Kent Shannon. Digital Agriculture Infrastructure in the USA and Germany. *Engineering Proceedings*, 9(1), 2021. ISSN: 2673-4591, DOI: 10.3390/engproc2021009001.
- [Bur99] Margaret M. Burnett. Visual Programming. In *Wiley Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons, Ltd, 1999. ISBN: 978-0-4713-4608-1, DOI: 10.1002/047134608X.W1707.
- [BWO<sup>+</sup>20] Markus Borg, Joakim Wernberg, Thomas Olsson, Ulrik Franke, and Martin Andersson. Illuminating a Blind Spot in Digitalization - Software Development in Sweden’s Private and Public Sector. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW’20,

- pages 299–302, New York, NY, USA, 2020. Association for Computing Machinery. ISBN: 978-1-4503-7963-2, DOI: 10.1145/3387940.3392213.
- [CFC<sup>+</sup>13] R. Confalonieri, M. Foi, R. Casa, S. Aquaro, E. Tona, M. Peterle, A. Boldini, G. D. Carli, A. Ferrari, G. Finotto, T. Guarneri, V. Manzoni, E. Movedi, A. Nisoli, L. Paleari, I. Radici, M. Suardi, D. Veronesi, S. Bregaglio, G. A. Cappelli, M. E. Chiodini, P. Dominoni, C. Francone, N. Frasso, T. Stella, and M. Acutis. Development of an app for estimating leaf area index using a smartphone. Trueness and precision determination and comparison with other indirect methods. *Computers and Electronics in Agriculture*, 96:67–74, 2013. ISSN: 0168-1699, DOI: 10.1016/j.compag.2013.04.019.
- [CFM<sup>+</sup>19] Maria Otilia Carvalho, Patrícia Fradinho, M. João Martins, Ana Magro, Anabela Raymundo, and Isabel de Sousa. Paddy rice stored under hermetic conditions: The effect of relative humidity, temperature and storage time in suppressing *Sitophilus zeamais* and impact on rice quality. *Journal of Stored Products Research*, 80:21–27, 2019. ISSN: 0022-474X, DOI: 10.1016/j.jspr.2018.11.002.
- [Cha87a] Shi-Kuo Chang. Icon Semantics – a Formal Approach to Icon System Design. *International Journal of Pattern Recognition and Artificial Intelligence*, 1(1):103–120, apr 1987. ISSN: 0218-0014, DOI: 10.1142/S0218001487000084.
- [Cha87b] Shi-Kuo Chang. Visual Languages: A Tutorial and Survey. *IEEE Software*, 4(1):29–39, 1987. ISSN: 1937-4194, DOI: 10.1109/MS.1987.229792.
- [CJS<sup>+</sup>17] Francisco Carpio, Admela Jukan, Ana Isabel Martín Sanchez, Nina Amla, and Nicole Kemper. Beyond Production Indicators: A Novel Smart Farming Application and System for Animal Welfare. In *Proceedings*

- of the Fourth International Conference on Animal-Computer Interaction*, ACI2017, New York, NY, USA, 2017. Association for Computing Machinery. ISBN: 978-1-4503-5364-9, DOI: 10.1145/3152130.3152140.
- [CK02] Jarinee Chattratchart and Jasna Kuljis. Exploring the Effect of Control-Flow and Traversal Direction on VPL Usability for Novices. *Journal of Visual Languages & Computing*, 13(5):471–500, 2002. ISSN: 1045-926X, DOI: 10.1006/jvlc.2002.0240.
- [CKR19] Francisco Ceballos, Berber Kramer, and Miguel Robles. The feasibility of picture-based insurance (PBI): Smartphone pictures for affordable crop insurance. *Development Engineering*, 4:100042, 2019. ISSN: 2352-7285, DOI: 10.1016/j.deveng.2019.100042.
- [CLDAOPMPA20] Antonio Manuel Ciruela-Lorenzo, Ana Rosa Del-Aguila-Obra, Antonio Padilla-Meléndez, and Juan José Plaza-Angulo. Digitalization of Agri-Cooperatives in the Smart Agriculture Context. Proposal of a Digital Diagnosis Tool. *Sustainability*, 12(4), 2020. ISSN: 2071-1050, DOI: 10.3390/su12041325.
- [CSJK17] Marcel Caria, Jasmin Schudrowitz, Admela Jukan, and Nicole Kemper. Smart farm computing systems for animal welfare monitoring. In *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics*, MIPRO, pages 152–157, Opatija, Croatia, 2017. IEEE. ISBN: 978-9-5323-3092-2, DOI: 10.23919/MIPRO.2017.7973408.
- [DAEA18] Gideon Danso-Abbeam, Dennis Sedem Ehiakpor, and Robert Aidoo. Agricultural extension and its effects on farm productivity and income: insight from Northern Ghana. *Agriculture & Food Security*, 7(74), 2018. ISSN: 2048-7010, DOI: 10.1186/s40066-018-0225-x.
- [DBR20] Kristin E. Davis, Suresh Chandra Babu, and Catherine Ragasa. *Agricultural extension: Global status*

- and performance in selected countries.* Washington, DC: International Food Policy Research Institute (IFPRI), 2020. ISBN: 978-0-89629-375-5, DOI: 10.2499/9780896293755.
- [DFG<sup>+</sup>16] Roberta De Bei, Sigfredo Fuentes, Matthew Gilliam, Steve Tyerman, Everard Edwards, Nicolò Bianchini, Jason Smith, and Cassandra Collins. VitiCanopy: A Free Computer App to Estimate Canopy Vigor and Porosity for Grapevine. *Sensors*, 16(4), 2016. ISSN: 1424-8220, DOI: 10.3390/s16040585.
- [ECA<sup>+</sup>19] Anton Eitzinger, James Cock, Karl Atzmanstorfer, Claudia R. Binder, Peter Läderach, Osana Bonilla-Findji, Mona Bartling, Caroline Mwongera, Leo Zurita, and Andy Jarvis. GeoFarmer: A monitoring and feedback system for agricultural development projects. *Computers and Electronics in Agriculture*, 158:109–121, 2019. ISSN: 0168-1699, DOI: 10.1016/j.compag.2019.01.049.
- [EEE21] Ann Nnenna Ezech, Anayochukwu Victor Eze, and Esther Onyinyechi Eze. Extension Agents’ Use of Mobile Phone Applications for Agricultural Extension Service Delivery in Ebonyi State Agricultural Development Programme, Nigeria. *Journal of Agricultural Extension*, 25, 2021. ISSN: 2408-6851, DOI: 10.4314/jae.v25i1.6.
- [ESEKA<sup>+</sup>21] AbdAllah M. El-Sanatawy, Ahmed S. M. El-Kholy, Mohamed M. A. Ali, Mohamed F. Awad, and Elsayed Mansour. Maize Seedling Establishment, Grain Yield and Crop Water Productivity Response to Seed Priming and Irrigation Management in a Mediterranean Arid Environment. *Agronomy*, 11(4), 2021. ISSN: 2073-4395, DOI: 10.3390/agronomy11040756.
- [ETDS20] Ezinne M. Emeana, Liz Trenchard, and Katharina Dehnen-Schmutz. The Revolution of Mobile Phone-Enabled Services for Agricultural Development (m-



- Agri Services) in Africa: The Challenges for Sustainability. *Sustainability*, 12(2), 2020. ISSN: 2071-1050, DOI: 10.3390/su12020485.
- [FBPT12] Sigfredo Fuentes, Roberta De Bei, C. Pozo, and Stephen D. Tyerman. Development of a smart-phone application to characterise temporal and spatial canopy architecture and leaf area index for grapevines. *Wine and Viticulture Journal*, 27(6):56–60, 11 2012. ISSN: 1838-6547.
- [FCO<sup>+</sup>16] Jason Connor Ferguson, Rodolfo Glauber Chechetto, Chris C. O'Donnell, Bradley Keith Fritz, Wesley Clint Hoffmann, Chet E. Coleman, Bhagirath Singh Chauhan, Steve William Adkins, Greg Robert Kruger, and Andrew J. Hewitt. Assessing a Novel Smartphone Application - SnapCard, Compared to Five Imaging Systems to Quantify Droplet Deposition on Artificial Collectors. *Computers and Electronics in Agriculture*, 128(C):193–198, October 2016. ISSN: 0168-1699, DOI: 10.1016/j.compag.2016.08.022.
- [FPEOF<sup>+</sup>14] Sigfredo Fuentes, C. Poblete-Echeverría, S. Ortega-Farias, S. Tyerman, and Roberta De Bei. Automated estimation of leaf area index from grapevine canopies using cover photography, video and computational analysis methods. *Australian Journal of Grape and Wine Research*, 20(3):465–473, 2014. ISSN: 1755-0238, DOI: 10.1111/ajgw.12098.
- [FSC13] Lutz Frommberger, Falko Schmid, and Chunyuan Cai. Micro-Mapping with Smartphones for Monitoring Agricultural Development. In *Proceedings of the 3rd ACM Symposium on Computing for Development*, ACM DEV'13, pages 1–2, New York, NY, USA, 2013. Association for Computing Machinery. ISBN: 978-1-4503-1856-3, DOI: 10.1145/2442882.2442934.
- [FSW<sup>+</sup>23] Laura Foster, Katie Szilagyi, Angeline Wairegi, Chidi Oguamanam, and Jeremy de Beer. Smart farming

- and artificial intelligence in East Africa: Addressing indigeneity, plants, and gender. *Smart Agricultural Technology*, 3:100132, 2023. ISSN: 2772-3755, DOI: 10.1016/j.atech.2022.100132.
- [GBLL15] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C. M. Leung. Developing IoT applications in the Fog: A Distributed Dataflow approach. In *2015 5th International Conference on the Internet of Things (IOT)*, pages 155–162, Seoul, Korea (South), 2015. ISBN: 978-1-4673-8058-4, DOI: 10.1109/IOT.2015.7356560.
- [GBP<sup>+</sup>23a] Girma Gebresenbet, Techane Bosona, David Patterson, Henrik Persson, Benjamin Fischer, Nerea Mandaluniz, Gherardo Chirici, Aleksejs Zacepins, Vitalijs Komasilovs, Tudor Pitulac, and Abozar Nasirahmadi. A concept for application of integrated digital technologies to enhance future smart agricultural systems. *Smart Agricultural Technology*, 5:100255, 2023. ISSN: 2772-3755, DOI: 10.1016/j.atech.2023.100255.
- [GBP<sup>+</sup>23b] Girma Gebresenbet, Techane Bosona, David Patterson, Henrik Persson, Benjamin Fischer, Nerea Mandaluniz, Gherardo Chirici, Aleksejs Zacepins, Vitalijs Komasilovs, Tudor Pitulac, and Abozar Nasirahmadi. A concept for application of integrated digital technologies to enhance future smart agricultural systems. *Smart Agricultural Technology*, 5:100255, 2023.
- [GKOK23] Stephen Gitahi, Dennis Mugambi Kaburu, Isaac Nyabisa Oteyo, and Stephen Kimani. A Model Implementation of Internet of Things (IoT)-based Smart Watering System for Crops using LoRaWAN. In *2023 IST-Africa Conference (IST-Africa)*, pages 1–8, Cape Town, South Africa, 2023. ISSN: 2576-8581, DOI: 10.23919/IST-Africa60249.2023.10187864.
- [GS22] Pedro Galhardo and Alberto Rodrigues da Silva. Combining Rigorous Requirements Specifications with

- Low-Code Platforms to Rapid Development Software Business Applications. *Applied Sciences*, 12(19), 2022. ISSN: 2076-3417, DOI: 10.3390/app12199556.
- [GTTSBR<sup>+</sup>19] Juan D. González-Teruel, Roque Torres-Sánchez, Pedro J. Blaya-Ros, Ana B. Toledo-Moreo, Manuel Jiménez-Buenda, and Fulgencio Soto-Valles. Design and Calibration of a Low-Cost SDI-12 Soil Moisture Sensor. 19(3), 2019. ISSN: 1424-8220, DOI: 10.3390/s19030491.
- [HBB<sup>+</sup>16] Jeffrey E. Herrick, Adam Beh, Edmundo Barrios, Ioana Bouvier, Marina Coetzee, David Dent, Emile Elias, Tomislav Hengl, Jason W. Karl, Hanspeter Liniger, John Matuszak, Jason C. Neff, Lilian Wangui Ndungu, Michael Obersteiner, Keith D. Shepherd, Kevin C. Urama, Rik Bosch, and Nicholas P. Webb. The Land-Potential Knowledge System (LandPKS): mobile apps and collaboration for optimizing climate change investments. *Ecosystem Health and Sustainability*, 2(3):e01209, 2016. ISSN: 2332-8878, DOI: 10.1002/ehs2.1209.
- [HD18] Jerry L. Hatfield and Christian Dold. Climate Change Impacts on Corn Phenology and Productivity. In Amanullah and Shah Fahad, editors, *Corn - Production and Human Health in Changing Climate*, chapter 6. IntechOpen, Rijeka, 2018. ISBN: 978-1-78984-156-5, DOI: 10.5772/intechopen.76933.
- [HHRHGM<sup>+</sup>16] Jose Luis Hernández Hernández, J. Ruiz-Hernández, Ginés García-Mateos, Jose Manuel Esquiva, Antonio Ruiz-Canales, and José Martínez. A new portable application for automatic segmentation of plants in agriculture. *Agricultural Water Management*, 183, 08 2016. ISSN: 1873-2283, DOI: 10.1016/j.agwat.2016.08.013.
- [HP15] Jerry L. Hatfield and John H. Prueger. Temperature extremes: Effect on plant growth and development.

- Weather and Climate Extremes*, 10:4–10, 2015. USDA Research and Programs on Extreme Events, ISSN: 2212-0947, DOI: 10.1016/j.wace.2015.08.001.
- [ISY<sup>+</sup>17] Mehmet Fatih IÅik, Yusuf Sönmez, Cemal Yilmaz, Veysel Özdemir, and Ercan Nurcan Yilmaz. Precision Irrigation System (PIS) using sensor network technology integrated with IOS/Android Application. *Applied Sciences (Switzerland)*, 7(891):1–14, 2017. ISSN: 2076-3417, DOI: 10.3390/app7090891.
- [JEM<sup>+</sup>16] René Jordan, Gaius Eudoxie, Kiran Maharaj, Renaldo Belfon, and Margaret Bernard. AgriMaps: Improving site-specific land management through mobile maps. *Computers and Electronics in Agriculture*, 123:292–296, 2016. ISSN: 0168-1699, DOI: 10.1016/j.compag.2016.02.009.
- [JHKS23] Mohd Javaid, Abid Haleem, Ibrahim Haleem Khan, and Rajiv Suman. Understanding the potential applications of Artificial Intelligence in Agriculture Sector. *Advanced Agrochem*, 2(1):15–30, 2023. ISSN: 2773-2371, DOI: 10.1016/j.aac.2022.10.001.
- [JYG<sup>+</sup>16] Prem Prakash Jayaraman, Ali Yavari, Dimitrios Georgakopoulos, Ahsan Morshed, and Arkady Zaslavsky. Internet of things platform for smart farming: Experiences and lessons learnt. *Sensors (Switzerland)*, 16(11):1–17, 2016. ISSN: 1424-8220, DOI: 10.3390/s16111884.
- [KAMH21] Hazem S. Kassem, Bader Alhafi Alotaibi, Muhammad Muddassir, and Ahmed Herab. Factors influencing farmers’ satisfaction with the quality of agricultural extension services. *Evaluation and Program Planning*, 85:101912, 2021. ISSN: 0149-7189, DOI: 10.1016/j.evalprogplan.2021.101912.
- [KFHB21] Mohammad Amin Kuhail, Shahbano Farooq, Rawad Hammad, and Mohammed Bahja. Characterizing

- Visual Programming Approaches for End-User Developers: A Systematic Review. *IEEE Access*, 9:14181–14202, 2021. ISSN: 2169-3536, DOI: 10.1109/ACCESS.2021.3051043.
- [KKE<sup>+</sup>19] Andrea Knierim, Maria Kernecker, Klaus Erdle, Teresa Kraus, Friederike Borges, and Angelika Wurbs. Smart farming technology innovations – Insights and reflections from the German Smart-AKIS hub. *NJAS - Wageningen Journal of Life Sciences*, 90-91:100314, 2019. ISSN: 1573-5214, DOI: 10.1016/j.njas.2019.100314.
- [LBJ01] Mounir Louhaichi, Michael M. Borman, and Douglas E. Johnson. Spatially Located Platform and Aerial Photography for Documentation of Grazing Impacts on Wheat. *Geocarto International*, 16(1):65–70, 2001. ISSN: 1752-0762, DOI: 10.1080/10106040108542184.
- [LCAD13] Richard K. Lomotey, Yiding Chai, Kazi A. Ahmed, and Ralph Deters. Web services mobile application for geographically dispersed crop farmers. In *Proceedings - 16th IEEE International Conference on Computational Science and Engineering, CSE 2013*, pages 151–158, Sydney, NSW, Australia, 2013. IEEE. ISBN: 978-0-7695-5096-1, DOI: 10.1109/CSE.2013.33.
- [LCJD13] Richard K. Lomotey, Yiding Chai, Shomoyita Jamal, and Ralph Deters. MobiCrop: Supporting crop farmers with a cloud-enabled mobile app. In *Proceedings - IEEE 6th International Conference on Service-Oriented Computing and Applications, SOCA 2013*, pages 182–189, Koloa, HI, USA, 2013. IEEE. ISBN: 978-1-4799-2701-2, DOI: 10.1109/SOCA.2013.19.
- [LK13] Bo Liu and A. Bulent Koc. SafeDriving: A mobile application for tractor rollover detection and emergency reporting. *Computers and Electronics in Agriculture*,

98:117–120, 2013. ISSN: 0168-1699, DOI: 10.1016/j.compag.2013.08.002.

[LMBS23] Martin J. Luna Juncal, Pietro Masino, Edoardo Bertone, and Rodney A. Stewart. Towards nutrient neutrality: A review of agricultural runoff mitigation strategies and the development of a decision-making framework. *Science of The Total Environment*, 874:162408, 2023. ISSN: 0048-9697, DOI: 10.1016/j.scitotenv.2023.162408.

[MAG16] E. Misaki, M. Apiola, and S. Gaiani. Technology for small scale farmers in Tanzania: A design science research approach. *Electronic Journal of Information Systems in Developing Countries*, 74(4):1–15, 2016. ISSN: 1681-4835, DOI: 10.1002/j.1681-4835.2016.tb00538.x.

[MAGT18] Ezra Misaki, Mikko Apiola, Silvia Gaiani, and Matti Tedre. Challenges facing sub-Saharan small-scale farmers in accessing farming information through mobile phones: A systematic literature review. *The Electronic Journal of Information Systems in Developing Countries*, 84(4):e12034, 2018. ISSN: 1681-4835, DOI: 10.1002/isd2.12034.

[MCCGdGF12] Francisco Javier Mesas-Carrascosa, Isabel Luisa Castillejo-González, Manuel Sánchez de la Orden, and Alfonso García-Ferrer. Real-time mobile phone application to support land policy. *Computers and Electronics in Agriculture*, 85:109–111, 2012. ISSN: 0168-1699, DOI: 10.1016/j.compag.2012.04.003.

[MLDGd23] Francisco Martínez-Lasaca, Pablo Díez, Esther Guerra, and Juan de Lara. Dandelion: A scalable, cloud-based graphical language workbench for industrial low-code development. *Journal of Computer Languages*, 76:101217, 2023. ISSN: 2590-1184, DOI: 10.1016/j.cola.2023.101217.

- [MLJ<sup>+</sup>20] Danse Myrtille, Klerkx Laurens, Reintjes Jorrit, Rabbinge Rudy, and Leeuwis Cees. Unravelling inclusive business models for achieving food and nutrition security in BOP markets. *Global Food Security*, 24:1–15, 2020. ISSN: 2211-9124, DOI: 10.1016/j.gfs.2020.100354.
- [MMC19] L. Myeni, M. E. Moeletsi, and A. D. Clulow. Present status of soil moisture estimation over the African continent. *Journal of Hydrology: Regional Studies*, 21:14–24, 2019. ISSN: 2214-5818, DOI: 10.1016/j.ejrh.2018.11.004.
- [MMCR13] Jean-Vivien Millo, Frédéric Mallet, Anthony Coadou, and S. Ramesh. Scenario-based verification in presence of variability using a synchronous approach. *Frontiers of Computer Science*, 7(5):650–672, 2013. ISSN: 2095-2236, DOI: 10.1007/s11704-013-3094-6.
- [MMJRCFP11] Jose M. Molina-Martínez, Manuel Jiménez, Antonio Ruiz-Canales, and Daniel G. Fernández-Pacheco. RaGPS: A software application for determining extraterrestrial radiation in mobile devices with GPS. *Computers and Electronics in Agriculture*, 78(1):116–121, 2011. ISSN: 0168-1699, DOI: 10.1016/j.compag.2011.06.009.
- [MMR<sup>+</sup>21] Zia Mehrabi, Mollie J. McDowell, Vincent Ricciardi, Christian Levers, Juan Diego Martinez, Natascha Mehrabi, Hannah Wittman, Navin Ramankutty, and Andy Jarvis. The global divide in data-driven farming. *Nature Sustainability*, 4:154–160, 2021.
- [MOA<sup>+</sup>16] Bruno Brandoli Machado, Jonatan P. M. Orue, Mauro S. Arruda, Cleidimar V. Santos, Diogo S. Sarath, Wesley N. Goncalves, Gercina G. Silva, Hemerson Pistori, Antonia Railda Roel, and Jose F. Rodrigues-Jr. BioLeaf: A professional mobile application to measure foliar damage caused by insect herbivory. *Computers and Electronics in Agriculture*,

- 129:44–55, 2016. ISSN: 0168-1699, DOI: 10.1016/j.compag.2016.09.007.
- [Mor10] John Paul Morrison. *Flow-Based Programming: A New Approach to Application Development*. CreateSpace, Paramount, CA, California, 2nd edition, 2010. ISBN: 978-1-4515-4232-5.
- [MP23] Eder Martinez and Louis Pfister. Benefits and limitations of using low-code development to support digitalization in the construction industry. *Automation in Construction*, 152:104909, 2023. ISSN: 0926-5805, DOI: 10.1016/j.autcon.2023.104909.
- [MPT<sup>+</sup>17] Quang Tran Minh, Trong Nhan Phan, Akihiko Takahashi, Tam Thai Thanh, Son Nguyen Duy, Mong Nguyen Thanh, and Chau Nguyen Hong. A Cost-Effective Smart Farming System with Knowledge Base. In *Proceedings of the Eighth International Symposium on Information and Communication Technology*, SoICT 2017, pages 309–316, Nha Trang City, Viet Nam, 2017. Association for Computing Machinery. ISBN: 978-1-4503-5328-1, DOI: 10.1145/3155133.3155151.
- [MSA<sup>+</sup>18] Bruno Brandoli Machado, Gabriel Spadon, Mauro S. Arruda, Wesley N. Goncalves, Andre C. P. L. F. Carvalho, and Jose F. Rodrigues-Jr. A smartphone application to measure the quality of pest control spraying machines via image analysis. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, Pau, France, April 2018. ACM. ISBN: 978-1-4503-5191-1, DOI: 10.1145/3167132.3167237.
- [MSR<sup>+</sup>18] Ajay Mittal, Sanat Sarangi, Saranya Ramanath, Prakruti V. Bhatt, Rahul Sharma, and P. Srinivasu. IoT-Based Precision Monitoring of Horticultural Crops-A Case-Study on Cabbage and Capsicum. In *Proceedings of GHTC 2018 - IEEE Global Humanitarian Technology Conference*, pages 1–7, San



- Jose, CA, 2018. IEEE. ISBN: 978-1-5386-5566-5, DOI: 10.1109/GHTC.2018.8601908.
- [MUH<sup>+</sup>13] Yukikazu Murakami, Slamet Kristanto Tirta Utomo, Keita Hosono, Takeshi Umezawa, and Noritaka Osawa. iFarm: Development of cloud-based system of cultivation management for precision agriculture. In *2013 IEEE 2nd Global Conference on Consumer Electronics, GCCE 2013*, pages 233–234, Tokyo, Japan, 2013. IEEE. ISBN: 978-1-4799-0892-9, DOI: 10.1109/GCCE.2013.6664809.
- [MVd19] Walter Maldonado, Taynara Tuany Borges Valeriano, and Glauco de Souza Rolim. EVAPO: A smart-phone application to estimate potential evapotranspiration using cloud gridded meteorological data from NASA-POWER system. *Computers and Electronics in Agriculture*, 156:187–192, 2019. ISSN: 0168-1699, DOI: 10.1016/j.compag.2018.10.032.
- [Net] NTK Getting Started Steps. <https://www.netlabtoolkit.org/overview/>, Accessed: 2022-11-25.
- [Ngu15] Vu Thien Nga Nguyen. *An Efficient Execution Model for Reactive Stream Programs*. PhD thesis, University of Hertfordshire, 2015. ISBN: 978-9-49231-289-1, DOI: 10.18745/th.16361.
- [NiF] Apache Ni-Fi Overview. <https://nifi.apache.org/docs.html>, Accessed: 2022-11-24.
- [Nof] Getting started with NoFlo. <https://noflojs.org/documentation/>, Accessed: 2022-11-24.
- [NPSO96] F. Nelson, T. Pickett, W. Smith, and L. Ott. The GreenStar precision farming system. In *Proceedings of Position, Location and Navigation Symposium - PLANS'96*, pages 6–9, Atlanta, GA, USA, 1996. ISBN: 0-7803-3085-4, DOI: 10.1109/PLANS.1996.509048.

- [NTGS19] Joseph Noor, Hsiao Yun Tseng, Luis Garcia, and Mani Srivastava. DDFlow: Visualized declarative programming for heterogeneous IoT networks. In *IoTDI 2019 - Proceedings of the 2019 Internet of Things Design and Implementation*, IoTDI'19, pages 172–177, Montreal, Quebec, Canada, 2019. ISBN: 978-1-4503-6283-2, DOI: 10.1145/3302505.3310079.
- [NU16] Chioma Udo Nwaobiala and Victoria Uchechi Ubor. Effectiveness of electronic wallet system of growth enhancement support scheme distribution among arable crop farmers in Imo State, South East Nigeria. *Scientific Papers Series Management, Economic Engineering in Agriculture and Rural Development*, 16(1):355–360, 2016. ISSN: 2284-7995.
- [OCC<sup>+</sup>13] Tyson E. Ochsner, Michael H. Cosh, Richard H. Cuenca, Wouter A. Dorigo, Clara S. Draper, Yutaka Hagimoto, Yann H. Kerr, Kristine M. Larson, Eni G. Njoku, Eric E. Small, and Marek Zreda. State of the Art in Large-Scale Soil Moisture Monitoring. *Soil Science Society of America Journal*, 77(6):1888–1919, 2013. ISSN: 1435-0661, DOI: 10.2136/sssaj2013.03.0093.
- [OKB<sup>+</sup>18] Isaac Nyabisa Oteyo, Kennedy Kambona, Clément Béra, Mary Esther Muyoka Toili, Stephen Kimani, Wolfgang De Meuter, and Elisa Gonzalez Boix. Dynamically processing agricultural data from controlled legume sites. In *The Sixth African Higher Education Week and RUFORUM Biennial Conference*, volume 17 of *RUFORUM Working Document Series*, pages 183–192, Nairobi, Kenya, 2018.
- [OKW<sup>+</sup>19] Turry Ouma, Agnes Kavoo, Cornelius Wainaina, Busayo Ogunya, Margaret Karanja, P. Lava Kumar, and Trushar Shah. Open data kit (ODK) in crop farming: mobile data collection for seed yam tracking in Ibadan, Nigeria. *Journal of Crop Improvement*,

- 33(5):605–619, 2019. ISSN: 1542-7536, DOI: 10.1080/15427528.2019.1643812.
- [OKZ<sup>+</sup>20] Isaac Nyabisa Oteyo, Kennedy Kambona, Jesse Zaman, Wolfgang De meuter, and Elisa Gonzalez Boix. Developing Smart Agriculture Applications: Experiences and Lessons Learnt. In *Proceedings of the 2020 African Conference on Software Engineering (ACSE 2020)*, volume 2689, Nairobi, Kenya, 2020.
- [OLO19] M. J. O’Grady, D. Langton, and G. M. P. O’Hare. Edge computing: A tractable model for smart agriculture? *Artificial Intelligence in Agriculture*, 3:42–51, 2019. ISSN: 2589-7217, DOI: 10.1016/j.aiia.2019.12.001.
- [OMC<sup>+</sup>16] Francesca Orlando, Ermes Movedi, Davide Coduto, Simone Parisi, Lucio Brancadoro, Valentina Pagani, Tommaso Guarneri, and Roberto Confalonieri. Estimating Leaf Area Index (LAI) in Vineyards Using the PocketLAI Smart-App. *Sensors*, 16(12), 2016. ISSN: 1424-8220, DOI: 10.3390/s16122004.
- [OMK<sup>+</sup>21] Isaac Nyabisa Oteyo, Matteo Marra, Stephen Kimani, Wolfgang De Meuter, and Elisa Gonzalez Boix. A Survey on Mobile Applications for Smart Agriculture Making Use of Mobile Software in Modern Farming. *SN Computer Science*, 2(4):1–16, 2021. ISSN: 2661-8907, DOI: 10.1007/s42979-021-00700-x.
- [OSZ<sup>+</sup>21] Isaac Nyabisa Oteyo, Angel Luis Scull Pupo, Jesse Zaman, Stephen Kimani, Wolfgang De Meuter, and Elisa Gonzalez Boix. Building Smart Agriculture Applications Using Low-Code Tools: The Case for DisCoPar. In *2021 IEEE AFRICON*, pages 1–6, Arusha, Tanzania, 2021. IEEE. ISBN: 978-1-6654-1984-0, DOI: 10.1109/AFRICON51333.2021.9570936.
- [OSZ<sup>+</sup>23] Isaac Nyabisa Oteyo, Angel Luis Scull Pupo, Jesse Zaman, Stephen Kimani, Wolfgang De Meuter, and

- Elisa Gonzalez Boix. Easing Construction of Smart Agriculture Applications Using Low Code Development Tools. In Shangguan Longfei and Priyantha Bodhi, editors, *Mobile and Ubiquitous Systems: Computing, Networking and Services*, volume 492, pages 21–43. Springer, Cham, 2023. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. ISBN: 978-3-031-34776-4, DOI: 10.1007/978-3-031-34776-4.2.
- [OT20] Isaac Nyabisa Oteyo and Mary Esther Muyoka Toili. Improving Specimen Labelling and Data Collection in Bio-science Research using Mobile and Web Applications. *Open Computer Science*, 10(1):1–16, 2020. ISSN: 2299-1093, DOI: 10.1515/comp-2020-0002.
- [PCSMC<sup>+</sup>17] A. Pérez-Castro, J. A. Sánchez-Molina, María Castilla, José Sánchez-Moreno, J. C. Moreno-Úbeda, and Juan José Magán. cFertigUAL: A fertigation management app for greenhouse vegetable crops. *Agricultural Water Management*, 183:186–193, 2017. Special Issue: Advances on ICTs for Water Management in Agriculture. ISSN: 0378-3774, DOI: 10.1016/j.agwat.2016.09.013.
- [PE08] F. J. Pierce and T. V. Elliott. Regional and on-farm wireless sensor networks for agricultural systems in Eastern Washington. *Computers and Electronics in Agriculture*, 61(1):32–43, 2008. Emerging Technologies For Real-time and Integrated Agriculture Decisions. ISSN: 0168-1699, DOI: 10.1016/j.compag.2007.05.007.
- [Pet17] Nikos Petrellis. A smart phone image processing application for plant disease diagnosis. In *2017 6th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pages 4–7, Thessaloniki, Greece, 2017. ISBN: 978-1-5090-4386-6, DOI: 10.1109/MOCAST.2017.7937683.

- [Pet19] Nikos Petrellis. Plant Disease Diagnosis for Smart Phone Applications with Extensible Set of Diseases. *Applied Sciences*, 9, 2019. ISSN: 2076-3417, DOI: 10.3390/app9091952.
- [PMFZRCAG19] Alejandra Perez-Mena, José Alberto Fernández-Zepeda, Juan Pablo Rivera-Caicedo, and Himer Avila-George. PulAm: An App for Monitoring Crops. In Jezreel Mejia, Mirna Muñoz, Álvaro Rocha, Adriana Peña, and Marco Pérez-Cisneros, editors, *Trends and Applications in Software Engineering*, pages 196–205, Cham, 2019. Springer International Publishing. ISBN: 978-3-030-01170-3, DOI: 10.1007/978-3-030-01171-0\_18.
- [PMHT18] Walther Palomino, Giorgio Morales, Samuel Huamn, and Joel Telles. PETEFA: Geographic Information System for Precision Agriculture. In *2018 IEEE XXV International Conference on Electronics, Electrical Engineering and Computing (INTERCON)*, pages 1–4, Lima, Peru, 2018. IEEE. ISBN: 978-1-5386-5491-0, DOI: 10.1109/INTERCON.2018.8526414.
- [PO15] Andres Patrignani and Tyson E. Ochsner. Canopeo: A Powerful New Tool for Measuring Fractional Green Canopy Cover. *Agronomy Journal*, 107(6):2312–2320, 2015. ISSN:1435-0645, DOI: 10.2134/agronj15.0150.
- [PWL<sup>+</sup>19] Paul R. Petrie, Yeniu Wang, Scarlett Liu, Stanley Lam, Mark A. Whitty, and Mark A. Skewes. The accuracy and utility of a low cost thermal camera and smartphone-based system to assess grapevine water status. *Biosystems Engineering*, 179:126–139, 2019. ISSN: 1537-5110, DOI: 10.1016/j.biosystemseng.2019.01.002.
- [PWT<sup>+</sup>18] Dieisson Pivoto, Paulo Dabdab Waquil, Edson Talamini, Caroline Pauletto Spanhol Finocchio, Victor Francisco Dalla Corte, and Giana de Vargas Mores. Scientific development of smart farming technologies

- and their application in Brazil. *Information Processing in Agriculture*, 5(1):21–32, 2018. ISSN: 2214-3173, DOI: 10.1016/j.inpa.2017.12.002.
- [RFMM20] Amine Roukh, Fabrice Nolack Fote, Sidi Ahmed Mahmoudi, and Said Mahmoudi. WALLeSMART: Cloud Platform for Smart Farming. In *32nd International Conference on Scientific and Statistical Database Management*, SSDBM 2020, Vienna, Austria, 2020. Association for Computing Machinery. ISBN: 78-1-4503-8814-6, DOI: 10.1145/3400903.3401690.
- [RJK<sup>+</sup>17] Abdul Rehman, Luan Jingdong, Rafia Khatoon, Imran Hussain, and Muhammad Shahid Iqbal. Modern Agricultural Technology Adoption its Importance, Role and Usage for the Improvement of Agriculture. *Life Science Journal*, 14(2):70–74, 2017. DOI: 10.7537/marslsj140217.10.
- [RKdL<sup>+</sup>22] Davide Di Ruscio, Dimitris Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, and Manuel Wimmer. Low-code development and model-driven engineering: Two sides of the same coin? *Software and Systems Modeling*, 21:437–446, 2022. ISSN: 1619-1374, DOI: 10.1007/s10270-021-00970-2.
- [RMBCC23] Alexandra F. Rocha, Ivan E. Mallque, Samuel I. Bellido-Contreras, and Pedro S. Castaneda. Chakri: Mobile Application to Reduce Dependence on Intermediaries in the Marketing of Products in Family Farming. In *Proceedings of the 8th International Conference on Industrial and Business Engineering*, ICIBE’22, pages 19–25, Macau, China, 2023. Association for Computing Machinery. ISBN: 978-1-4503-9758-2, DOI: 10.1145/3568834.3568908.
- [RMMH<sup>+</sup>09] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch:

- Programming for All. *Communications of the ACM*, 52(11):60–67, nov 2009. ISSN: 0001-0782, DOI: 10.1145/1592761.1592779.
- [RSF17] M. R. Reisinger, J. Schrammel, and P. Frohlich. Visual languages for smart spaces: End-user programming between data-flow and form-filling. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 165–169, Los Alamitos, CA, USA, oct 2017. IEEE Computer Society. ISSN: 1943-6106, DOI: 10.1109/VLHCC.2017.8103464.
- [RSS<sup>+</sup>10] Trond Rafoss, Knut Sælid, Arild Sletten, Lars Fredrik Gyland, and Liv Engravslia. Open geospatial technology standards and their potential in plant pest risk management—GPS-enabled mobile phones utilising open geospatial technology standards Web Feature Service Transactions support the fighting of fire blight in Norway. *Computers and Electronics in Agriculture*, 74(2):336–340, 2010. ISSN: 0168-1699, DOI: 10.1016/j.compag.2010.08.006.
- [RSS<sup>+</sup>23] Kamil Rybiński, Michał Śmiałek, Agris Sostaks, Krzysztof Marek, Radosław Roszczyk, and Marek Wdowiak. Visual Low-Code Language for Orchestrating Large-Scale Distributed Computing. *Journal of Grid Computing*, 21, 2023. ISSN: 1572-9184, DOI: 10.1007/s10723-023-09666-x.
- [RYM<sup>+</sup>15] Minwoo Ryu, Jaeseok Yun, Ting Miao, Il Yeup Ahn, Sung Chan Choi, and Jaeho Kim. Design and implementation of a connected farm for smart farming system. In *2015 IEEE SENSORS - Proceedings*, pages 1–4, Busan, Korea (South), 2015. IEEE. ISBN: 978-1-4799-8202-8, DOI: 10.1109/ICSENS.2015.7370624.
- [SBK<sup>+</sup>21] Elsayed Said Mohamed, A. A. Belal, Sameh Kotb Abd-Elmabod, Mohammed A. El-Shirbeny, A. Gad, and Mohamed B. Zahran. Smart farming for improving agricultural management. *The Egyptian Journal of*

- Remote Sensing and Space Science*, 24(3, Part 2):971–981, 2021. ISSN: 1110-9823, DOI: 10.1016/j.ejrs.2021.08.007.
- [SDRIP23] Apurvanand Sahay, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Analyzing business process management capabilities of low-code development platforms. *Software: Practice and Experience*, 53(4):1036–1060, 2023. ISSN: 1097-024X, DOI: 10.1002/spe.3177.
- [SDRP20] Apurvanand Sahay, Davide Di Ruscio, and Alfonso Pierantonio. 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’20, Virtual Event, Canada, 2020. Association for Computing Machinery. ISBN: 978-1-4503-8135-2, DOI: 10.1145/3417990.3420197.
- [SGR22] Eashwar Sivakumar, Geetha Ganesan, and Ragavi. Harnessing I4.0 Technologies for Climate Smart Agriculture and Food Security. In *The 5th International Conference on Future Networks & Distributed Systems*, ICFNDS 2021, pages 504–510, New York, NY, USA, 2022. Association for Computing Machinery. ISBN: 978-1-4503-8734-7, DOI: 10.1145/3508072.3508175.
- [SLPF22] João Costa Seco, Hugo Lourenço, Joana Parreira, and Carla Ferreira. Nested OSTRICH: Hatching Compositions of Low-Code Templates. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, MODELS ’22, pages 210–220, Montreal, Quebec, Canada, 2022. Association for Computing Machinery. ISBN: 978-1-4503-9466-6, DOI: 10.1145/3550355.3552442.
- [SNN18] Peerasak Serikul, Nuttapun Nakpong, and Nitigan



- Nakjuatong. Smart Farm Monitoring via the Blynk IoT Platform. In *2018 Sixteenth International Conference on ICT and Knowledge Engineering*, pages 70–75, Bangkok, Thailand, 2018. IEEE. ISSN: 2157-099X, DOI: 10.1109/ICTKE.2018.8612441.
- [SPKB<sup>+</sup>22] Samuel J. Sutanto, Spyridon Paparrizos, Gordana Kranjac-Berisavljevic, Baba M. Jamaldeen, Abdulai K. Issahaku, Bizoola Z. Gandaa, Iwan Supit, and Erik van Slobbe. The Role of Soil Moisture Information in Developing Robust Climate Services for Smallholder Farmers: Evidence from Ghana. *Agronomy*, 12(2), 2022. ISSN: 2073-4395, DOI: 10.3390/agronomy12020541.
- [STCR13] M. I. Sudozai, Shamsuddin Tunio, Qamaruddin Chachar, and Inayatullah Rajpar. Seedling establishment and yield of maize under different seed priming periods and available soil moisture. *Sarhad Journal of Agriculture*, 29(4):515–527, 2013. ISSN: 1016-4383.
- [STM22] Vivek Sharma, Ashish Kumar Tripathi, and Himanshu Mittal. Technological revolutions in smart farming: Current trends, challenges & future directions. *Computers and Electronics in Agriculture*, 201:107217, 2022. ISSN: 0168-1699, DOI: 10.1016/j.compag.2022.107217.
- [VLA<sup>+</sup>16] G. Vellidis, V. Liakos, J. H. Andreis, C. D. Perry, W.M. Porter, E. M. Barnes, K. T. Morgan, C. Fraisse, and K. W. Migliaccio. Development and assessment of a smartphone application for irrigation scheduling in cotton. *Computers and Electronics in Agriculture*, 127:249–259, 2016. ISSN: 0168-1699, DOI: 10.1016/j.compag.2016.06.021.
- [VVANDM22] Louise Van Verre, Humberto Rodriguez Avila, Jens Nicolay, and Wolfgang De Meuter. FLOREnce: A Hybrid Logic-Functional Reactive Programming Language. REBLS 2022, pages 24–36, Auckland, New

- Zealand, 2022. Association for Computing Machinery. ISBN: 978-1-4503-9911-1, DOI: 10.1145/3563837.3568339.
- [Was19] Robert Waszkowski. Low-code platform for automating business processes in manufacturing. *IFAC-PapersOnLine*, 52(10):376–381, 2019. 13th IFAC Workshop on Intelligent Manufacturing Systems IMS 2019. ISSN: 2405-8963, DOI: 10.1016/j.ifacol.2019.10.060.
- [WFHB17] Achim Walter, Robert Finger, Robert Huber, and Nina Buchmann. Smart farming is key to developing sustainable agriculture. *PNAS*, 114(24):6148–6150, 2017.
- [WGVB17] Sjaak Wolfert, Lan Ge, Cor Verdouw, and Marc Jeroen Bogaardt. Big Data in Smart Farming - A review. *Agricultural Systems*, 153:69–80, 2017. ISSN: 0308-521X, DOI: 10.1016/j.agsy.2017.01.023.
- [WKW<sup>+</sup>18] Zhenglin Wang, Anand Koirala, Kerry Walsh, Nicholas Anderson, and Brijesh Verma. In Field Fruit Sizing Using A Smart Phone Application. *Sensors*, 18(10), 2018. ISSN: 1424-8220, DOI: 10.3390/s18103331.
- [WS15] S. Wyche and C. Steinfield. Why don’t farmers use cell phones to access market prices? Technology affordances and barriers to market information services adoption in rural Kenya. *Journal of Information Technology for Development*, 22(2):320–333, 2015. ISSN: 1554-0170, DOI: 10.1080/02681102.2015.1048184.
- [WS18] Theerayod Wiangtong and Phaophak Sirisuk. IoT-based Versatile Platform for Precision Farming. In *2018 18th International Symposium on Communications and Information Technologies (ISCIT)*, pages 438–441, Bangkok, Thailand, 2018. IEEE. ISBN: 978-1-5386-8458-0, DOI: 10.1109/ISCIT.2018.8587989.

- [YCLF13] Jiuyan Ye, Bin Chen, Qingfeng Liu, and Yu Fang. A precision agriculture management system based on Internet of Things and WebGIS. In *International Conference on Geoinformatics*, pages 1–5, Kaifeng, China, 2013. IEEE. ISSN: 2161-024X, DOI: 10.1109/Geoinformatics.2013.6626173.
- [YS12] Intaravanne Yuttana and Sumriddetchkajorn Sarun. BaiKhao (rice leaf) app: a mobile device-based application in analyzing the color level of the rice leaf for nitrogen estimation. In Tsutomu Shimura, Guangyu Xu, Linmi Tao, and Jesse Zheng, editors, *Optoelectronic Imaging and Multimedia Technology II*, volume 8558, pages 96–102. International Society for Optics and Photonics, SPIE, 2012. DOI: 10.1117/12.2001120.
- [YST<sup>+</sup>17] Qiangyi Yu, Yun Shi, Huajun Tang, Peng Yang, Ankun Xie, Bin Liu, and Wenbin Wu. eFarm: A Tool for Better Observing Agricultural Land Systems. *Sensors*, 17(3), 2017. ISSN: 1424-8220, DOI: 10.3390/s17030453.
- [Zam18] Jesse Zaman. *DISCOPAR: A Visual Reactive Flow-Based Domain-Specific Language for Constructing Participatory Sensing Platforms*. PhD thesis, Vrije Universiteit Brussel, 2018. ISBN: 978-9-49231-289-1.
- [ZKD18] Jesse Zaman, Kennedy Kambona, and Wolfgang De Meuter. DISCOPAR: A visual reactive programming language for generating cloud-based participatory sensing platforms. In *REBLs 2018 - Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, Co-located with SPLASH 2018*, REBLs 2018, pages 31–40, Boston, MA, USA, 2018. ISBN: 978-1-4503-6070-8, DOI: 10.1145/3281278.3281285.
- [ZKD21] Jesse Zaman, Kennedy Kambona, and Wolfgang De Meuter. A reusable & reconfigurable Citizen Observatory platform. *Future Generation Computer Systems*,

114:195–208, 2021. ISSN: 0167-739X, DOI: 10.1016/j.future.2020.07.028.

- [ZPP<sup>+</sup>20] Ziyuan Zhang, Zhihua Pan, Feifei Pan, Jun Zhang, Guolin Han, Na Huang, Jialin Wang, Yuying Pan, Zizhong Wang, and Ruiqi Peng. The Change Characteristics and Interactions of Soil Moisture and Temperature in the Farmland in Wuchuan County, Inner Mongolia, China. *Atmosphere*, 11(5), 2020. ISSN: 2073-4433, DOI: 10.3390/atmos11050503.