A VISUAL PROGRAMMING TOOL FOR WORKFLOW CUSTOMIZATION IN COCO-PEAT MANUFACTURING

Dehipola H. M. S. N

IT21291678

BSc (Hons) degree in Information Technology

Specializing in Software Engineering

Department of Computer Science and Software Engineering

Sri Lanka Institute of Information Technology

Sri Lanka

April 2025

A VISUAL PROGRAMMING TOOL FOR WORKFLOW CUSTOMIZATION IN COCO-PEAT MANUFACTURING

Dehipola H. M. S. N

IT21291678

Dissertation submitted in partial fulfillment of the requirements for the Bachelor of Science (Hons) in Information Technology Specializing in Software Engineering

Department of Computer Science and Software Engineering

Sri Lanka Institute of Information Technology

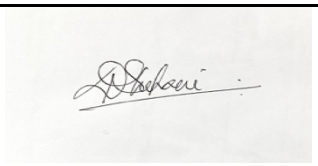
Sri Lanka

April 2025

DECLARATION

I declare that this is my own work, and this proposal does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any other university or Institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to Sri Lanka Institute of Information Technology, the nonexclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

Name	Student ID	Signature
Dehipola H. M. S. N	IT21291678	

The above candidates are carrying out research for the undergraduate Dissertation under my supervision.

Supervisor: Mr.Vishan Jayasinghearachchi	Signature:	Date: 2025/04/10
Co-Supervisor: Mr. Jeewaka Perera	Signature:	Date:2025/04/10

ABSTRACT

This research presents a domain-specific, visual workflow customization tool designed for the coco-peat manufacturing industry, where frequent workflow modifications are essential but often limited by the technical barriers of existing systems. Medium-scale manufacturers and exporters, who typically lack programming expertise, require a system that enables them to define and manage process workflows independently. To address this, the proposed tool introduces a drag-and-drop interface and a Domain-Specific Language (DSL) editor, both of which support real-time validation and plugin-level customization. Built using a modular architecture with React, Node.js, MongoDB, and gRPC-based communication to an external core system, the solution empowers non-technical users to create, version, and assign workflows with minimal friction. The tool enforces domain-specific validation rules, ensures logical correctness through a stepwise validation engine, and enables the creation of reusable plugins. Overall, the system bridges the gap between domain knowledge and technical execution, supporting flexible and user-driven automation in manufacturing environments.

Keywords: Workflow customization, Domain-Specific Language, Visual programming, Coco-peat industry, non-technical users, Domain-specific, Validation engine, Plugin creation, Manufacturing automation

ACKNOWLEDGEMENT

First and foremost, I would like to express my heartfelt gratitude to my supervisor, **Mr. Vishan Jayasinghearachchi**, for his invaluable guidance, continuous encouragement, and unwavering support throughout the duration of this undergraduate research project. His expertise and dedication were instrumental in shaping the direction and success of this study.

I am also sincerely thankful to my co-supervisor, **Mr. Jeewaka Perera**, for his constructive feedback, technical insights, and constant readiness to provide assistance. His contributions significantly enhanced both the quality and scope of the research.

This project was completed as a team effort, and I extend my deepest appreciation to my team members for their hard work, collaboration, and consistent dedication. Their efforts were essential in overcoming challenges and achieving our shared goals.

Additionally, this research involved exploring areas beyond our initial expertise, and I am grateful to those—both individuals and external resources—who supported us in bridging those knowledge gaps and expanding our understanding across disciplines.

Lastly, I would like to extend my heartfelt thanks to my family, friends, and colleagues for their continuous support, motivation, and understanding. Their encouragement gave me the strength and focus to see this journey through to completion

Table of Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Description
DSL	Domain-Specific Language
GUI	Graphical User Interface
UI	User Interface
EC	Electrical Conductivity
IoT	Internet of Things
gRPC	gRPC Remote Procedure Call
REST	Representational State Transfer
API	Application Programming Interface
UI/ UX	User Interface/ User Experience
JWT	JSON Web Token
JSON	JavaScript Object Notation

1. INTRODUCTION

1.1 General Introduction

In the rapidly changing digital landscape of today, industries are using automation, intelligent systems, and workflow management technologies more and more to increase responsiveness, productivity, and traceability. To keep their competitive edge and adapt to the increasing complexity of managing operations, many industries—from manufacturing to logistics to agriculture—are moving from manual procedures to technologies. Workflow management has become a focal area in this transformation, helping organizations to architect, deploy, and monitor their workflow tasks in a structured and efficient manner.

Workflow management systems are often designed in such a way that they are either generalized and too complicated to operate easily, or they require a knowledgeable technical person to operate and customize. Consequently, many domain-specific industries, especially those involving manual and semi-manual processes, cannot generate workflow tools that fit their needs, skills, and adaptability. This is particularly seen in medium-scale industry, where a more frequent need to adapt workflows to changing conditions does not meet flexibility, usability, or contextual fit with the available workflow tools.

The coco-peat industry serves as one example of an expanding area towards sustainable agricultural practices, which calls for a customizable process flows that are built with quality standards, environmental data and export requirements. In these environments, workflows often need to be created and adjusted by non-technical users, such as exporters and manufacturers. Existing approaches do not support these users and often create a divide between domain knowledge and technical execution. This absence of workflows with visual cues, aligned toward domain knowledge, to allow domain experts to manage the operational logic autonomously reinforces the importance of flexibility, usability, and user autonomy.

This research project addresses the gap by providing a customizable tool that provides a visual workflow management platform for coco-peat manufacturers. The platform aims to bridge the divide between domain knowledge and digital workflow management through intuitive user interfacing, real-time validation and feature sets designed for both exporters and manufacturers. This will allow domain experts to create, modify, and manage workflows, all without programming expertise or knowledge besides an understanding of their workflow.

1.2 Background Literature

Customizing workflow has become a crucial component of contemporary industrial software systems. Traditional workflow management systems are frequently less flexible and easy for non-technical users to utilize due to the increasing complexity of domain-specific needs. Current developments in drag-and-drop-based interaction models, visual programming tools, DSLs, GUIs, and workflow management systems are discussed in this overview of the literature. The objective is to point out areas where current research and tools have limitations and to support the necessity of a visual workflow customization tool built especially for the coco-peat production domain.

1.2.1 Workflow Management System

Workflow management entails the coordination, assessing, and intermingling of complex procedures through a specified sequence of tasks, and relates to the integration of workflows and the automation of items to improve efficiency, as we saw in the literature earlier [1]. This literature covered the integration of modern workflow management systems into existing environments. Modern systems allow businesses to automate routine processes and improve workflows, therefore it is imperative to document the supports necessary for effective workflow management, which cover process modeling, execution environments and monitoring.

In another study based on an acute general surgical service, practical advances had taken place based on mobile-based workflow solutions. Mobile task management combined the ability to engage their mobile device to receive real-time updates and assigning tasks, thus the mobile solution positively and significantly enhanced the effectiveness and efficiency of clinical workflows. The ability to integrate mobile mediums increased workflow so well that it could be seen that workflows improved clinically as errors declined, increased staff communication and engagement in the management of patients improved [2].

Another relevant study presented ColorTree, a batch configuration tool created expressly for phylogenetic trees. ColorTree demonstrates a utility of domain-specific tooling for use in workflow management systems to support batch actions that simplify and speed the customization of workflows for domain-specific applications. ColorTree provides a clear and responsive user interface for a comprehensive view of the data for handling a significant data set and data visualization experience that greatly reduces user effort and increases accuracy in data-driven decisions in domain-specific scientific workflows [3].

1.2.2 Visual Programming and Drag-and-Drop Interfaces

Visual programming environments (VPEs) allows users to develop system logic visually, reducing the need for text-based coding and allowing non-technical users to explore programming. RoboStudio is an

example of a visual programming environment developed to support educational robotics where users will develop robot behaviors through simple visual blocks. This not only helps to shorten the learning curve associated when programming a robot but also enhances educational engagement and effectiveness when implementing a robotic programming course [4].

SimStack is another useful illustration of a visual framework, which allows scientists to design simulation workflows using a simple drag-and-drop interface. SimStack eliminates the need for programming and allows users to quickly construct and revise advanced simulation pipelines, making the workflow development process far more efficient and productive for the research and development community [5].

Direct annotation using drag-and-drop methods for labeling images was explored for potential usability and efficiency improvements. The study demonstrated large increases in user productivity and accuracy through drag-and-drop actions, underscoring the value of visual interaction alternatives as an improved user experience and efficiency approach [6].

Another example of a graphical programming environment developed to create simulation scenarios reinforces the benefits of visual interactions quickly and intuitively for various grouping, sequencing, and organizing of simulation components. With visual interactions, users can easily assemble and modify components to create explorations and to address relevant barriers with greater usability and accessibility, especially for non-technical users, developed to manage complex simulation tasks [7].

1.2.3 Domain-Specific Languages

DSLs are modeling or programming languages developed specifically for a certain domain. They result in greater expressiveness and ease of use in that domain, as compared to general-purpose languages. During the study, a domain-specific language called CSX was found. This is a constraint-based DSL for managing the configurations of industrial complex printing systems [8]. This system employs constraint solvers to automate the exploration of configuration space, great reducing configuration, and productivity burdens in an industrial setting.

In a separate study presented the capacity to create DSLs targeted the field of digital printing systems. The research argues for DSL effectiveness in contingent complicated system constructs while making configuration easier and improving the maintainability of systems as well some clear evidence on the effectiveness of DSLs in particular settings industrial domains [9].

Another research related to using model-based testing and code generation within the domain of the Oil Industry Language (OIL), demonstrated DSL capabilities based on automated complex, domain-

specific tasks that create smooth workflows and shorter development times based on automated testing and code generation methods that work effectively in context [10].

The study found out the impact of modular architectures on performance with Incremental SGLR parsing algorithms, which boosts an efficient parsing mechanism that is very important for DSL development and real-time application. Efficiently parsing the description generated by the DSL contributes significantly to the practicability of DSLs, especially in conditions that require fast or precise responses to configuration [11].

A study proposed automated methods to generate DSL APIs from domain-specific metamodels, proposing an approach that abstracts the complexity of implementation. The proposal significantly simplifies the integration of DSL translators into an application, opening the possibility for effective DSL adoption and integration into domain-specific applications [12].

In the domain of socially assistive robotics, HealthBot is a system that introduced an XML-based DSL to specify robot behaviors using finite state machines. This DSL enabled domain experts without programming experience to configure robots to take specific actions and to respond to the actions of elderly patients in situations of social care. The proposed structure provided flexibility that facilitated both rapid prototyping and behavior module reuse, which closely aligned with the principles of modular, visual customization found in modern DSL tools [13]. In addition, the functionality to use graphical editors for state transitions added an intuitive layer of abstraction to the behavior specification, making it ready to be configured as part of a visual programming environment targeting non-programmers.

1.2.4 Graphical User Interfaces for Workflow Tools

The role of GUIs is critical for easing user interactions within complex workflow systems. Another study presented a scriptable graphical user interface engine for embedded platforms, which is both a flexible and lightweight solution for resource-limited systems. The GUI allows for configuration in a straightforward and effective way, including modifiable user-driven elements to extend the usefulness of the system across multiple embedded domains [14].

The paper [15] performed a comparative study of text-based interfaces versus GUIs in the context of nursing orders. The research showed clear benefits associated with GUIs, including decreased response time, minimum user errors, and an increase in total user satisfaction with the work-related tasks. This research highlights the importance of intuitive GUI designs in workflow management to allow for increased user performance and accuracy in work-related tasks in.

A universal web interfaces was successfully created for robot control applications that utilized a

modular type of GUI that adapts to different hardware configurations. Additionally, this highlights the demand for adaptable and flexible GUI designs for different workflow situations, which allows users to interact in a more efficient way and reduces the adaptation time for learning other systems [16].

1.2.5 Human-Centric Interaction and Usability

User-centered types of interaction focused on usability are critical to increasing user acceptance and productivity in workflow systems. Research indicates that customizing interaction styles to context and user requirements can improve usability and users' perception of usability by facilitating dynamic adaptation of the interface in different operational environments. Adaptive interface design can also reduce cognitive load and ease interaction in cognitively demanding tasks to enhance more efficient workflows to the [17].

1.3 Research Gap

Workflow management systems, visual programming environments, DSLs, and GUIs have advanced significantly, but they still have significant constraints when it comes to applying these technologies to actual, domain-specific manufacturing settings like the production of coco-peat. Existing workflow management tools have typically been built from a general purpose, which does not allow for specificity in unique functional procedures, frequent changes, or role-based operations in the unique coco-peat supply chain. Most applications do not allow for the unique needs of the coco-peat domain that include grading, cutting, drying, and washing processes, which require tailored workflows that vary between manufacturers and exporters.

Many of the workflow applications available today, do not provide the required flexibility and customizability across the workflow for coco-peat production. They generally conform to very rigid workflows that rely on pre-existing templates that cannot easily be modified, replaced, or built upon to fit newly determined operational needs at the code level. Therefore, workflow is typically inefficient or obsolete in scenarios in which frequent change seems necessary. Conversely, coco-peat production environments, producers, and exporters must create and modify workflows repeatedly for customer needs or contingencies of machinery and/or sensors (e.g., moisture levels, EC levels). Flexibility and customizability cannot be achieved quickly or efficiently without such a tool.

Another major concern is the reliance on IT professionals or software developers to implement modifications. The current tools usually involve the user's interaction with backend logic or configuring the system in languages or scripts. This becomes particularly difficult for small and medium-sized enterprises (SMEs) in the coco-peat industry, who will not have access to technical persons. Factory supervisors and field experts, who are both domain experts, usually do not actively participate in

configuring a system due to the technicality introduced by systems tools.

Additionally, while some platforms provide a visual programming capability, these tools are not meant to interpret user requirements in the context of a domain (i.e., coco-peat) or translate the requirements to logic to execute. Most visual editors have support for process modeling, but almost none specifically support domain-specific constraints, validations or runtime behaviors relevant in coco-peat production. Also, similar to the gap in integration with sensors, existing visual tools do not interface with IoT sensors or support data-driven decision-making that requires real-time execution of the workflow environment, all of which is critical for modern agricultural production and processing.

The absence of domain-specific DSLs and user-friendly graphical interfaces is another clear drawback. Despite their ability to encapsulate domain information technically and practically, DSLs still have access issues. Most DSL implementations are made with a technical audience focus and provide minimal assistance for modification or user-friendly interaction. In the lack of a graphical user interface (GUI) or interactive environment, DSLs are difficult for non-technical individuals to understand and use effectively.

In summary, current workflow tools:

- Are not designed for coco-peat manufacturing needs.
- Depend on software developers or IT professionals.
- Are not easily adaptable to frequent changes in workflow.
- Are not usable by non-technical users.
- Do not support integration with IoT data and backend execution logics.

Thus, this research outlines a critical and urgent gap in the availability of a domain-specific visually driven programmable workflow customization tool for the coco-peat space that is designed to afford non-technical users the ability to customize their own workflows via intuitive drag-and-drop GUI interfaces. The flow could be customized with DSL-based customization logic and engagement mechanisms that are responsive to IoT mechanisms as well. By providing domain experts the ability to manage their own workflows, increases efficiency and facilitates innovation and engagement at every level of the manufacturing process.

1.4 Research Problem

The demand for accuracy, efficiency, and flexibility in agricultural manufacturing industries such as coco-peat production highlights the need for technology-fit specific solutions that allow end-users to build their solutions. Coco-peat manufacturing has many steps—grading, cutting, washing, drying and

packing—that often need to be changed often based on client specifications, environmental conditions and machine differences. These changes are difficult to do with the current workflow management systems available.

Most current workflow management systems are either too generic for broader ERP use or technical and require programming or technical knowledge to set up configurations. They lack the flexibility to provide domain-specific scenarios, and they are often not flexible enough to allow frequent changes to setup workflows. To complicate things, they are designed for IT professionals or more technical workers, which widens the discovery of the gap between what end-users know and the existing tools being developed.

In the coco-peat industry, exporters often connect with buyers directly and relay certain needs, requests, preferences, and so forth, to their manufacturers. This consequently results in a dynamic, fast-changing workflow environment. For instance, one order may require double washing and an exact level of electrical conductivity (EC) is adhered to, while another order may require preference to be given to a certain drying time. The need for these changes, unless there are customizable, user-centered tools, cannot happen quickly and will lead to delays, miscommunications, or inefficient production.

Currently, those change requirements require developers or IT experts to coordinate with, as they understand how the backend workflow engines work, and can adjust the logic as needs require. This dependency creates bottlenecks, reduces responsiveness, and adds unnecessary complications to the daily operations. Non-technical staff and stakeholders, such as the exporters and manufacturers themselves, are not able to be directly involved in the design and control of workflows even though they are very knowledgeable of this domain and process, which would improve speed and efficiency.

Though there are visual programming environments, they frequently lack the plugin-based architecture or domain-specific logic required for real-time execution. Similarly, due to their textual and abstract syntax, DSLs are yet inaccessible to non-programmers, even if they can encapsulate expert knowledge in a programmable format. These limitations highlight the gap between current systems' usability and configurability. The lack of a visual, drag-and-drop workflow customizer specific to the coco-peat domain that is usable by a non-technical audience is an important area of research. This customization would have to integrate the graphical interface workflow, domain-specific logic using a DSL, a plugin extensible architecture, and integration with real-time IoT data and execution engines.

The research question guiding this analysis is:

How can a domain-specific, visual workflow customization tool be developed to allow non-technical users (exporters and manufacturers) to modify and manage coco-peat manufacturing

workflows without technical expertise?

Solving this research problem is important not only to improve manufacturers' and exporters' operational efficiencies and autonomy, but also contributes to creating a scalable model for user-centric workflow customization for other agricultural and industrial domains. The solution must combine sufficient technical inputs to provide valid performance, user-friendliness for non-technical users to manage complex backend processes visually using simple and intuitive front-end interfaces. The proposed research will address and fulfill the gap by providing a system that combines visual customization using a modular architecture with executable domain-specific configurations in coco-peat supply chains.

1.5 Research Objectives

1.5.1 General Objective

The primary objective of the research is to develop a visual programming workflow customization tool that is domain-specific and tailored for non-technical users in the coco-peat manufacturing sector, including manufacturers and exporters. The tool's goal is to provide a simple graphical user interface that enables workflow design, modification, and management without the need for expertise in programming. The proposed tool will integrate a domain-specific logic through a backend-compatible DSL and will be able to provide real-time feedback through IoT data. Overall, the solution seeks to create even more operational autonomy, possibly less dependence on developers, and improved responsiveness to production processes.

1.5.2 Specific Objectives

1. **To identify which domain-specific workflow elements are involved in the production of coco-peat (e.g., grading, washing, drying):**
   A crucial aspect of this research is gaining an extensive understanding of the operational and functional framework of the coco-peat production process. In particular, there are several procedures that need to be performed in order to turn raw coconut husks into goods made from coco-peat. These procedures typically involve filtering the husks based on their quality, washing them with the proper EC levels, drying them under regulated settings, and packing the final product, though there may be variations in how these steps are carried out. Each step has its own requirement, with operation constraints and optionality based on buyers' orders. Based on identifying and modeling these parts in the coco-peat manufacturing process, a visual tool can provide a range of configurable modules as a pallet of building blocking specifically for their domain. This also contributes to greater

modularity and extensibility of the design.

2. **To analyze current DSL-based systems and workflow customization tools to determine their limitations in supporting non-technical users:**
   An important part of the research is examining and assessing tools that currently exist in the space of visual workflow management and DSL frameworks across domains. This involves comparing a combination of both open-source and proprietary platforms providing workflow modeling, visual editors, and DSL interpreters. The analysis will focus on user interface (UI) design, learning curves, capabilities for customization, and the suitability to be deployed by non-programmers. Most tools reviewed either lack the specificity to support a particular domain, or include interface complexity that makes them unsuitable for non-technical groups across the stakeholder environment. Ultimately, this phase of research is to illustrate the limitations of existing tools in usability, flexibility, and accessibility, and provide a basis of exploration for a new, more inclusive, user-centered tool.

3. **To design an interface for visual workflow customization that enables drag-and-drop workflow step configuration:**
   When considering the identified domain-specific components and the limitations imposed by the existing system, a new user interface will be created that includes a drag-and-drop interface. Users will be able to visually select, arrange, and configure workflow steps on a canvas so they can rapidly and efficiently build manufacturing processes. Each element of functionality will represent either a plugin or process component (e.g. Wash, Dry, Grade) and will have attributes that can be modified through a simple set of form input. Visual components like icons, colors, and links will also be integrated into the user interface to show dependencies or sequence flow. The final product will be a functional and aesthetically pleasing design that caters to the requirements of both non-technical users as well as professionals in the coco-peat manufacturing industry.
4. **To be able to integrate a DSL capable of translating visual setups into logic that can be executed:**
   The visual configurations designed by users need to be transformed into machine-readable logic to be executed in the backend. This goal is to create or adapt a DSL to encode this workflow based on user actions in the graphical user interface. Each block placed on the canvas will relate to a command or structure in the DSL, and the system will compile or serialize the collective visual representation into a DSL script. The DSL should be expressive and limited to actions in the domain, including safety and ease of use. It is intended to be a layer between the user input and the system enforcement, thereby enabling automation without exposing users directly to technical aspects of the syntax.

5. **To enable real-time customization and response with production-related IoT sensor data (e.g., moisture, EC level):**

   Coco-peat production is rooted heavily in physical parameters, specifically electrical conductivity (EC), moisture content, and time . This described goal is centered on the integration of sensor data into the workflow tool in order for the users to be able to make data-driven decisions. For example, the user could set conditional steps such as "repeat wash if EC > 1.0" or "wait until moisture < 20% before moving to drying." This point allows the workflows to be dynamically adapted while a process is executing. The interface would display sensor data and provide the user the ability to create logic for basic rules that require nothing to be coded yet allow for responsiveness in real-time scenarios.

6. **To conduct user testing with subject matter experts to assess the utility and effectiveness of the developed tool:**

   Following development, the tool must be verified and assessed by actual participants in the coco-peat supply chain, such as manufacturers and exporters. The goal is to determine whether the tool fulfills its design goals of usability, accessibility, efficiency, and accuracy. Usability testing would be scheduled, where users would be asked to create and modify workflows based on prescribed scenarios. Feedback would be taken through observation, interviews, and questionnaires, leading to practical findings about time to complete tasks, error rate, learning curve, and satisfaction. The results would be used to revise the tool while validating its future application in real-world.

2. METHODOLOGY

2.1 Methodology

The approach to developing the proposed visual workflow customization tool adheres to a Design Science Research Methodology (DSRM) because DSRM is especially suited to address technology-based innovations that seek to tackle real-world problems through the design and evaluation of functional artifacts. This methodology allows for a structured and methodical approach to problem identification, artifact design, development, demonstration, evaluation, and communication of results that suits the applied nature of this research.

According to the context of this research project, the primary problem is that non-technical users in the coco-peat manufacturing industry, like exporters and manufacturers, are not able to design, customize, and manage operational workflows, or any kind of workflow, without relying on software developers. Existing workflow management systems either provide generic workflows or require some level of domain knowledge of the programming language and, as a result, are not accessible by these non-technical stakeholders. The artifact developed through this research is assisting with this challenge by providing a domain-specific visual drag-and-drop interface with an embedded DSL engine so that non-technical users can design and manage workflows independently.

The approach uses a user-centred, iterative development cycle to guarantee continuous validation and requirement alignment. Several key phases are included in this iterative process:

- **Requirement analysis,** in which functional dependencies and domain-specific steps are identified through analysing workflows from real-world coco-peat production environments.
- **Interface design,** where prototypes are created using visual programming structures to enable intuitive user interaction.
- **DSL integration and plugin logic mapping** involves mapping GUI configurations to backend-executable logic using a unique DSL.
- **Validation framework development** ensures that every textual or visual input is correct in both syntax and semantics before it is executed.
- **Real-time sensor integration** allows context-aware workflow execution by integrating IoT data into workflow conditions.

During these stages, the research emphasizes the challenges and needs of users who are not programmers. ensuring that the software is enabled via a visual or form user interface and all aspects of the software—plugin development, workflows, and real-time monitoring—are easily accessible.

Furthermore, it is a modular system with separation between the frontend user interfaces (UI) for visualisation of the system, the backend logic processor, the plugin registration module, the DSL compiler and the core execution system. This separation of concerns can improve maintainability, scalability, and extensibility to other domains beyond coco-peat manufacture.

Lastly, we have emphasised evaluative feedback from practitioner domain experts (exporters and manufacturers) to validate usability and functional performance. The user testing and feedback aspect of the methodology provides the opportunity to adjust and adapt the system to real-world production limits, user input behaviour and expectations.

2.1.1 System Architecture

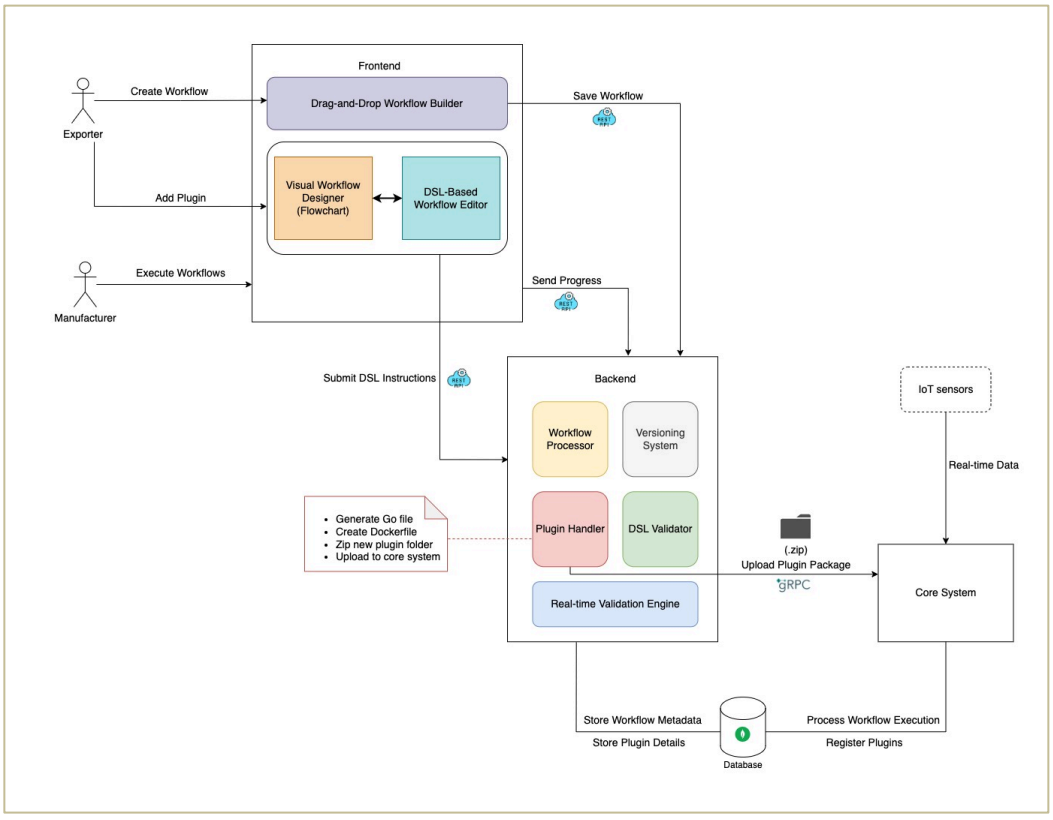


Figure 1: High-Level Architecture of the Workflow Customization System

The high-level architecture represents the design of the workflow customization tool with a focus on the interaction between the Frontend and Backend layers that composed a module of this tool, as well as their interactions with an external Core System. Additionally, it describes interactions between the exporters and manufacturers and the overall system, as well as how the internal modules interact with each other to support the workflow creation module, the plugin customization module, the validation of the workflows, and communication with the Core System for plugin initiation and real time workflow monitoring.

The exporter has two responsibilities: either build new workflows or customize existing workflows in the frontend interface using the Drag-and-Drop Workflow Builder. The Drag-and-Drop Workflow Builder encompasses two supporting tools: the Visual Workflow Designer, which enables the user to define steps via flowchart elements in a visual representation, and the DSL-Based Workflow Editor, which serves as a tool for defining and editing domain-specific syntax. Together these two editors support two-way conversion between visual flowcharts and DSL-based syntax. The exporter can define workflows by adding steps (plugins), setting up parameters, and checking structure without needing any programming knowledge.

Once a workflow is configured, the exporter can save the workflow configuration to the backend via REST APIs. The backend includes various functional modules, such as the Workflow Processor that is responsible for parsing and building workflow logic, the Versioning System which handles the historical states of workflows, and also the DSL Validator that is responsible for checking structure and syntax errors. In addition, there is a Real-Time Validation Engine that provides continuous feedback while workflows and plugins process through the creation, which helps by displaying errors or validation outputs in real-time to support the user through the identification of and modifications of workflow and plugin errors before saving or assigning.

When exporters would like to add new plugins (i.e., custom steps) through the Plugin Handler Module, the plugin data such as the name, the applicable IoT sensor, and the instructions in a flow chart are all sent to the module. That module is responsible for automating the process of creating the plugin on the backend which consists of the following steps:

- Generating a Go source code file that represents the plugin logic.
- Creating a Docker file that describes the plugin execution environment.
- Saving both files to a folder identified by the plugin name.
- Zipping the folder and sending the zipped directory to Core System via gRPC communication.

Then the Core System unzips the plugin package, registers it as a functional step in the workflows, and

makes it available for reuse in future workflows. The Core System connects to the IoT sensors (e.g., EC or moisture sensors) and obtains information in real-time which can then be used to define conditional logic for the Workflow runner (e.g., "Repeat wash if EC > 1.5").

On the Manufacturer side, the workflows assigned by exporters are visible. These workflows contain ordered plugins, and sub-steps. When the manufacturer goes through each step, the times and completion progress are sent back to the backend through REST APIs. Each step is logged and marked the status as "Not started", "In Progress", or "Completed". The exporter has visibility over each log as it occurs for monitored or traceable purposes.

- The system also includes a centralized database that contains the following information:
- Workflow metadata and versioning histories.
- The definitions and registration details of the plugins.
- The execution logs from manufacturers. This information is maintained in persistent storage for monitoring and traceability and to allow revisiting previous workflows and comparing creativity through executions of the process flow.

Overall, the architecture ensures seamless communication between users and a complex backend environment. This uses visual programming, DSL processing, plugin automation, and real-time execution for a cohesive tool for the needs of the coco-peat industry.

2.1.2 Workflow Creation Lifecycle

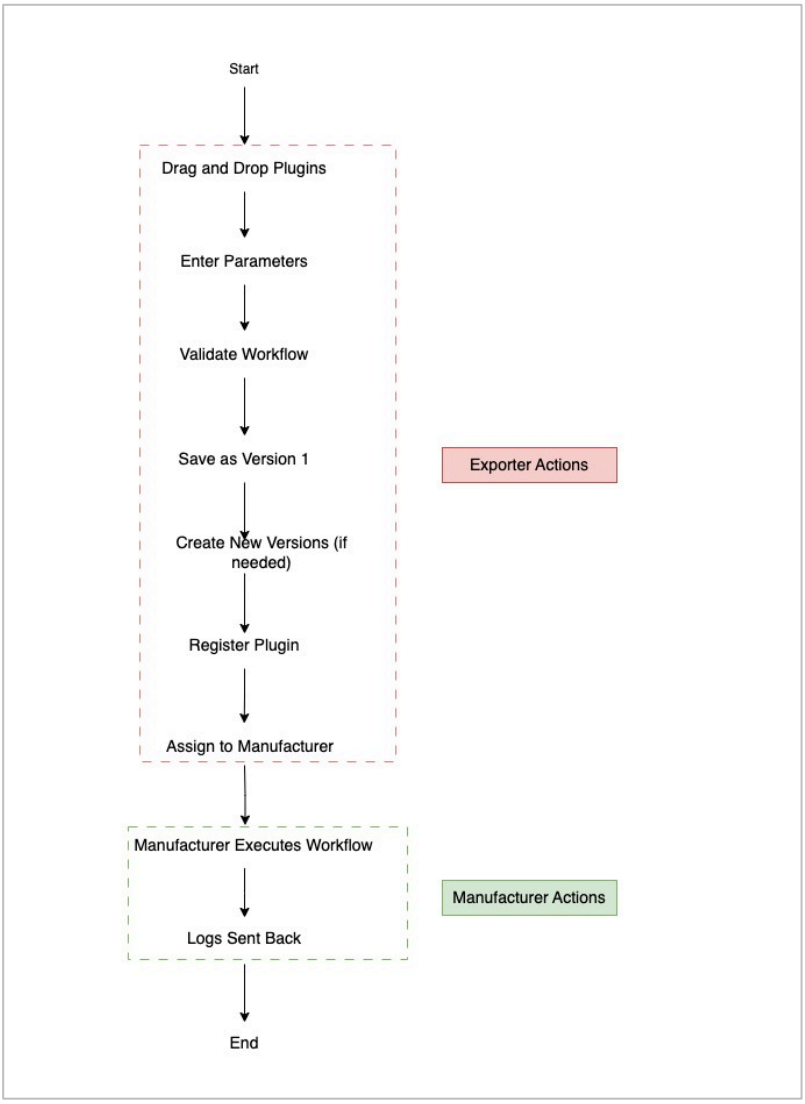


Figure 2: Workflow Creation and Execution Lifecycle

The process of creating the workflow lies at the core of the proposed system. This is a phase where exporters can model the domain-specific operational steps used in the coco-peat production pipeline. The production lifecycle was designed specifically for non-technical users, simplifying the programming process without sacrificing operational flexibility in the final product and offering these capabilities through a drag-and-drop visual interface and built-in validation capabilities.

2.1.2.1 Initiating Workflow and Interface

The first action of exporters is to access the workflow, featuring a drag-and-drop capability, from the front-end interface. Upon entry, the user is presented with two main sections:

- The sidebar is a listing of available plug-ins, i.e., pre-defined steps such as "Grading", "Washing", and "Drying".
- Canvas is the workflow design space.

Exporters can drag (click-and-drag) from the sidebar and drop those into the canvas to create a stepwise production flow. Each plugin should be dropped into the canvas to indicate the desired sequence of operation as needed in the real-world coco-peat production processes.

2.1.2.2 Adjusting Plugin Settings

Once the user has ordered their plugins as the requirement, the user will configure the plugin settings (or other similar steps). Each plugin has one or more configuration fields based on each plugin's functionality. For example:

- A washing plugin may need threshold values for EC and the time.
- A grading plugin may have some textural or colour ranges.

All of these configurations are done through in-place form fields when configuring a step. The system will ensure that all required fields will be marked accordingly to guide users in an accessible format. Proper documentation is offered for ease of use, and domain experts will understand the consequences of their configurations even without a technical background.

2.1.2.3 Real-time Validation During Workflow Construction

The Real-Time Validation Engine is triggered when the user is building the workflow. The validation checks that are carried out by this engine include:

- Syntax checks look for stages that are broken or incorrect, which could lead to a syntax problem.
- Dependency checks: Verifying that the procedures are carried out in the correct order (e.g., drying cannot be done before washing).

- Completeness of parameters - Making sure required field is not left blank.
- Semantic check - Such as, checking for duplicate steps, circular dependencies, or missing a start and/or end.

All of these validations occur in a terminal panel at the bottom of the workflow canvas, with errors happening in real time as the user makes changes. Warnings and successes are also shown with colors that help guide the user through the construction process seamlessly.

This continuous feedback loop significantly reduces the chances of deploying a workflow that is incomplete or illogical while also giving the user confidence while constructing the logic.

2.1.2.4 Building and Versioning

After the exporter has approved the design and configuration of the workflow, he is supposed to click the designated button to initiate the build process. While the build process is in progress: The system performs one final validation pass.

- The system will perform final validation pass.
- If the validation is successful, then the system will generate a Workflow ID and save the workflow with Version 1 in "Draft" status.

Exporters will have access to a summary page where they can see a visual of the workflow. The user can see the full step sequence and all parameters. The exporter can return to the builder and make changes if further adjustments are needed. For every change followed by a successful build, a new version (i.e., Version 2) comes into existence under the same Workflow ID. This version system provides for iterative improvement, experimentation, or full confirmation before deployment.

There is an essential registration step after the exporter completes a workflow version for execution. The system requires the exporter to register each plugin referenced in the workflow in the external Core System before the workflow gets assigned to a manufacturer. This allows the core to know that this workflow intends to use particular, pre-existing plugins rather than requiring the uploading of plugin code.

This ensures that the core can clearly recognize whether it can correctly identify and perform the plugin logic needed at runtime. The exporter can only proceed with the activity after the plugins have been registered for a specific workflow version. At that time, a manufacturer can be assigned for the finalized version of the workflow, and it would move from "Draft" to "Pending" task, meaning that it is waiting to be executed.

From declaring plugins to assigning them and validating and versioning workflows, this sequenced process facilitates a consistent, transparent, and ready state at all sequencing of workflow development and execution with the system.

2.1.3 Plugin Creation Lifecycle

Although the system comes with a number of predefined plugins to address standard procedures in the coco-peat production such as Grading, Washing, and Drying, it also gives exporters the ability to create their own plugins. This capability is key to scalability of the organization and for the user to enlarge the system to meet either new production steps that arise, or variations in workflow that are specific to the customer in question.

The plugin creation lifecycle allows non-technical users to define, structure and deploy new executable steps through a guided visual or textual configuration process, ultimately integrating those steps, however they are defined, into the larger workflow creation system.

2.1.3.1 Initiating a New Plugin

To start, the exporter accesses the "Add New Plugin" section of the interface. The workflow starts with the entry of basic metadata:

- Plugin Name - This allows the exporter to uniquely identify the step.
- A relevant IoT Sensor from a predefined list. (i.e., EC sensor, moisture sensor). This allows the plugin to use the dynamic data from the sensor at runtime.

This association will allow the system to route the sensor inputs appropriately and execute logic based on the readings from real-world data when the workflow is executed later.

Two Modes of Plugin Definition

The system provides two alternative methods for specifying plugin behavior:

- **Visual Flowchart Designer**

This method allows the exporter to define the plugin logic utilizing a drag-and-drop functionality. The experience has a palette of icons (shapes) sorted into categories to represent types of operations:

- Input blocks (e.g., receive sensor value)
- Conditional blocks (e.g., IF EC > 1.5)
- Action blocks (e.g., Repeat step, Proceed)

- Termination blocks (e.g., End)

The user creates a logical flowchart by dragging and connecting different blocks (shapes) right on the canvas. Each block represents a functional module—condition, action, data input, etc. This user interface differs from traditional flowchart applications that separate the items via properties panels. In contrast, the user has the ability to edit the content of the block in the block itself—such as entering the sensor thresholds or condition logic available within the shape. It enhances the interactive experience while simultaneously customizing functions by allowing more non-technical users to use and customize. The user experience is supported by comprehensive user documentation that ensures clarity and consistency while assisting exporters in the proper definition of logic using the available flowchart elements.

- **DSL-Based Workflow Editor**

As an alternative, users can define instructions using structured syntax specific to coco-peat processing language by utilizing the DSL editor. Even for users without any programming knowledge, the DSL syntax specifically created for this tool is simple and easy to understand.

2.1.3.2 Bidirectional Conversion and Real-time Validation

A key feature of the plugin creation tool is the bidirectional conversion between the visual flowchart and the DSL editor. Which is, users will be able to:

- Build a flowchart and automatically create the equivalent of DSL instructions.
- Write DSL instructions and generate them as a flowchart.

For each conversion, the system will perform validity checks including:

- Syntax validity for DSL.
- Logical flow validity for flowchart (e.g., missing paths, unreachable steps).
- Compatibility with selected sensor input.
- Should be a valid starting point and ending point in the logic.

All validities will be indicated in real-time with visual highlights and accompanied by descriptive feedback messages before proceeding with a valid plugin logic structure.

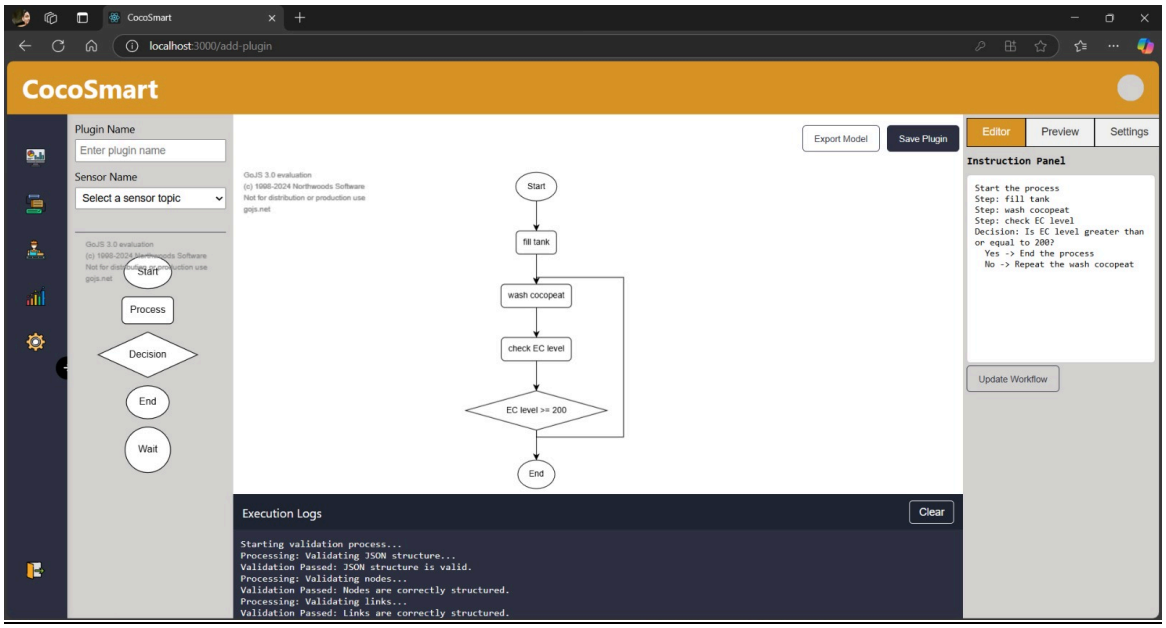


Figure 3: New Plugin Creation using Visual Editor and DSL Instructions Editor

2.1.3.3 Automated Plugin Packaging

Following a successful validation process, the system enters the packaging phase, which automates the deployment of the backend. This includes:

- Go Source File Creation:
  The flowchart or DSL input is used to build a Go file that represents the plugin's core functionality. This file specifies the runtime environment in which the plugin will be executed.

- Creation of a Docker file:
  To specify the dependencies, runtime environment, and execution command for the plugin, a Docker file is created. This ensures system isolation and consistency.

- Plugin Folder Creation:
  A specially designated folder linked to the plugin contains both files. Metadata like the sensor type, plugin ID, and creation timestamp might also be present in this directory.

- Zipping and Uploading:
  The folder is automatically compressed and uploaded to the external Core System using the gRPC protocol. The upload indicates that the plugin is prepared for runtime environment execution and registration.

2.1.3.4 Plugin Availability and Reusability

After being successfully uploaded and registered by the Core System:

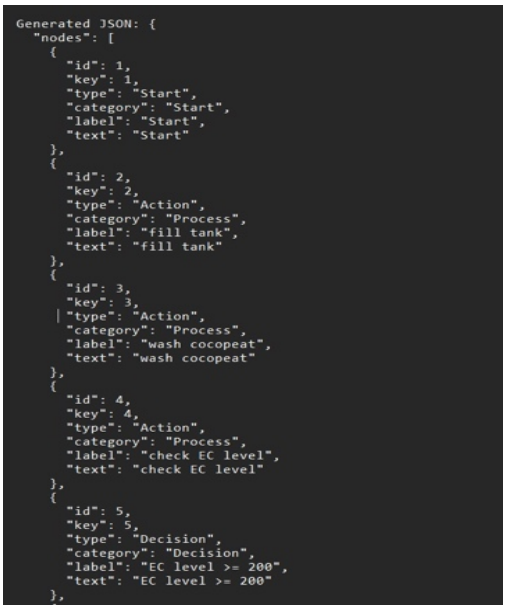
- The plugin appears in the frontend sidebar next to other plugins that have already been defined.
- The plugin is reusable by exporters in any process they create in the future.
- The plugin is now accessible from any workflow version and manufacturer assignment since it is a part of the plugin registry.

By ensuring that plugin development is non-technical, scalable, and modular, this lifecycle enables exporters to innovate and adapt to changing production needs without the assistance of developers.

2.1.4 DSL and Validation Mechanism

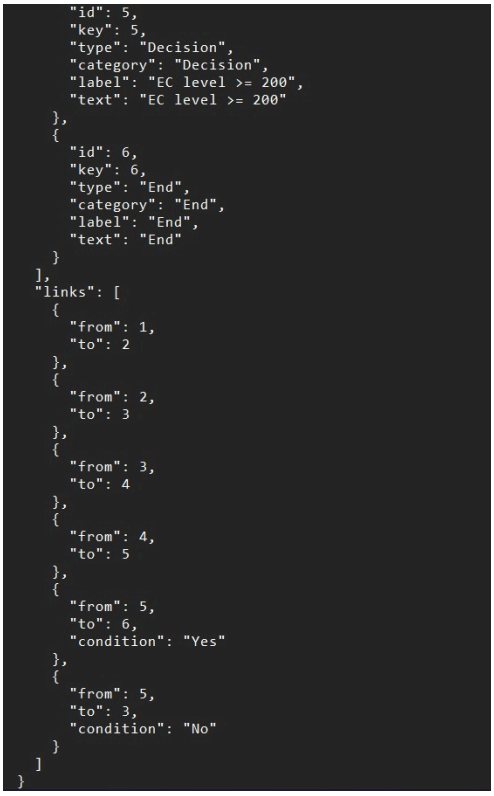
An important aspect of this system is the ability to establish plugin logic via a DSL or an interactive flowchart editor. These offer two ways of accommodating user preferences—non-technical users can utilize a drag-and-drop logic composition method while others can choose to write rules in a more readable advice format. The DSL and flowchart editor are tightly coupled through a bidirectional conversion mechanism to allow easy navigation between both views.

The design of the DSL emphasizes simplicity and a connection to domain-specific contexts. A plugin's logic structure represents a sequence of actions and a decision based on a condition using a syntax that looks structured and natural-language-like. The logic of every plugin is saved in JSON format. Nodes (steps, conditions, start/end locations) and links (flow pathways) connecting nodes make up the JSON structure.



```
Generated JSON: {
  "nodes": [
    {
      "id": 1,
      "key": 1,
      "type": "Start",
      "category": "Start",
      "label": "Start",
      "text": "Start"
    },
    {
      "id": 2,
      "key": 2,
      "type": "Action",
      "category": "Process",
      "label": "fill tank",
      "text": "fill tank"
    },
    {
      "id": 3,
      "key": 3,
      "type": "Action",
      "category": "Process",
      "label": "wash cocopeat",
      "text": "wash cocopeat"
    },
    {
      "id": 4,
      "key": 4,
      "type": "Action",
      "category": "Process",
      "label": "check EC level",
      "text": "check EC level"
    },
    {
      "id": 5,
      "key": 5,
      "type": "Decision",
      "category": "Decision",
      "label": "EC level >= 200",
      "text": "EC level >= 200"
    },
```


Figure 4: JSON format of the nodes



```
      "id": 5,
      "key": 5,
      "type": "Decision",
      "category": "Decision",
      "label": "EC level >= 200",
      "text": "EC level >= 200"
    },
    {
      "id": 6,
      "key": 6,
      "type": "End",
      "category": "End",
      "label": "End",
      "text": "End"
    }
  ],
  "links": [
    {
      "from": 1,
      "to": 2
    },
    {
      "from": 2,
      "to": 3
    },
    {
      "from": 3,
      "to": 4
    },
    {
      "from": 4,
      "to": 5
    },
    {
      "from": 5,
      "to": 6,
      "condition": "Yes"
    },
    {
      "from": 5,
      "to": 3,
      "condition": "No"
    }
  ]
}
```


Figure 5: JSON format of links

When a plugin is built visually using the flowchart editor, it generates its logic DSL using the function generateDSL(). Prior to generating instructions, the system executes validation across several stages, checking that the logic is complete, syntactically valid, and safe to execute. The validation process is an asynchronous promise-based function, and as it executes, logs and instructions are generated via callback methods.

The validation procedures listed below are completed in order:

1. JSON Structure Validation:
   Verifies that nodes and links are arrays with the proper structure.

2. Node Validation:
   Verifies that every node has a distinct key, belongs to a legitimate category (Start, Process, Decision, End), and has content that is readable by humans. Invalid node types and duplicate or absent keys are regarded as validation errors.

3. Link Validation:
   Verifies that every connection accurately refers to and from nodes. This validation stage makes sure that there are no wrongly placed references or broken flow transitions in the workflow.

4. Start Node Check:
   Verifies the existence of a single Start node with a minimum of one outgoing link.

5. End Node Verification:
   Confirms that there is only one End node and that it has no presence of outbound links. This ensures a clean completion of the process.

6. Decision Node Authority Rules:
   Verifies that the condition expression (e.g., >=, ==) is usable and that decision nodes contain a minimum of two outgoing branches (Yes/No). Something along the lines of Is EC level >= 200? is parsed to make DSL commands comprehensible by humans.

7. Orphaned Node Detection:
   This feature finds and eliminates nodes that are obviously off of the workflow route and may result in logic that cannot be reached.

The user receives feedback for each validation outcome in the frontend as messages logged in real-time, allowing them to notice and fix issues right away. If all validations pass, the system traverses the validated graph structure, and generates DSL instructions, starting from the Start node and recursively moving through the connected node. If there are decision branches, Yes and No instructions are inserted into the list based on the order in which they occur in the graph.

Beyond DSL from flowcharts, the system allows the user to do the opposite: take DSL instructions and render them back into JSON as a flowchart. This assures that the user can start in either mode, shift

between states whenever desired, yet maintain the same logical flow. Validation occurs in both directions so that if an issue arises, it can be identified logically no matter how the user constructs the plugin.

This DSL and validation mechanism offers strong guarantees for the logical correctness of plugins before they are produced and sent out to the core system. It enables non-technical users to safely express complex logic in their workflows without writing code, and it prevents wrong workflows from being configured and/or executed. The stepwise validation flow and real-time error feedback also enhance the user experience, mitigate configuration errors, and contribute to the system's overall robustness and reliability.

2.1.5 Version Control and Assignment

The system handles a version control function at the workflow level. Each new or updated version of a workflow is saved under a designated Workflow ID and is assigned an increasing version number (Version1, Version2, etc.).

This allows for the following to occur:

- Allows exporters to experiment with workflow design and implementation.
- Enables auditability and traceability.
- Enables the rollback of previous, older, or comparative versions.

The workflow state changes from "Draft" to "Pending" after a version is approved by the exporter and given to a manufacturer. The manufacturer will be able to perform only the assigned workflows. This process of manual approval ensures only approved and validated workflows are deployed in the production environment.

2.1.6 Workflow Execution by Manufacturer

Manufacturers log in to view the workflows assigned to them. Each workflow includes the following visual information:

- Workflow ID and title
- List of plugins with execution order
- Sub steps in each plugin
- Status indicators (Not Started, In Progress, Completed)

Manufacturers can:

- Start each plugin step and sub step.
- Observe sensor data .
- View returned validation results or system messages.
- Submit step completion logs.

These actions will be captured and timestamped in the system so that exporters can see real-time execution as what are currently running, completed, or pending. This ability to see the real-time execution status enhances accountability and provides more transparency to the supply chain process.

2.1.7 IoT Integration and Real-Time Feedback

The system design supports sensor-aware workflow configuration by allowing exporters to create IoT-based rules within workflows. While the IoT sensors are connected to the external Core System, this aspect of the system provides frontend and backend functionality that allows users to use sensor logic in definitions of plugins.

Exporters can create conditional logic in DSL (for example, “If EC > 1.5, repeat the washing step”) that is encapsulated in the plugin definition at the time of plugin creation. The Core System will then evaluate a DSL rule during runtime while executing the workflow.

When a manufacturer executes the plugin with sensor conditions,

- The core system will handle the collection of data from the linked IoT sensors (such as moisture and EC level) when a manufacturer runs the plugin with sensor conditions.
- Using the sensor reading as specified by the DSL's logic, the core system will determine whether the condition is met.
- The core system will decide to proceed, repeat, or skip the step based on that decision.

While this logic for execution occurs externally, the exporter controls how to define them through the frontend plugin creation interface. The sensor-related conditions become part of the DSL instructions and the logic defined in the plugin, which makes the conditions actionable by the Core System as an incentive.

To ensure transparency and traceability:

- The Core system uses gRPC to send execution logs or the results of those decisions (or actions) based on sensor data back to the backend.

- Afterwards that, they are retrieved and displayed to manufacturers (as prompts or statuses) and exporters (in the activity log) via the frontend interface.

By enabling domain experts to set up conditional logic based on sensor inputs, this indirect integration ensures that even while the workflow customization tool does not analyze IoT data, it is essential in making workflows responsive to environmental variability.

2.2 Commercialization Aspect of the Product

The proposed visual programming workflow customization tool has been designed with strong focus within the coco-peat manufacturing industry in the agricultural exports sector in Sri Lanka. This section explores insight into the commercialization potential of the product, outlining the value proposition to the stakeholders involved, and importantly the readiness for market, scalability and advantages over competitors within an evolving Agri-tech space.

2.2.1 Market Opportunity and Problem Fix

Grading, washing, drying, packing, and delivering are just a few of the domain-specific processes involved in the manufacture of coco-peat; many of them need to be modified frequently in response to customer requirements or environmental circumstances. Nevertheless, the digital workflow technologies currently available on the market are either too general-purpose, overly complex, or not made for non-programmers to execute at the manufacturing floor level.

By providing a domain-specific, no-code, and user-friendly visual interface, the suggested system fills this gap and enables manufacturers and exporters to manage, modify, and model operations without the help of software engineers. It directly addresses the following needs of the market:

1. Increasing global standards for traceability and process transparency in agriculture:
   As international buyers and certification organizations have shifted their focus toward traceability, manufacturers must show how and when processes are carried out in their production. The proposed system supports this by providing the high-level definition of workflow, version control, and logging execution. Hence, providing transparency of everything they have done each step of the way.

2. Medium-sized production requires automation that is both controllable and adaptable:
   Although they lack the resources for large-scale IT solutions, medium-scale coco-peat producers require flexible workflow automation to satisfy evolving client requirements. This

program provides a modular, cost-effective solution that allows users define and modify procedures while maintaining execution integrity and validation.

3. Offering non-technical users, the ability to engage in part in the configuration and control of processes:
   Workflow design is usually managed by IT professionals, while in manufacturing environments, the actual knowledge of operational management lies with the domain experts such as manufacturers and exporters. By providing a context-specific, no-code visual interface, this technology delivers control in their hands and empowers users to manage activities efficiently and autonomously.

This solution acts as a bridge between the backend execution systems and non-technical users. The external core system handles the actual execution, such as executing plugin code or responding to IoT sensor data, while exporters and manufacturers engage with a visual interface to specify workflows and logic. Essentially, the tool converts user-defined, domain-specific workflows into logic that can be executed on the backend, facilitating the smooth transition between automated execution environments and human process planning.

2.2.2 Value Proposition

There are several advantages to commercialization using this method:

Table 1:Different features of the tool and their benefits

Feature	Benefit
Drag-and-drop workflow builder	Reduces training time and eliminates obstacles related to programming.
DSL or visual editor for plugin creation	Allows steps to be designed for production requirements.
Workflow version controlling	Ensures quality control, traceability, and repeatability.
Integration with IoT sensor data (via core)	Allows for sensor-driven, real-time decision-making without the need for complicated programming.
Lightweight backend and modular structure	Enables implementation, even for small and medium-sized manufacturers.
Empowers domain-experts	Allow to create and manage workflows on their own without assistance from programmers or IT experts.

These characteristics provide an affordable and incredibly flexible way to digitize and streamline production processes in the agricultural processing industry.

2.2.3 Target Audience and Adoption

The system's primary end-users are manufacturers and exporters of coco-peat, especially small and medium-sized enterprises where traceability, flexibility, and adaptability is critical. Manufacturers are the ones who carry out these workflows as part of their daily operations, but exporters form a significant group that develops workflows in response to client needs with certain specifications guided by export quality standards and environmental conditions.

Non-technical users, such as factory supervisors, production coordinators, and exporter operations managers, who have their domain knowledge but lack formal experience in programming or system configuration, are the target audience for this system. Instead of depending on IT specialists, a non-technical user should be able to independently model and change processes by using a drag-and-drop user interface and pre-built logic tools with domain-specific functionality.

In terms of adoption, system could initiate as a trial at a single processing site of one manufacturer to test effectiveness and usability. Upon successful deployment, the system can be applied horizontally across other exporters and cooperatives in the agricultural supply chain. In addition, Agri-tech solution providers or industry facilitators (like regional development boards or government innovation agencies) may leverage the adoption of the tool across processing sites in order to enhance traceability and comply with export conditions.

2.2.4 Revenue Model

The system has a lot of potential for commercialization through several revenue models, depending on how it is packaged and supplied, even though it is being developed as a research project. Possible monetization techniques:

- **Software-as-a-Service (SaaS):** Exporters and manufacturers can purchase the product as a web-based subscription platform, paying a monthly or yearly charge that is determined by the quantity of users or workflows they manage.

- **Per-Factory Licensing:** Considering tiers of pricing for small, medium, and large businesses, a one-time or renewal license may be granted per factory or exporter account.

- **Plugin Marketplace:** An ecosystem of validated and reusable plugins may develop as use increases. Plugins for specialized processing activities could be created and published by domain experts or qualified consultants, with platform providers and developers splitting the profits.

- **Training and Support Packages:** For businesses that need help during the implementation stage, paid onboarding, training sessions, and high-quality support services may be provided.

These models provide flexibility in aligning the product to high-value export-oriented businesses as well as low-cost rural activities.

2.2.5 Scalability and Future Expansion

The system has been developed to support modularity and extensibility to allow for high technical and operational scalability.

From a technical aspect:

- The plug-in architecture allows for new task and processing logic to be integrated without affecting the developmental core.
- The DSL and visual interface can be extended for more advanced condition blocks, parallel task execution, or approvals.
- Third party API integrations can be introduced as optional modularity such as quality certification systems or export logistics requirements.

From a business aspect:

- The solution can be modified beyond the coco-peat project to other agro-processing sectors such as tea, cinnamon, coir, or organic composting projects. Each have similar workflows and requirements for traceability.
- The solution can also be localized to different languages and compliance frameworks, to support international markets.
- In the long-term additional modular features can be added such as mobile interfaces.

This will create a complete suite of production and supply change management tools. Overall, the system was designed as not just a static tool for coco-peat processing, but as a foundation for a dynamic and evolving platform that supports agile workflow and management for the agricultural processing sector.

2.3 Testing and Implementation

This chapter provides a comprehensive description about the implementation and testing processes used to build, integrate, and authenticate the visual workflow customization tool. The system attempts to serve as a bridge between technical and non-technical users and environments for execution in coco-peat manufacturing. In implementation, the system introduced main modules such as a drag-and-drop workflow builder, plugin creation module, DSL validation engine, and integration layer with the external core execution system. In testing, the system was assessed for correctness, reliability, usability, and consistent interaction through unit, integration, validation, and system testing.

2.3.1 System Implementation Overview

2.3.1.1 Frontend Interface (User Interaction Layer)

The proposed tool was created as a modular software system, which makes maintenance, scalability, and concern separation simple. With a frontend, backend, and integration into an external core system,

the software is structured as a layered system. Each layer was in charge of operations and interacted with the others via standardized protocols like gRPC and REST APIs.
The system's frontend functions as the primary interaction point for exporters and manufacturers, allowing non-technical users to visually define, manage, and configure the workflows and plugins. The frontend is built using React.js, a component-based JavaScript library which enables reactive user interfaces and dynamic state management.

To maintain a uniform and modular visual design, the system uses styled-components, a popular CSS-in-JS framework for styling React components. Styled-components improve maintainability and make the design process more responsive by enabling styles to be scoped to individual components and modified dynamically based on properties and the current state of the application. This method, which is tightly coupled with the logic and states of the Workflow Builder and Plugin Creation modules, has produced a visually refined interface.

The frontend consists of the following key interactive modules:

1. Drag-and-Drop Workflow Builder

This module offers exporters a simple way to construct workflows by enabling drag-and-drop functionality, which uses react-beautiful-dnd to enable the dragging and dropping of pre-defined plugins from a sidebar onto a central canvas, making it simple and accessible to order and reorder items within the workflow canvas.

When the user adds a plugin to the canvas, the user can edit required values (e.g., thresholds, time durations) that are step-specific directly into the plugin block. Meanwhile, as the workflow structure develops, a serialization of the layout and configuration is continuously transmitted to be stored and versioned later in the backend.

2. Flowchart Editor (Plugin Logic Designer using GoJS):

   The flowchart editor provides the ability for users to visually design the plugin's internal logic, especially useful for anyone who is not a programmer and prefers a diagrammatic context over textual programming context. The flowchart editor in GoJS is a powerful JavaScript library for building interactive diagrams.

   Specifically, the implementation includes:

   - The custom GoJS diagram component, which is set up with node templates depending on the various logic types (Start, Action, Condition, End).

- Users can easily drag and connect blocks (nodes) with links (arrows) and even specify the direction of flow between blocks.
- Each block is available to edit in place—users can just click into the shape to add logic like “EC > 100,” instead of needing to use an external property panel.

Internally the flowchart is just a GoJS model represented in JSON format, which records all nodes, links, and metadata; the JSON can be persisted and also processed by the backend for DSL generation and validation.

3. DSL-Based Editor

In addition to a visual editor, the system provides a text-based DSL editor that allows users to specify the plugin logic themselves using the project’s custom DSL, which is designed to be lightweight and easy to read.

A real-time DSL editor is implemented using a controlled text area component in React that enables users to write and edit domain-specific instructions directly. As a controlled input, the text area maintains its value in the state of the React component, so the system can respond to state changes in real time. This enables the system to validate the string and structure of any DSL instructions as the user types, which ensures that only valid logic can reach the build stage of being executed. While this is lightweight and minimal, it is effective in guiding non-programmers through a structured, rule-based method of creating a plugin, without requiring interaction with complicated code editors.

4. Bidirectional Conversion via JSON

A very powerful feature of the system is the way that it allows bidirectional synchronization between the DSL and the flowchart view. This work is done by:

- Converting the flowchart representation (utilizing GoJS) into a regular JSON schema.
- Parsing that JSON into a corresponding DSL command set using the backend logic.
- Alternatively, parsing DSL command set into a JSON structure and return it to GoJS to create the flowchart representation.

The intermediary JSON representation acts as a bridge early on in the process to handle the visual vs textual representation, ensuring that:

- The same logic is evaluated similarly regardless of either representation mode.
- Changes made in flow or DSL are reflected immediately following either representation mode's conversion.

- The structure and movement (e.g. connecting nodes, conditional logic) are maintained and confirmed before saving.

This overall architecture allows users to fluidly shift between visual and DSL representations based on their comfort, while encapsulating both representation modes in the same operation for consistency, validation, and saving in a regular base representation.

2.3.1.2 Backend Logic Layer

The workflow customization system's main processing component is the backend logic layer. It develops plugins, handles workflow data, assesses logic, extracts user actions, and controls all external Core System interactions. The event-driven, non-blocking architecture of Node.js, which was chosen for this layer's construction, makes it appropriate for I/O-intensive operations (e.g., API calls, reading and writing files, and reading and writing data in a database). There are several logical modules that compose up the entire backend. Every module is in charge of a particular aspect of the system's operation. Each module manages any persistent data using MongoDB and interacts with the Core System (using gRPC) and the frontend (using REST APIs).

1. Workflow Processing and Versioning

When an exporter creates a workflow in the drag-and-drop interface, the data about the structure and metadata (the plugin sequence, the configuration values, and the plugin IDs) is sent to the backend. Here:

- A new workflow gets a unique Workflow ID.
- A version number is assigned to each workflow build, such as Version 1, Version 2, and so on.
- Every version has the same Workflow ID and is stored in MongoDB.

The version number, timestamp, assigned manufacturer, and current stage (Draft/Pending) are all included in the metadata. Users can modify workflows over time with this procedure, which also preserves the history for auditability and possible reversal.

2. DSL Parsing and Validation Engine

In the backend, there is a custom DSL validation engine that reviews textual logic that has been written by the exporter or produced through a flowchart. What is being validated is:

- Syntax Validation: Making sure the instruction follows the domain-specific grammar.

- Keyword Validation: Validating whether supported DSL terms and operations are utilized.
- Sensor Validation: Validating all referenced IoT sensors to be valid and registered.
- Structural Validation: Finding incompleteness of rules, un-reachable conditions, and circular logic.

The DSL is first parsed into a structured JSON representation; this representation allows for a common intermediary between the textual editing and the visual editor. If the DSL validates successfully, the same JSON will be continue to be used by the backend, for additional processing.

3. Bidirectional DSL – Flowchart Conversion

The flowchart editor and DSL-based editor are both derived from the same JSON schema.

- When users are working on the flowchart editor, GoJS serializes the diagram into a JSON structure with nodes (steps) and links (flow).
- This JSON is sent to the backend and translated into DSL instructions.
- When the DSL instructions are written, the backend parses it back into the same JSON structure to send back to the frontend to render the flowchart.

This allows customers to switch between views with the same logic for parsing and validation.

4. Plugin Handler Module

The Plugin Handler is a critical component that automatically converts the validated plugin logic into deployable packages of code that can run in a runtime. This process is made up of the following steps:

- Go File Creation: The logic is represented using either DSL or flowchart-style logic to create a .go file that describes the plugin's behavior.
- Docker file Creation: Also, a Docker file will be created in order to represent the runtime environment in which the code will run. The Docker file will describe the base image, the commands to run the application and the dependencies you specified.
- Folder Structure: Both of these files will be located in a folder named from the name of the plugin title from the user, along with a generated timestamp.
- Zipping: Once encapsulated in a folder, the file would be zipped for transport.
- gRPC Upload: This zipped file is sent over a gRPC connection to the external Core System, which makes sure that the plugin is registered and is executable in the core runtime.

Once registered, the plugin would be displayed in the frontend's sidebar and available to use in any workflow.

5. Core System Communication via gRPC

The backend keeps a live gRPC connection with the Core System that allows it to:

- Register plugins that have been uploaded by the exporter.
- Assign workflows to specific manufacturers.
- Receive logs and execution feedback, which will be sent to the frontend for the exporters and manufacturers to see.

This communication layer abstracts the complexity of low-level runtime behavior, which allows the workflow customization tool to remain focused on configuration and management.

6. Workflow and Plugin Metadata Management

In addition to managing logic and validation, the backend will also manage metadata, including:

- Workflow id, title, version history, status, and user assignment.
- Plugin name, sensor association, creation time/date, and registration status.
- Execution logs (start and end timestamps, if there were any errors, and completion flags).

All of this metadata is stored in MongoDB, which allows simple querying, filtering, and history tracking of a lot of information, which is a key to full transparency throughout all system components that supports future transformation features such as reporting, dashboards, analytics, etc.

2.3.2 Testing Strategy

2.3.2.1 Unit Testing

2.3.2.1.1 Frontend Unit Testing

Frontend unit testing was performed to confirm the functionality of individual UI components, input fields, state management, and logic for context. The core purpose was to confirm that key components like the canvas, DSL viewer, and plugin input forms rendered and behaved as expected when presented in isolation from the full system.

All frontend tests were generated using React Testing Library (RTL) and Jest and fireEvent, which allowed the simulation of user interaction.

Using the React Testing Library (RTL), the Canvas Component was one of the main areas tested for how the plugin blocks rendered after being dragged and dropped on the canvas. The tests verified that the correct number of plugins showed after interacting with the object and that each plugin's label matched the user's input or selection. This demonstrated that the logic for rendering plugins worked correctly.

Further, the tests confirmed that the DSL View Component behaved accurately, displaying the transformed DSL code in the preview area. The generated instructions are rendered using a <pre> element by the DSL View Component, and the preview is updated in real-time if modifications are made to the plugin logic or flowchart structure.

Input handling was tested with the Plugin Form component, which accepts configuration values like sensor thresholds. The Plugin Form was tested using Jest and the React Testing Library and found that the component accepted user data in the input fields, and confirmed that upon the form being submitted appropriate callback functions were triggered with the proper values—ensuring the user input is communicating back to the backend to prepare for processing logic.

The custom React Context created for this tool was also used to evaluate the plugin state management mechanism. The tests confirmed that the initial context state was empty, and upon calling the addPlugin() function successfully, the context updated as expected, and dependent components re-rendered with the updated state. This demonstrated that context was being shared correctly across components.

All of the tests were performed using the React Testing Library and Jest, and the fireEvent was used to simulate user actions like clicking, typing, or submitting forms. Overall, the suite of unit tests confirms that the frontend components are functional and ready to call the backend integration.

2.3.2.1.1 Backend Unit Testing

The accuracy of the system's fundamental logic and API operations was confirmed through backend unit testing. Jest, a popular JavaScript testing framework that works with Node.js, was used to run these tests. In order to confirm the expected behaviour for both typical and edge conditions, the focus was on evaluating isolated modules for file generation, plugin management, and user authentication.

To check if a new user was registered, the registerUser(req) function was tested. When the email was unique to the system, this function passed testing; when the email was already registered, it gave the appropriate error messages. To confirm appropriate authentication behaviour, the authentication module was tested using loginUser(req). The tests specifically evaluated if true credentials produced a secure, authorised JWT token. To make sure the system could distinguish between authentication failure scenarios, additional tests verified the system's response to invalid case email entries and when the password was entered incorrectly.

The logic for file manipulation was validated through the updateFile(content, filePath) function. Test confirmed that valid content was effectively written to the file system, while missing content led to the expected error handling. Similarly, the functionality was demonstrated by the generateFile(...) function, generating .go files for plugins at the specified location while also storing the associated errors for the validation of mandatory inputs.

The unit tests validated that all backend logic was implemented in a safe and reliable manner, ensuring that each core function performed before being integrated into the larger system.

2.3.2.2 Integration Testing

Though integration testing focusses for the way various system components interact with one another as together, unit testing evaluates the accuracy of individual functions. To make sure that the backend modules in charge of processing, creating, packing, and uploading the plugins worked together accurately and error-free, integration testing was required for this project.

The main purpose of the integration tests was centered around the processAll function of the complete backend pipeline to receive a plugin, process it, and register it once it has been uploaded. The processAll function acts as a sequence of dependent actions:

- Updating content in the Docker file.
- Generating the Go source code file.
- Zipping the folder of content that contains the plugin files.
- Uploading the zipped archive to the Core System via gRPC.
- Returning a success status back to the frontend.

The aim of the tests was to simulate a real API request to the ‘/api/plugins/processAll’ endpoint to verify that each step was executed correctly (without actually running a gRPC server or any element that interacted with the file system, etc.).

To accomplish this, various tools and libraries were used:

Table 2; Various tools and their purpose used for integration testing

Tool	**Purpose**
Jest	Tester runnter for executing and organizing the test suite
Supertest	Simulates HTTP requests to the API routes from Express.js
mock-fs	Mocks the Node.js file system in order to safely test file operations
sinon	Stubs the gRPC upload function, simulating responses from the server

The test was started by stubbing the gRPC UploadFile method using sinon, which would allow to simulate the plugin uploading successfully without creating a live connection to the Core System. Then used mock-fs to fake the plugin's folder structure and files, so no alteration of the real file system would occur during test execution.

Next, a POST request to the /api/plugins/processAll endpoint with Supertest was tested, simulating a user action from the frontend. The request payload included:

- New Docker file content
- Go plugin code
- Plugin name and other metadata

This triggered the full processAll pipeline. The test would take a look to verify that:

- The mocked file system accurately represented the updated and generated files.
- The zip archive was created.
- The gRPC upload function was invoked and returned a success message.
- The API response had the expected status and success message.

This test would be classified as an integration test, as it tested the interactions between multiple modules with functionality, such as:

- File generation and update,
- Zipping logic,
- gRPC communication,
- API routing and response handling.

Although mocks were used to represent external systems, the logic flow through the backend remained

unchanged, allowing normal system behavior to be replicated accurately in a test environment. Confirming the successful execution of this test indicates that the plugin processing pipeline within the system is integrated and functional, validating the stability of backend automation before the workflow is transferred to the core execution layer.

2.3.2.3 Validation Testing

Validation testing was conducted to capture incomplete or invalid data submitted by users and execute processes before reaching critical execution stages. Given that the system allows non-technical users to create custom workflows and plugins, there was a need for robust validation, so this was also a core requirement. Ultimately, the validation tests showed that workflows and plugins logic would only be accepted if they adhered to both structural restrictions and any business-specific constraints.

Validation testing consisted of two main areas: (1) validation of workflows created through the tool, and (2) validating flowcharts before DSL generation. All validation functions were tested using Jest and mocked requests, responses, and callbacks so they could be run independently of other system parts.

2.3.2.3.1 Workflow Validation Testing

To maintain workflow integrity and domain specific adherence, a series of middleware functions were tested, which reside on the backend and are triggered during the workflow submit process that allow the system to check for various factors.

Table 3 Implemented workflow validations function and purpose

Validator Function	**Purpose**
validateWorkflowStructure	Ensures steps array exists, valid plugin names, and unique step orders
validateRequiredPlugins	Ensures the mandatory steps (e.g., grading, cutting, washing) exist
validateWorkflowOrder	Ensures that workflow follows proper order: grading, cutting, washing
validateWorkflowVersion	Confirms version of workflow submitted matches the one in the database

Test Summary:

- Rejected workflows missing steps, duplicate step orders, or invalid plugin names.

- Identified and rejected workflows that were submitted out of order, or had unsupported plugin combinations.
- Properly identified version inconsistencies before assigning a workflow to the manufacturer.

As a result, it confirmed that the middleware layer was able to correctly enforce workflow errors and adhere to plugin capabilities before a workflow could saved, assigned, or executed.

2.3.2.3.2 DSL-to-Flowchart Validation Testing

Another important point of validation was the conversion of flowchart diagrams to DSL code. Throughout both directions of conversion between DSL and flowchart representations, validation was applied, as both conversions leverage a common intermediate JSON representation. Regardless of whether the user created a flowchart graphically or wrote the DSL code manually, the system takes the input and converts it to a JSON representation, which it then validates before rendering or generating. The generateDSL() function is responsible for flowchart to DSL conversions, while the DSL to flowchart parser uses the same validation logic on the run, to validate the DSL before re-rendering the flowchart diagram.

Table 4: How each node is validated in during the conversion of DSL instructions and flowchart

Validation Step	**Purpose**
JSON Structure	Checks that nodes and links are in an array structure
Node Validation	Validates that the keys are unique, there are valid categories (e.g., Start, Process, Decision, End), and text fields are provided where required
Link Validation	Ensures that every from and to node ID exists in the set of nodes
Start/End Rules	Ensures there is one Start node and one End node, along with the appropriate connection rules
Decision Node Rules	Validates that each of the decision nodes has at least 2 links outgoing links and valid conditions
Orphan Detection	Identifies isolated (orphaned) nodes that are not connected to the flow

Test Summary:

- Rejected invalid or incomplete JSON representations.
- Validated that start/end nodes were missing, the flow was disconnected, and decision boxes were malformed.
- Valid flowcharts were converted to valid DSL with proper logging and validation.

These validations were crucial to verifying that the flowchart could be reliably construed in executable plugin logic and, importantly, that non-technical users would receive real-time feedback during the plugin design process.

To accomplish this, various tools and libraries were used:

Table 5: Tools and methodologies used for validation testing

Tool/ Methodology	**Purpose**
Jest	Main testing framework for verifying unit and middleware
Sinon / Jest Mocks	Stubbed Express.js request (req), response (res), and next() method
Set-based checks	Internally utilized for detecting uniqueness (e.g., step order, node keys) and relationship integrity (e.g., link references)

As a result of the validation testing, the system showed a high capacity for detecting and handling user errors before they could affect their workflow execution or plugins were deployed. The middleware-based workflow validation effectively blocked all incomplete or insufficiently structured submissions, and direct user entry of workflows to the system was not permitted unless there was proper workflow structure. The flowchart and DSL conversion mechanisms always detected logical errors, such as missing starting or endpoint, orphaned nodes, and missing branching decisions before build or save was allowed. Validations performed also enforced structural and logical integrity of both workflows and plugins. Additionally, the reaction time of the plug-in frontend ensured logical, structural user experience by providing clear and immediate feedback. Validation testing confirmed the system fundamentally adhered to rules of the domain, reduced the possibility of user configuration error, and generally did not compromise the stability or security of the user-built workflow itself.

2.3.2.4 System Testing

System testing was undertaken to assess the software in its entirety, as an operating system. The goal was to confirm the system is capable of executing real-world workflows from an exporter and manufacturer perspective, including plugin creation, workflow definition, validation, packaging, assignment, and execution. The focus of this phase of testing was to ensure all components of the previously tested modules—frontend interface, backend logic, DSL validation, and plugin logic—operated together as designed.

A series of end-to-end test scripts were executed to simulate common use case scenarios. Exporters were able to create workflows using the drag-and-drop builder, create plugin logic using either the flowchart or DSL editor, switch between the visual and code view safely, and all content was validated. When validating plugins, plugins were automatically packaged into Go and Docker files, zipped, and uploaded to the Core System using mocked gRPC. Versions of workflows were saved and assigned, and all statuses were updated as workflows were executed.

Error handling was tested by submitting malformed workflows, invalid DSL code, and incomplete flowcharts, all of which were correctly rejected by the system while presenting user-friendly error messages and did not permit invalid data to continue to execution. Testing functionality was also verified on manufacturer's side by simulating running of workflows and confirming that the user has successfully seen progress messages with the accurate status of the workflows.

The system testing results verify the application performs as intended and can be used in real-world contexts. The application supports non-technical users being able to identify and manage workflows on their own, properly validated input, was able to complete operations throughout the total workflow lifecycle, while maintaining all communication throughout all application layers. The application is stable, functional, and ready for pilot deployment.

3. RESULTS AND DISCUSSION

3.1 Overview

This chapter discusses the findings from the testing stage and highlights a critical evaluation of the system's performance against its intended objectives. It evaluates the reliability, usability, and overall effectiveness of the tool in real use cases, with a focus on the perspective of non-technical users like exporters and manufacturers. Furthermore, It discusses the accuracy and validity of the essential functions, the limitations of the tool's use, and how it might practically be improved to make it more robust for use in the future.

3.2 Testing Outcomes and System Results

The system completed a thorough testing procedure that included unit testing, integration testing, system testing and validation testing, all of which focused on a particular set of functionality and the interaction of components.

In practice, exporters were able to:

- Develop workflows with drag-and-drop tool.
- Specify plugin logic through either a visual flowchart or DSL instructions.
- Switch between visual and textual views without losing logic.
- Save workflow versions and assign them to manufacturers.

Manufacturers were able to:

- View assigned workflows.
- Execute a step-by-step procedure.
- Log their progress allowing exporters to monitor activity in real-time.

Each of these scenarios demonstrated a full cycle of plugin-creating, workflow-creating, assigning and executing, without requiring any technical knowledge. Each step in the process was confirmed in real-time as invalid state changes were not permitted to save or execute. Furthermore, the system displayed predictable behavior under simulated interaction with the Core System in file uploading and plugin registering, where mocked gRPC responses were used.

3.3 Research Findings

Several significant insights into how domain-specific, visual systems could assist non-technical users in industrial environments—particularly in the coco-peat industry—were discovered during the development and testing of the workflow customization tool. The following key findings summarize the practical contributions and resulting learning from the system:

1. **Non-technical users can create complex workflows on their own.**
   The most important finding was an exporter and maker, from their own experience, creating, updating, and assigning workflows without requiring any software development expertise. The drag-and-drop interface relieved cognitive load by allowing users to manage and represent processes visually. This finding indicates that mixing the visual programming paradigm with a domain-specific language can facilitate tasks that are rigorously considered to require technical programming action.

2. **Flexible user representation for visual and textual editors.**
   The system includes a flowchart visual editor and a textual DSL editor, and it lets users work in whichever format they are more comfortable in working with. In particular, it allows for seamless bidirectional conversion between flowchart and DSL. Dual representation makes the tool adaptive to different user preferences or skill level, tomake it inclusive, and highly accommodating for novice users and technically inclined domain specialists.

3. **Built-in validations prevent workflow and logic errors.**
   Another major outcome was the strength of the validation mechanisms that operate in real-time. During the testing of the product, the system prevented users from saving or executing a workflow that had logical or functional errors, every time. If an invalid workflow or plugin was present, users received prompt, helpful feedback indicating to them there were problems - such as missing steps, improper sequence, or bad logic. This feature adds to system robustness by assuring users that they will only move to the next step of the workflow with successfully executing and logically sound workflows.

4. **Plugin build is modular, reusable and extendable.**
   The plugin build process was highly effective and user-friendly. Users could build custom steps using the visual editor, convert them to DSL, and automatically build the associated Go and Docker files, which were zipped and uploaded to the core system for execution. Once built, the plugin was its own reusable component and would show up in the sidebar for use in the next workflow. It follows that the product is modular, extendable and accelerated workflow processes in a manufacturing context.

5. **Versioning of workflows increases traceability and quality assurance.**
   The addition of workflow versioning allowed users to keep multiple iterations of a workflow under a common identifier. Users could build workflows, test them, revise them, and compare versions before finalizing. This versioning model facilitates traceability, decreases the risk of incorrect modifications, and allows users to revert or reassign older versions if needed, which enhances confidence and auditability.

6. **Role-based interaction between exporters and manufacturers is effective.**
   The system provided a clear separation of responsibilities between exporters (who are responsible for the creative component of workflow design and assigning workflows) and manufacturers (who perform the execution component of workflows). Each role only had access to those functions required for their role and the interaction between the two roles was maintained through status updates and execution logs. This confirmed that the system design would fit well with the actual structure of real-world coco-peat production environments.

   These findings suggest that the workflow customization tool serves its stated purpose - to bridge domain knowledge with technical implementation, and provides a well-defined, reliable means of managing customizable manufacturing workflows in medium-scale industry.

3.4 Discussion and Observations

Though the system met its objectives and performed effectively during testing, several kinds of findings point to areas that could use further improvement:

- More Advanced Plugin Logic Support: The current DSL supports some basic conditional logic. his user tier would benefit from extra flexibility if nested conditions, loops, or reusable logic blocks were supported.
- Plugin Dependency Awareness: The system relies on the exporter to confirm plugin compatibility. Automated checks on dependencies and plugin relationships would reduce the risk of error.
- UI Feedback: Real-time validation feedback is currently shown through terminal views, and additional visual feedback (e.g., inline red borders, contextual popups, etc.) could enhance the user experience even further.

These insights highlight areas for upgrading from a working prototype to a production-quality

solution, even though none of them show apparent weaknesses.

3.5 Accuracy and Validation Effectiveness

To test the system's capacity to manage both valid and invalid user inputs, our team developed and implemented a set of 15 to 20 validation test cases. These test cases were designed to demonstrate scenarios that an exporter or manufacturer might perform as they create a workflow or configure a plugin. The validation tests consisted of testing positive cases (valid code, correct structure, valid syntax) and negative cases (missing pathways, invalid flowchart, invalid DSL expressions).

Validation was done both at the point of workflow submission (via the backend middleware) and at the point of processing the provided input (DSL engine flowchart conversion). The system was expected to:

- Accept inputs for workflows and plugins that had valid structures without interruption.
- Reject invalid input while displaying descriptive error messages visible to the user.
- Prevent faulty or illogical workflows from proceeding into the build or execution phase.

The performance was exactly as expected for every sample: it reliably accepted correct inputs and rejected invalid ones. This demonstrates that the system is prepared for use in practice by verifying that the backend middleware, conversion logic, and real-time validation engine functioning according to strategy.

4. CONCLUSIONS

This research set out to address the challenge faced by medium-scale manufacturing industries—particularly within the coco-peat sector—where workflow customization is needed frequently, yet is often hindered by a lack of technical expertise. Existing systems are either too generalized or require programming knowledge, limiting their usability by domain experts such as exporters and manufacturers. The goal was to bridge this gap through a domain-specific, visual workflow customization tool tailored to the operational needs and user skills of these stakeholders.

The proposed solution successfully enables non-technical users to independently create, customize, and assign workflows without coding. Through a drag-and-drop interface combined with a DSL editor, users can define logic visually or textually. Real-time validation mechanisms ensure the logical correctness of each plugin and workflow version before deployment. Furthermore, the tool supports plugin reusability, workflow versioning, and seamless role-based interaction between exporters and manufacturers.

Developed using a modular architecture consisting of React for the frontend, Node.js for the backend, MongoDB for persistent data handling, and gRPC for external communication, the system demonstrates the feasibility of empowering domain users with technical control in an accessible manner. The validation engine, DSL-Flowchart conversion, and plugin packaging processes work cohesively to ensure both usability and reliability.

In conclusion, this system provides a concrete solution to the problem identified at the outset: allowing domain experts to control workflow logic without technical intervention. It not only meets the research objectives but also lays the foundation for future enhancements, such as more advanced plugin logic, real-time IoT integration, and broader industry adaptation.

5. REFERENCES

[1] A. Bugaje, "Research of the workflow management," *International Journal of Advanced Research in Computer Science and Software Engineering,* vol. 6, no. 6, pp. 163-167, 2016.

[2] R. Goh, K. Robertson, R. Asokkumar and L. Anderson, "Mobile task management tool that improves workflow of an acute general surgical service," *ANZ Journal of Surgery,* vol. 90, no. 7-8, pp. 1281-1286, 2020.

[3] A. Seitz, K. Zuberbühler, A. Waser and A. Brodbeck, "ColorTree: a batch customization tool for phylogenetic trees," *Bioinformatics,* vol. 30, no. 23, pp. 3525-3526, 2014.

[4] J. Bergmann, M. Lindner and O. Urbann, "RoboStudio: A visual programming environment for educational robotics," in *2014 IEEE Global Engineering Education Conference (EDUCON)*, Istanbul, 2014.

[5] D. S. Robertson, , "A Modular Framework for Simulation Configuration," *Computers & Education,* vol. 56, no. 3, pp. 709-719, 2011.

[6] B. Shneiderman and K. Hyunmo, "Direct annotation: a drag-and-drop strategy for labeling photos," in *Proceedings of the IEEE International Conference on Information Visualization*, London, UK, 2000.

[7] E. Ganea, L. Barbulescu, E. Dumitrascu and M.-A. B, "Graphical Programming Environment for Creating Simulation Scenarios," *2023 IEEE Conference on Working in Education and Research (RoEduNet),* pp. 341-346, 2023.

[8] M. P. Denkers, A. van Deursen, T. van der Storm, E. Visser and A. Zaidman, "Taming complexity of industrial printing systems using a constraint-based DSL: An industrial experience report," *Software: Practice and Experience,* vol. 53, no. 10, pp. 2026-2064, 2023.

[9] J. DENKERS, Domain-Specific Languages for Digital Printing Systems, Netherlands, 2024.

[10] M. J. Jarrah and H. AI-Kilidar, "Code generation and model-based testing in context of OIL," *International Journal of Computer Applications,* vol. 69, no. 7, pp. 1-7, 2013.

[11] "Performance Impact of the Modular Architecture in the Incremental SGLR Parsing Algorithm," in *Springer*, Heidelberg, 2013.

[12] T. Vajk, G. Farser and P. McMinn, "Raising the Abstraction of Domain-Specific Model Translator Development," in *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, Boston, 2018.

[13] C. Jayawardena, I. Kuo, C. Datta, R. Q. Stafford, E. Broadbent and B. A. MacDonald , "Design, implementation and field tests of a socially assistive robot for the elderly: HealthBot Version 2," in *4th IEEE RAS & EMBS International Conference on Biomedical Robotics and Biomechatronics (BioRob)*, Rome, Italy, 2012.

[14] Milivoj Božić, Dušan Živkov, Ištvan Pap and Goran Miljković, "Scriptable graphical user interface engine for embedded platforms," in *21st Telecommunications Forum (TELFOR)*, Belgrade, Serbia, 2013.

[15] Nancy Staggers and David Kobus, "Comparing response time, errors, and satisfaction between text-based and graphical user interfaces during nursing order tasks," *Journal of the American Medical Informatics Association (JAMIA),* vol. 7, no. 2, pp. 164-176,

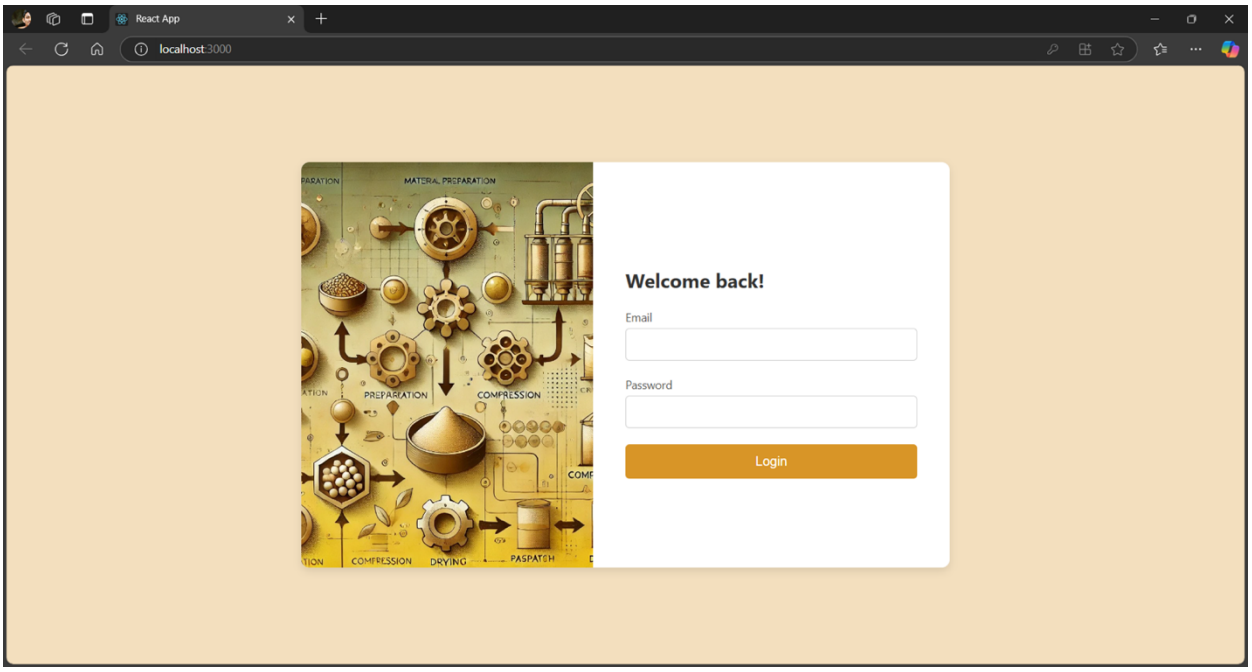
2000.

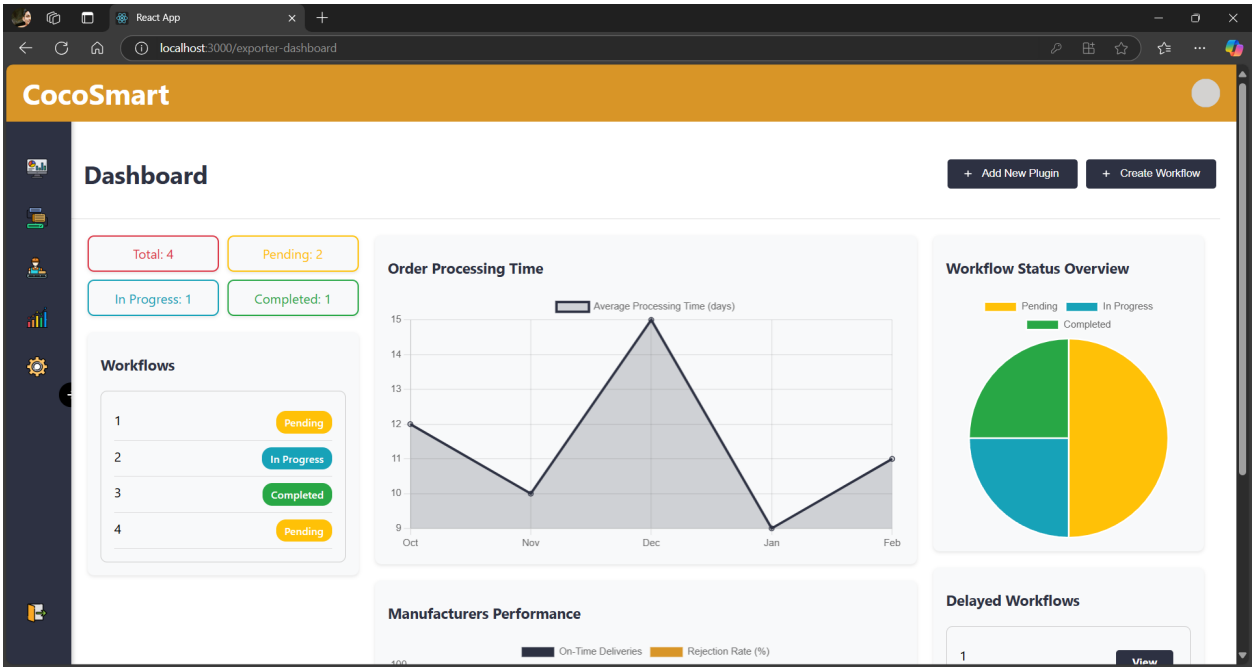
[16] Jan Koch, Max Reichardt and Karsten Berns, "Universal web interfaces for robot control frameworks," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Nice, France, 2008.

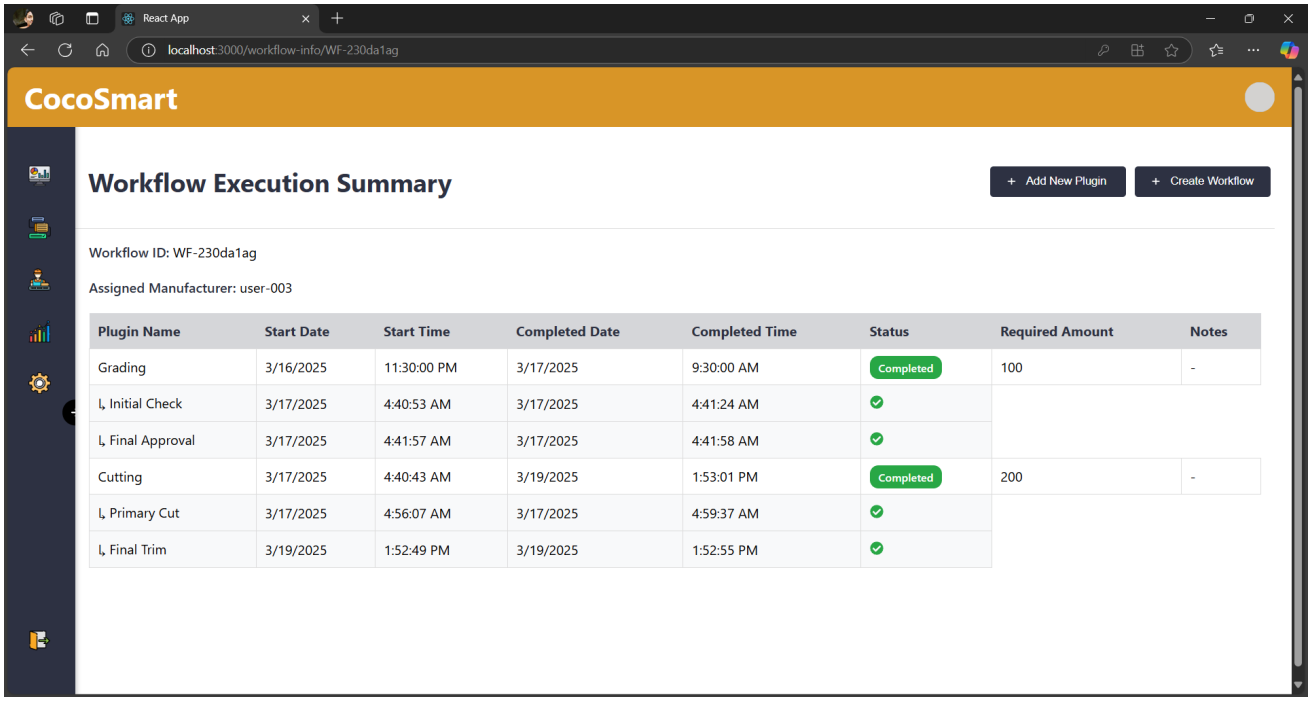
[17] H. Holz and D. Meurers, "Interaction styles in context: Comparing drag-and-drop, point-and-touch, and touch in a mobile spelling game," *International Journal of Human–Computer Interaction,* vol. 37, no. 9, pp. 835-850, 2021.

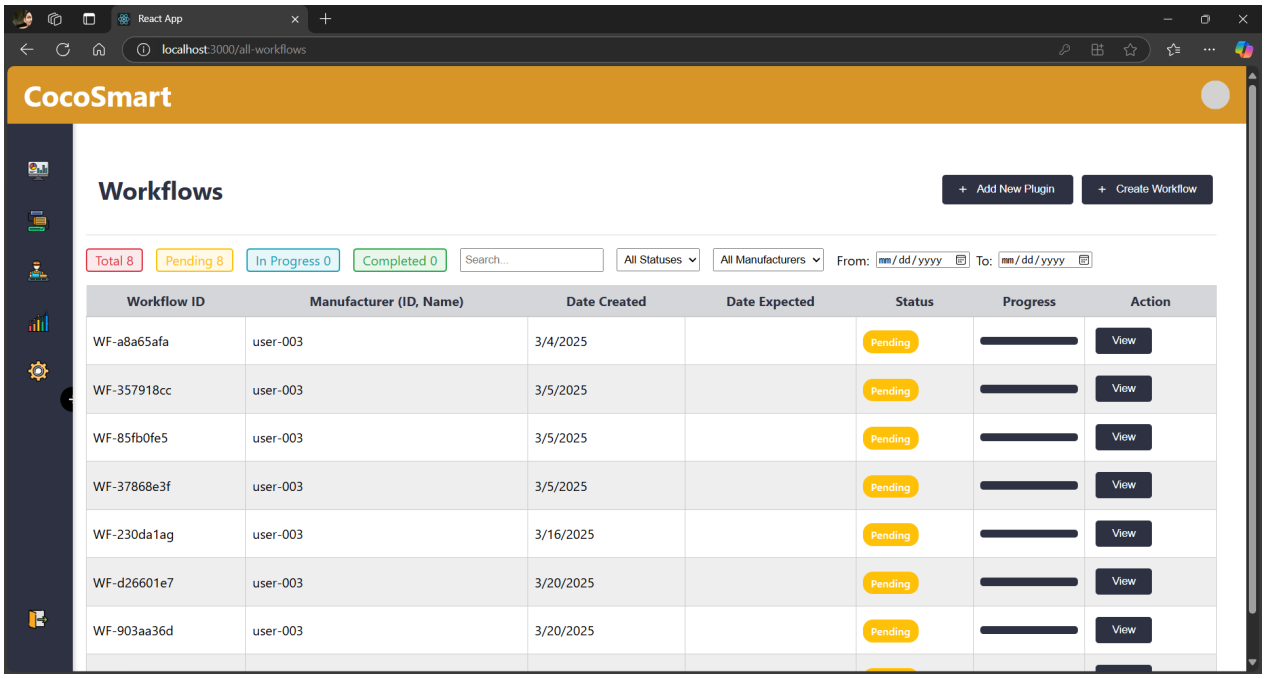
6. APPENDICES

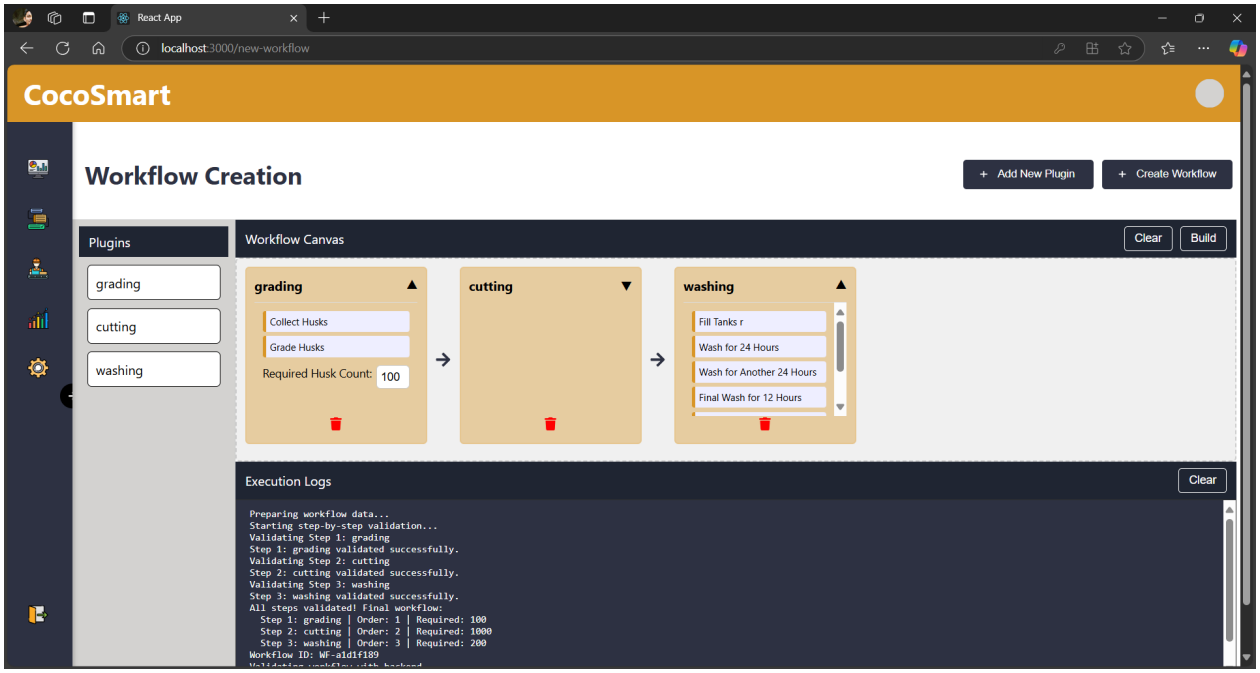
1.1 APPENDIX A: User Interfaces of the Workflow Customization Tool

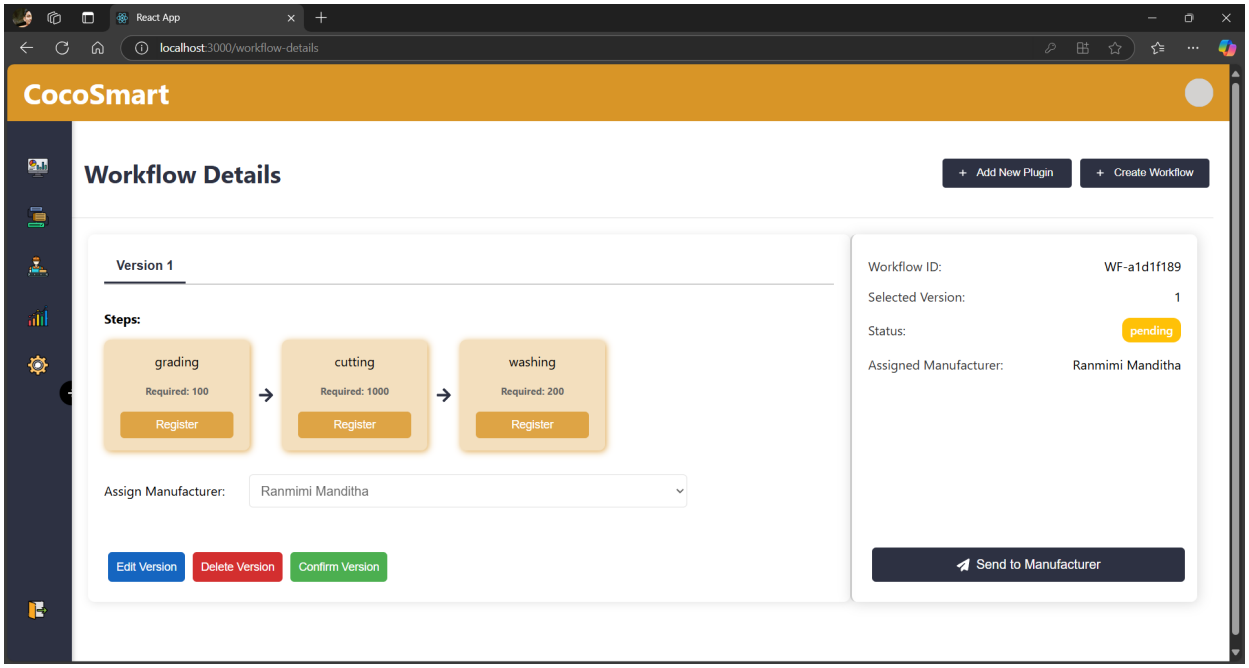


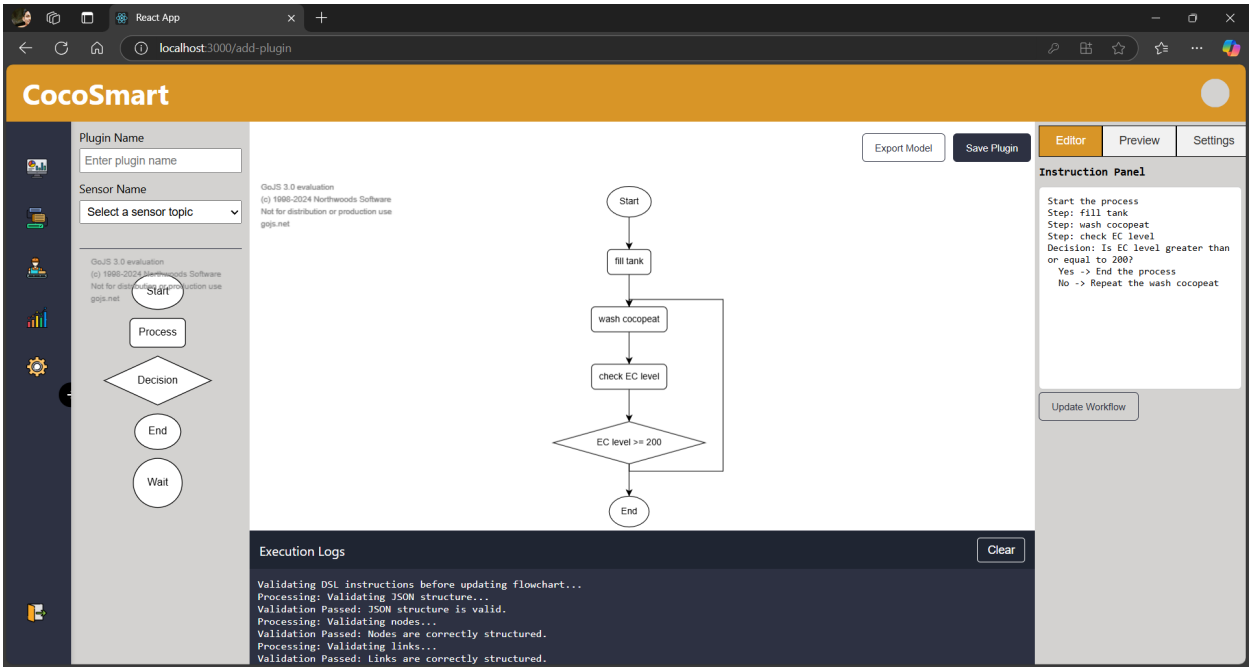


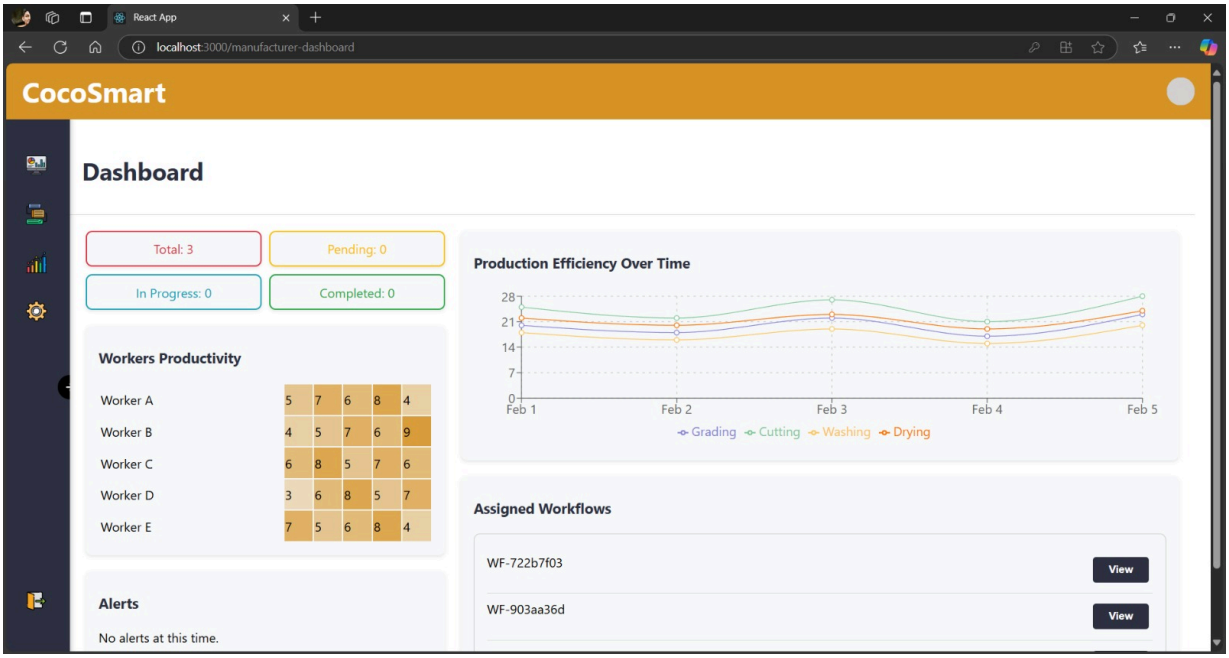

React App
localhost:3000/workflow-info/WF-230da1ag
CocoSmart
Workflow Execution Summary
+ Add New Plugin
+ Create Workflow
Workflow ID: WF-230da1ag
Assigned Manufacturer: user-003
Plugin Name | Start Date | Start Time | Completed Date | Completed Time | Status | Required Amount | Notes
Grading | 3/16/2025 | 11:30:00 PM | 3/17/2025 | 9:30:00 AM | Completed | 100 | -
↳ Initial Check | 3/17/2025 | 4:40:53 AM | 3/17/2025 | 4:41:24 AM
↳ Final Approval | 3/17/2025 | 4:41:57 AM | 3/17/2025 | 4:41:58 AM
Cutting | 3/17/2025 | 4:40:43 AM | 3/19/2025 | 1:53:01 PM | Completed | 200 | -
↳ Primary Cut | 3/17/2025 | 4:56:07 AM | 3/17/2025 | 4:59:37 AM
↳ Final Trim | 3/19/2025 | 1:52:49 PM | 3/19/2025 | 1:52:55 PM

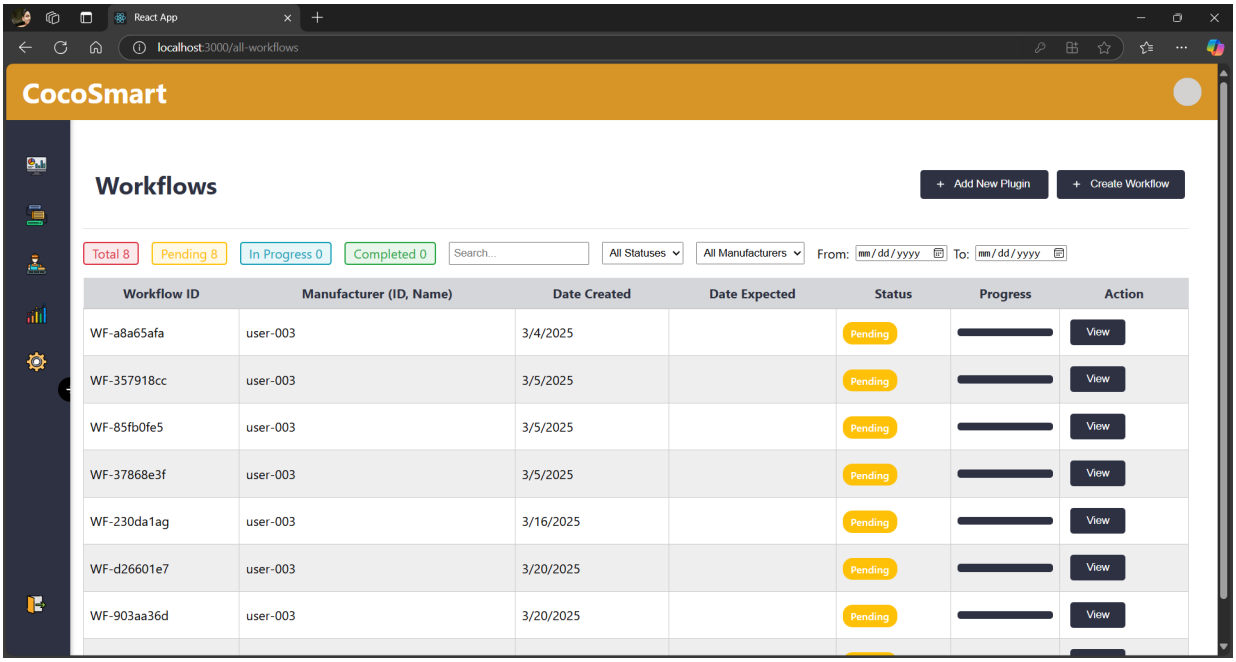

React App
localhost:3000/all-workflows
CocoSmart
Workflows
+ Add New Plugin
+ Create Workflow
Total 8
Pending 8
In Progress 0
Completed 0
Search...
All Statuses
All Manufacturers
From: mm/dd/yyyy To: mm/dd/yyyy
Workflow ID | Manufacturer (ID, Name) | Date Created | Date Expected | Status | Progress | Action
WF-a8a65afa | user-003 | 3/4/2025 | Pending | View
WF-357918cc | user-003 | 3/5/2025 | Pending | View
WF-85fb0fe5 | user-003 | 3/5/2025 | Pending | View
WF-37868e3f | user-003 | 3/5/2025 | Pending | View
WF-230da1ag | user-003 | 3/16/2025 | Pending | View
WF-d26601e7 | user-003 | 3/20/2025 | Pending | View
WF-903aa36d | user-003 | 3/20/2025 | Pending | View

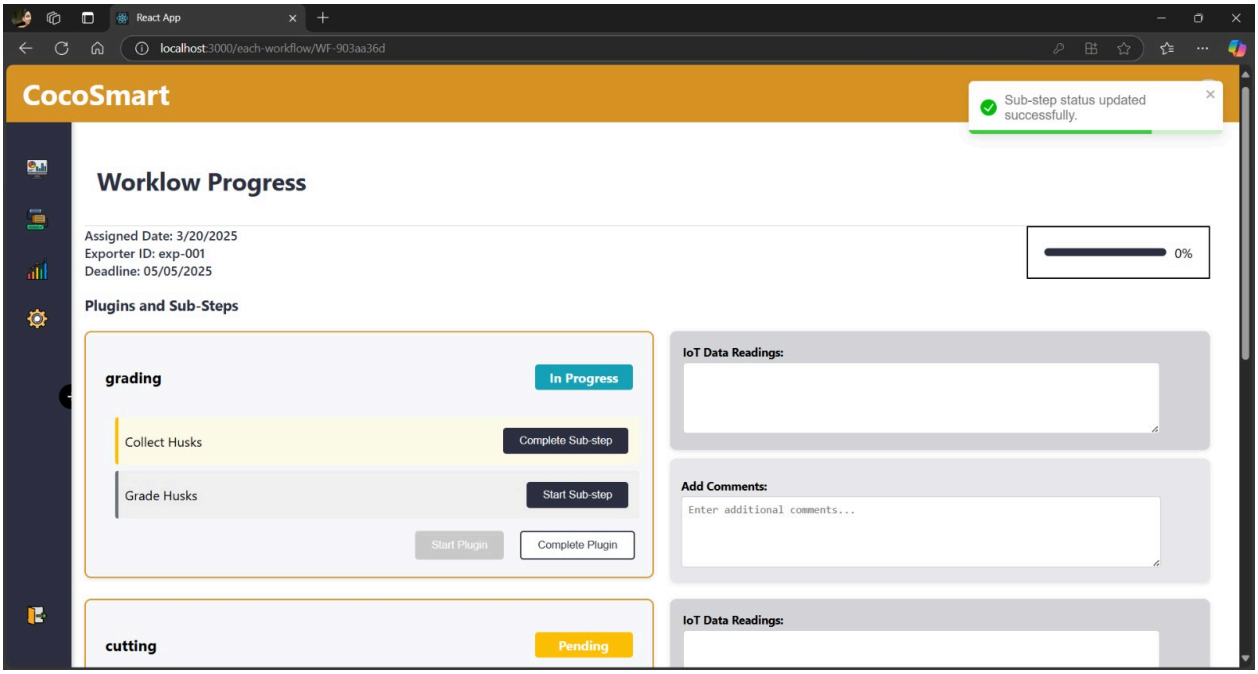

React App
localhost:3000/new-workflow
CocoSmart
Workflow Creation
+ Add New Plugin
+ Create Workflow
Plugins
grading
cutting
washing
Workflow Canvas
Clear
Build
grading
Collect Husks
Grade Husks
Required Husk Count: 100
cutting
washing
Fill Tanks r
Wash for 24 Hours
Wash for Another 24 Hours
Final Wash for 12 Hours
Execution Logs
Clear
Preparing workflow data...
Starting step-by-step validation...
Validating Step 1: grading
Step 1: grading validated successfully.
Validating Step 2: cutting
Step 2: cutting validated successfully.
Validating Step 3: washing
Step 3: washing validated successfully.
All steps validated! Final workflow:
Step 1: grading | Order: 1 | Required: 100
Step 2: cutting | Order: 2 | Required: 1000
Step 3: washing | Order: 3 | Required: 200
Workflow ID: WF-a1d1f189

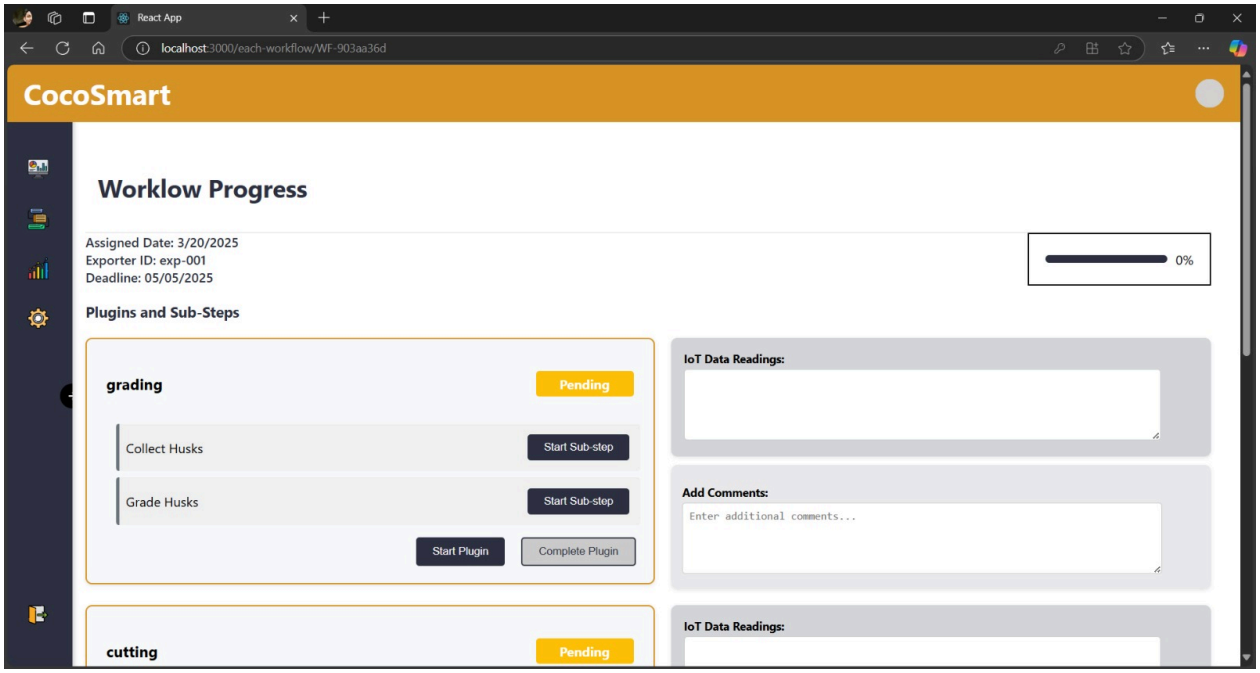

React App
localhost:3000/workflow-details
CocoSmart
Workflow Details
+ Add New Plugin
+ Create Workflow
Version 1
Steps:
grading
Required: 100
Register
cutting
Required: 1000
Register
washing
Required: 200
Register
Assign Manufacturer:
Ranmimi Manditha
Edit Version
Delete Version
Confirm Version
Workflow ID:
WF-a1d1f189
Selected Version:
1
Status:
pending
Assigned Manufacturer:
Ranmimi Manditha
Send to Manufacturer

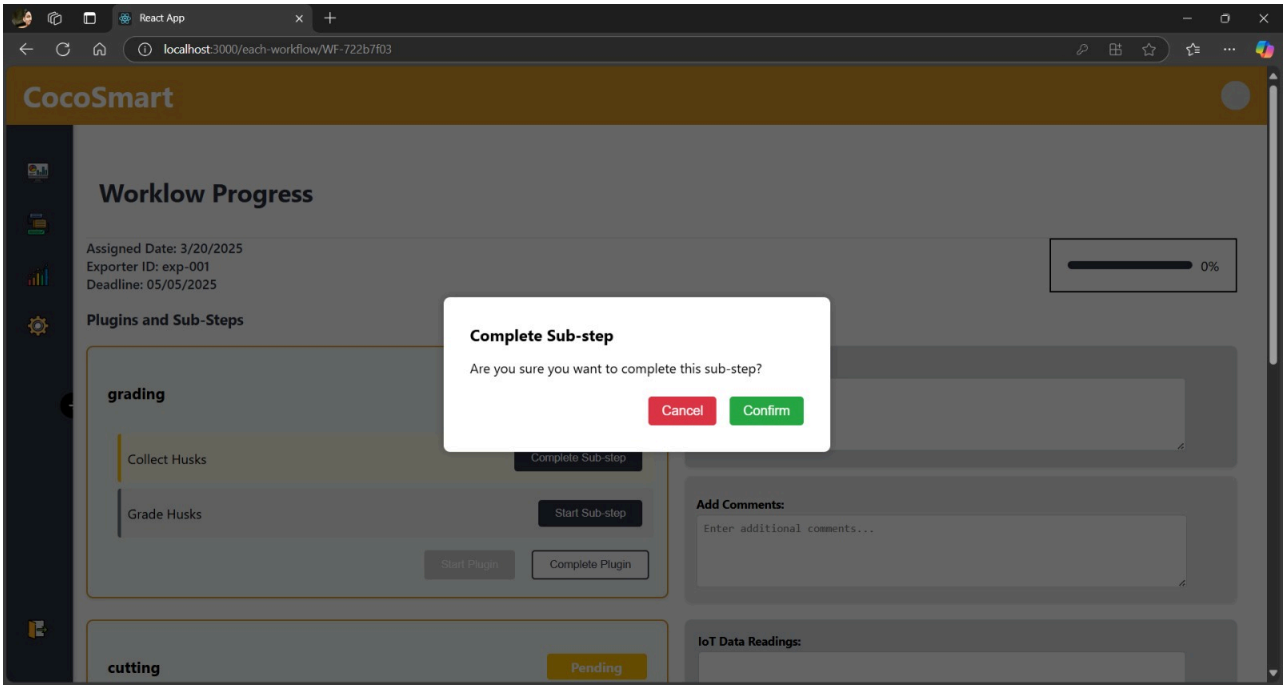

React App
localhost:3000/add-plugin
CocoSmart
Plugin Name
Enter plugin name
Sensor Name
Select a sensor topic
Start
Process
Decision
End
Wait
Export Model
Save Plugin
Start
fill tank
wash cocopeat
check EC level
EC level >= 200
End
Editor
Preview
Settings
Instruction Panel
Start the process
Step: fill tank
Step: wash cocopeat
Step: check EC level
Decision: Is EC level greater than or equal to 200?
Yes -> End the process
No -> Repeat the wash cocopeat
Update Workflow
Execution Logs
Clear
Validating DSL instructions before updating flowchart...
Processing: Validating JSON structure...
Validation Passed: JSON structure is valid.
Processing: Validating nodes...
Validation Passed: Nodes are correctly structured.
Processing: Validating links...
Validation Passed: Links are correctly structured.

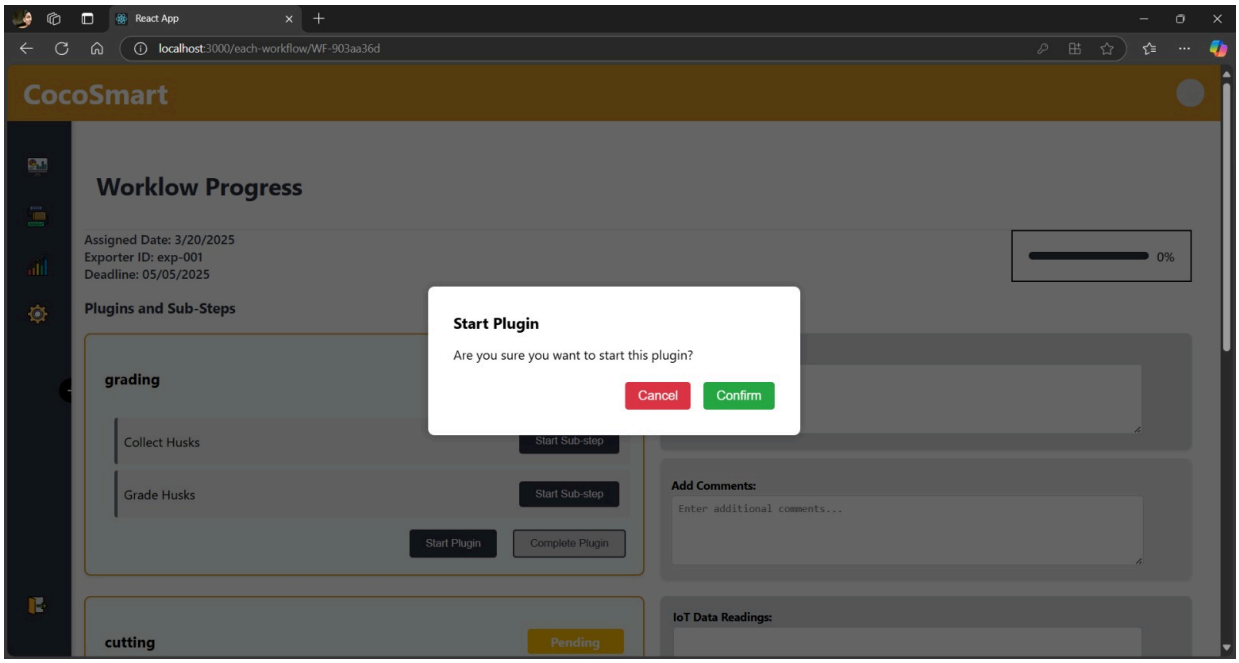

React App
localhost:3000/manufacturer-dashboard
CocoSmart
Dashboard
Total: 3
Pending: 0
In Progress: 0
Completed: 0
Workers Productivity
Worker A 5 7 6 8 4
Worker B 4 5 7 6 9
Worker C 6 8 5 7 6
Worker D 3 6 8 5 7
Worker E 7 5 6 8 4
Alerts
No alerts at this time.
Production Efficiency Over Time
28
21
14
7
0
Feb 1
Feb 2
Feb 3
Feb 4
Feb 5
Grading
Cutting
Washing
Drying
Assigned Workflows
WF-722b7f03
View
WF-903aa36d
View


React App
localhost:3000/all-workflows
CocoSmart
Workflows
+ Add New Plugin
+ Create Workflow
Total 8
Pending 8
In Progress 0
Completed 0
Search...
All Statuses
All Manufacturers
From: mm/dd/yyyy
To: mm/dd/yyyy
Workflow ID
Manufacturer (ID, Name)
Date Created
Date Expected
Status
Progress
Action
WF-a8a65afa
user-003
3/4/2025
Pending
View
WF-357918cc
user-003
3/5/2025
Pending
View
WF-85fb0fe5
user-003
3/5/2025
Pending
View
WF-37868e3f
user-003
3/5/2025
Pending
View
WF-230da1ag
user-003
3/16/2025
Pending
View
WF-d26601e7
user-003
3/20/2025
Pending
View
WF-903aa36d
user-003
3/20/2025
Pending
View

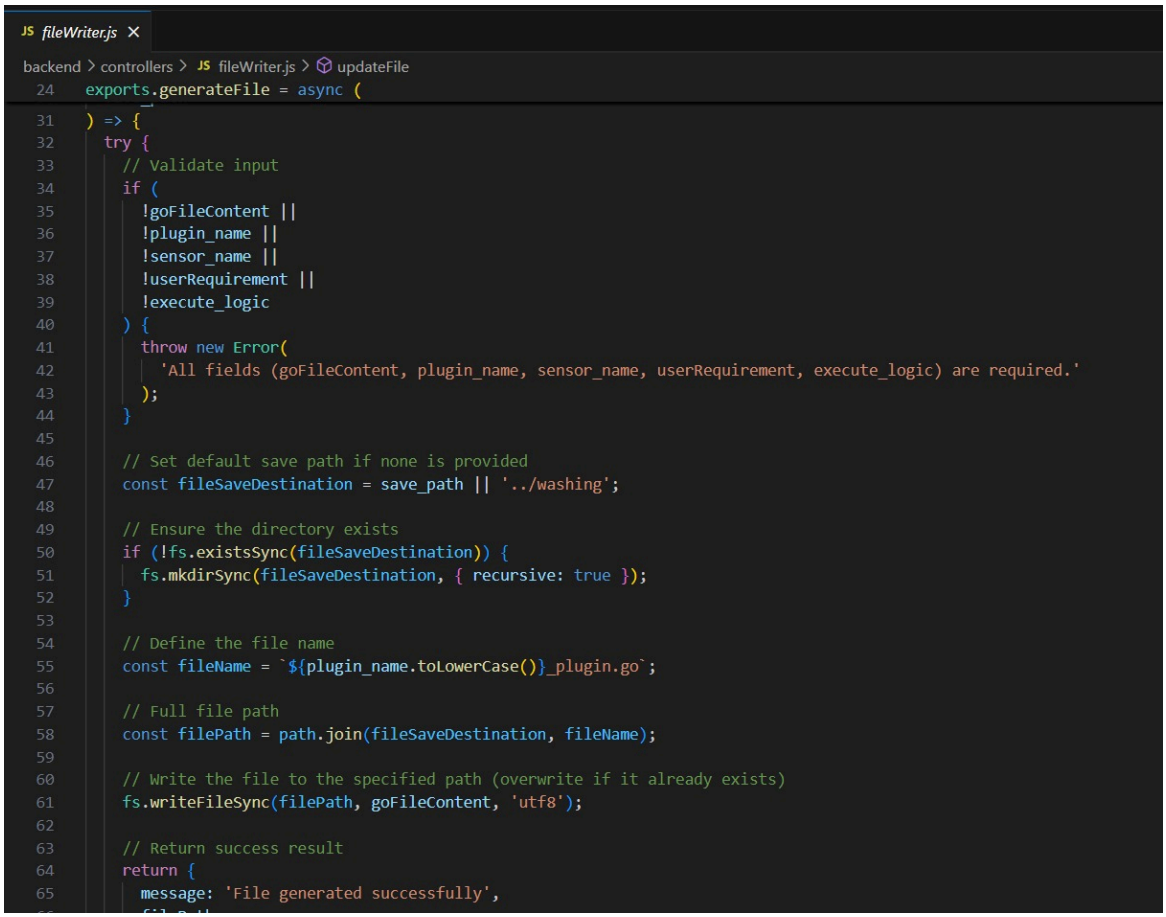

React App
localhost:3000/each-workflow/WF-903aa36d
CocoSmart
Sub-step status updated successfully.
Worklow Progress
Assigned Date: 3/20/2025
Exporter ID: exp-001
Deadline: 05/05/2025
0%
Plugins and Sub-Steps
grading
In Progress
Collect Husks
Complete Sub-step
Grade Husks
Start Sub-step
Start Plugin
Complete Plugin
IoT Data Readings:
Add Comments:
Enter additional comments...
IoT Data Readings:
cutting
Pending


React App
localhost:3000/each-workflow/WF-903aa36d
CocoSmart
Worklow Progress
Assigned Date: 3/20/2025
Exporter ID: exp-001
Deadline: 05/05/2025
0%
Plugins and Sub-Steps
grading
Pending
Collect Husks
Start Sub-step
Grade Husks
Start Sub-step
Start Plugin
Complete Plugin
IoT Data Readings:
Add Comments:
Enter additional comments...
cutting
Pending
IoT Data Readings:


React App
localhost:3000/each-workflow/WF-722b7f03
CocoSmart
Worklow Progress
Assigned Date: 3/20/2025
Exporter ID: exp-001
Deadline: 05/05/2025
0%
Plugins and Sub-Steps
Complete Sub-step
Are you sure you want to complete this sub-step?
Cancel
Confirm
grading
Collect Husks
Grade Husks
Start Sub-step
Complete Plugin
Add Comments:
Enter additional comments...
cutting
Pending
IoT Data Readings:


React App
localhost:3000/each-workflow/WF-903aa36d
CocoSmart
Worklow Progress
Assigned Date: 3/20/2025
Exporter ID: exp-001
Deadline: 05/05/2025
0%
Plugins and Sub-Steps
grading
Collect Husks
Start Sub-step
Grade Husks
Start Sub-step
Start Plugin
Complete Plugin
Start Plugin
Are you sure you want to start this plugin?
Cancel
Confirm
Add Comments:
Enter additional comments...
IoT Data Readings:
cutting
Pending

1.2 APPENDIX B: Code Snippets of the Plugin Creation Process



```
fileWriter.js
backend > controllers > fileWriter.js > updateFile
24  exports.generateFile = async (
31  ) => {
32    try {
33      // Validate input
34      if (
35        !goFileContent ||
36        !plugin_name ||
37        !sensor_name ||
38        !userRequirement ||
39        !execute_logic
40      ) {
41        throw new Error(
42          'All fields (goFileContent, plugin_name, sensor_name, userRequirement, execute_logic) are required.'
43        );
44      }
45
46      // Set default save path if none is provided
47      const fileSaveDestination = save_path || '../washing';
48
49      // Ensure the directory exists
50      if (!fs.existsSync(fileSaveDestination)) {
51        fs.mkdirSync(fileSaveDestination, { recursive: true });
52      }
53
54      // Define the file name
55      const fileName = `${plugin_name.toLowerCase()}_plugin.go`;
56
57      // Full file path
58      const filePath = path.join(fileSaveDestination, fileName);
59
60      // Write the file to the specified path (overwrite if it already exists)
61      fs.writeFileSync(filePath, goFileContent, 'utf8');
62
63      // Return success result
64      return {
65        message: 'File generated successfully',
```

pluginController.js

backend > controllers > pluginController.js > grpcFun > grpcFun

```
const NewPlugin = require('../models/Plugin');
const fs = require('fs');
const path = require('path');
const archiver = require('archiver');
const grpc = require('@grpc/grpc-js');
const protoLoader = require('@grpc/proto-loader');
const axios = require('axios');
const { updateFile } = require('../controllers/fileWriter');
const { generateFile } = require('../controllers/fileWriter');

// Save Plugin JSON
exports.savePlugin = async (req, res) => {
  try {
    const { plugin_name, nodes, links } = req.body;

    if (!plugin_name || !nodes || !links) {
      return res
        .status(400)
        .json({ success: false, message: 'Missing required fields' });
    }

    // Check if plugin already exists
    const existingPlugin = await NewPlugin.findOne({ plugin_name });

    if (existingPlugin) {
      return res
        .status(400)
        .json({ success: false, message: 'Plugin already exists' });
    }

    const plugin = new NewPlugin({ plugin_name, nodes, links });
    await plugin.save();
    res
      .status(201)
      .json({ success: true, message: 'Plugin saved successfully' });
  } catch (error) {
    res.status(500).json({ success: false, message: error.message });
```

pluginController.js

backend > controllers > pluginController.js > grpcFun > grpcFun

```
// Fetch Plugin JSON by Plugin Name
exports.getPlugin = async (req, res) => {
  try {
    const plugin = await NewPlugin.findOne({
      plugin_name: req.params.plugin_name,
    });

    if (!plugin) {
      return res
        .status(404)
        .json({ success: false, message: 'Plugin not found' });
    }

    res.json({ success: true, data: plugin });
  } catch (error) {
    res.status(500).json({ success: false, message: error.message });
  }
};

// Load the protobuf
const PROTO_PATH = path.join(__dirname, '../grpc-node-server/plugin.proto');
const packageDefinition = protoLoader.loadSync(PROTO_PATH, {
  keepCase: true,
  longs: String,
  enums: String,
  defaults: true,
  oneofs: true,
});

const pluginProto = grpc.loadPackageDefinition(packageDefinition).plugin;

// gRPC Client setup
const client = new pluginProto.MainService(
  '0.0.0.0:50051',
  grpc.credentials.createInsecure()
);
```

pluginController.js

backend > controllers > pluginController.js > processAll > processAll > folderPath

```
110  exports.processAll = async (req, res) => {
111    try {
112      const {
113        updateContent,
114        goFileContent,
115        plugin_name,
116        sensor_name,
117        userRequirement,
118        execute_logic,
119        save_path,
120      } = req.body;
121
122      // Step 1: Update the first file
123      await updateFile(updateContent, `../washing/${plugin_name}.dockerfile`);
124
125      // Step 2: Generate the second file
126      const folderPath = await generateFile(
127        goFileContent,
128        plugin_name,
129        sensor_name,
130        userRequirement,
131        execute_logic,
132        save_path
133      );
134
135      // Step 3: Zip the folder
136      const pluginFolderPath = path.resolve('../', 'washing');
137      const outputZipPath = path.resolve('../', 'washing.zip');
138
139      zipFolder(pluginFolderPath, outputZipPath)
140        .then(() => {
141          console.log('Folder successfully zipped!');
142          // path to the file  to upload
143          uploadFile('../washing.zip');
144          // Step 4: Upload the zipped folder via gRPC
145          const zipFileData = fs.readFileSync(outputZipPath);
```

pluginController.js

backend > controllers > pluginController.js > processAll > processAll > folderPath

```
110  exports.processAll = async (req, res) => {
139      zipFolder(pluginFolderPath, outputZipPath)
140        .then(() => {
141          console.log('Folder successfully zipped!');
142          // path to the file  to upload
143          uploadFile('../washing.zip');
144          // Step 4: Upload the zipped folder via gRPC
145          const zipFileData = fs.readFileSync(outputZipPath);
146          const grpcRequest = {
147            filename: 'washing.zip',
148            filedata: zipFileData,
149          };
150
151          client.UploadFile(grpcRequest, (err, response) => {
152            if (err) {
153              console.error('Error uploading file via gRPC:', err);
154              return res
155                .status(500)
156                .json({ message: 'Error uploading file via gRPC', error: err });
157            }
158
159            res.status(200).json({
160              message: 'All steps completed successfully!',
161              update: 'File updated successfully',
162              generate: 'File generated successfully',
163              zipUpload: response.message,
164            });
165          });
166        })
167        .catch((err) => console.error('Error zipping folder:', err));
168    } catch (err) {
169      console.error('Error processing all steps:', err);
170      res
171        .status(500)
172        .json({ message: 'Error processing all steps', error: err.message });
173    }
174  };
```

pluginController.js

backend > controllers > pluginController.js > processAll > processAll

```
176  // Folder Zip method
177  function zipFolder(folderPath, outputZipPath) {
178    return new Promise((resolve, reject) => {
179      // Create a file to stream the archive data to.
180      const output = fs.createWriteStream(outputZipPath);
181      const archive = archiver('zip', { zlib: { level: 9 } }); // Best compression level
182
183      // Listen for events
184      output.on('close', () => {
185        console.log(`Zipped ${archive.pointer()} total bytes`);
186        resolve();
187      });
188
189      archive.on('error', (err) => reject(err));
190
191      // Pipe archive data to the output file
192      archive.pipe(output);
193
194      // Append the folder to the archive
195      archive.directory(folderPath, false); // `false` prevents nesting in a subfolder
196
197      // Finalize the archive
198      archive.finalize();
199    });
200  }
201
202  function uploadFile(filePath) {
203    try {
204      const filename = filePath.split('/').pop();
205      const fileData = fs.readFileSync(filePath);
206
207      const request = {
208        filename: filename,
209        filedata: fileData,
210      };
211
212      newPluginClient.NewPluginCreate(request, (err, response) => {
```

fileWriter.js

backend > controllers > fileWriter.js > updateFile

```
const fs = require('fs');
const Port = require('../models/Port');
const path = require('path');

// Endpoint to update the file
exports.updateFile = async (content, filePath) => {
  return new Promise((resolve, reject) => {
    if (!content) {
      return reject(new Error('Content is required'));
    }

    // Write the updated content to the file
    fs.writeFile(filePath, content, (err) => {
      if (err) {
        console.error(err);
        reject(new Error('Error updating the file'));
      } else {
        resolve({ message: 'File updated successfully!' });
      }
    });
  });
};

exports.generateFile = async (
  goFileContent,
  plugin_name,
  sensor_name,
  userRequirement,
  execute_logic,
  save_path
) => {
  try {
    // Validate input
    if (
      !goFileContent ||
      !plugin_name ||
      !sensor_name ||
```

pluginController.js

backend > controllers > pluginController.js > processAll > processAll

```
177  function zipFolder(folderPath, outputZipPath) {
178    return new Promise((resolve, reject) => {
189      archive.on('error', (err) => reject(err));
190
191      // Pipe archive data to the output file
192      archive.pipe(output);
193
194      // Append the folder to the archive
195      archive.directory(folderPath, false); // `false` prevents nesting in a subfolder
196
197      // Finalize the archive
198      archive.finalize();
199    });
200  }
201
202  function uploadFile(filePath) {
203    try {
204      const filename = filePath.split('/').pop();
205      const fileData = fs.readFileSync(filePath);
206
207      const request = {
208        filename: filename,
209        filedata: fileData,
210      };
211
212      newPluginClient.NewPluginCreate(request, (err, response) => {
213        if (err) {
214          console.error('Error uploading file:', err);
215          return;
216        }
217
218        console.log('Server response:', response.message);
219      });
220    } catch (err) {
221      console.error('Failed to read the file:', err.message);
222    }
223  }
```