

**ENHANCING TRANSPARENCY IN THE COCONUT SUPPLY CHAIN THROUGH A
SOFTWARE ENGINEERING APPROACH**

Group ID: 24-25J-313	
IT21308284	Vithanage H.D
IT21291678	Dehipola H.M.S.N
IT21289484	Manditha K.D.R
IT21576966	Weedagamaarachchi K.S

B.Sc. (Hons) Degree in Information Technology Specialized in Software Engineering

Department of Information Technology

Sri Lanka Institute of Information Technology
Sri Lanka

April 2025

**ENHANCING TRANSPARENCY IN THE COCONUT SUPPLY CHAIN THROUGH A
SOFTWARE ENGINEERING APPROACH**

Group ID: 24-25J-313	
IT21308284	Vithanage H.D
IT21291678	Dehipola H.M.S.N
IT21289484	Manditha K.D.R
IT21576966	Weedagamaarachchi K.S

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

Department of Information Technology

Sri Lanka Institute of Information Technology
Sri Lanka


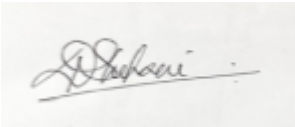


April 2025

DECLARATION

Declaration of the Candidates

“I declare that this is my own work and this dissertation1 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, we 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	Date
Vithanage H.D	IT21308284		2025/04/11
Dehipola H. M. S. N	IT21291678		2025/04/11
K.D.R Manditha	IT21289484		2025/04/11
Weedagamaarachchi K.S	IT21576966		2025/04/11

Declaration of the Supervisor

The above candidate has carried out research for the bachelor’s degree Dissertation under my supervision.

Supervisor: Mr.Vishan Jayasinghearachchi	Signature:	Date: 2025/04/11
Co-Supervisor: Mr. Jeewaka Perera	Signature:	Date:2025/04/11

ABSTRACT

The increasing global demand for sustainable agricultural practices has highlighted the need for transparent and adaptable supply chain systems particularly in industries such as coconut peat production, where processing workflows are complex and often manually managed. This research presents a modular, plugin-based software architecture designed to enhance transparency, automation, and traceability in the coconut peat supply chain.

The proposed system integrates four core technological pillars: a microkernel-inspired architecture for scalable plugin orchestration; a low-code workflow customization tool powered by a domain-specific language (DSL); real-time IoT-based automation using ESP32 and MQTT; and blockchain-backed traceability through optimized Ethereum smart contracts. Each manufacturing process—such as husk grading, washing, drying, and packaging is modeled as a dynamically managed plugin executed within isolated containers orchestrated by K3s. Communication between components is facilitated by gRPC for high-throughput and fault-tolerant interaction.

A visual drag-and-drop workflow editor enables non-technical users to define and deploy custom workflows, which are converted into Go-based configuration files and interpreted by the core system. Simultaneously, IoT sensors provide real-time environmental data to trigger workflow transitions, while smart contracts log key events on the blockchain to ensure data integrity and auditability. Gas optimization techniques were applied to improve smart contract efficiency, resulting in a 19% reduction in deployment costs and 14% savings during execution.

The system was validated through component-level and integrated testing, demonstrating its effectiveness in real-time responsiveness, plugin stability, sensor feedback accuracy, and blockchain logging. This work introduces a transformative model for digitizing agricultural supply chains in Sri Lanka and beyond.

Keywords: Plugin-based Architecture, Microkernel System, Workflow Customization, Domain-Specific Language (DSL), IoT Integration, Blockchain Traceability, Smart Contract Optimization

ACKNOWLEDGEMENT

First and foremost, we would like to express my sincere gratitude to my supervisor, Mr. Vishan Jayasinghearachchi, for his invaluable guidance, encouragement, and unwavering support throughout the course of this undergraduate research project. His expertise, constructive feedback, and commitment to academic excellence have been instrumental in shaping the direction and depth of this study.

we are also deeply thankful to my co-supervisor, Mr. Jeewaka Perera, for his continuous support, timely insights, and readiness to assist whenever challenges arose. His contribution significantly enriched the quality of this work, and his mentorship was a vital source of motivation throughout the project journey.

This project was the result of a collaborative effort, and we would like to extend my heartfelt appreciation to my team members for their dedication, hard work, and seamless cooperation. Their persistence and enthusiasm helped overcome numerous obstacles, and their contributions were key to achieving our shared objectives.

Additionally, this research ventured into several interdisciplinary domains that extended beyond our initial areas of expertise. We are grateful to the individuals, communities, and technical resources that helped bridge these knowledge gaps. Their insights not only supported the development of this project but also contributed to my personal and professional growth.

Finally, we would like to express my heartfelt thanks to my family, friends, and colleagues for their constant encouragement, patience, and emotional support. Their belief in my abilities was a continuous source of strength, especially during the most demanding phases of this research.

TABLE OF CONTENTS

DECLARTION..... i

 Declaration of the Candidates i

 Declaration of the Supervisor i

ABSTRACT ii

ACKNOWLEDGEMENT..... iii

LIST OF FIGURES vi

LIST OF TABLES vii

LIST OF ABBREVAITION viii

INTRODUCTION..... 1

 General Introduction 1

 Reimagining Supply Chains with Plugin-Based Architecture 1

 Workflow Customization for Non-Technical Users 1

 IoT-Powered Automation and Sensor Feedback 1

 Blockchain for Transparency, Trust, and Cost Efficiency 2

 Towards a Scalable, Transparent, and Intelligent Supply Chain 2

 Literature Survey 2

 Plugin-Based System Architecture in Supply Chain Management 2

 Workflow Customization Tool and Domain-Specific Language (DSL) 3

 Blockchain-Backed Transparency and Gas-Efficient Smart Contracts 3

 IoT-Driven Automation for Real-Time Quality Assurance 3

 Research Gap 4

 Lack of Domain-Specific Workflow Customization Tools 4

 Inadequate Runtime Plugin Management in Distributed Architectures 4

 Missing Integration Between IoT Sensors and Workflow Engines 4

 Lack of Efficient Blockchain-Integrated Logging Systems 4

 Unexplored Relationship Between Code Complexity and Smart Contract Gas Costs 5

 Research Problem 5

 Core Issue..... 5

 Specific Problem Dimensions 5

 Research Question..... 6

 Sub-Questions 6

 Research Objectives 6

 Research Aim 6

 General Objective..... 6

 Specific Objectives..... 6

METHODOLOGY 8

 Methodology 8

 Overview of Methodological Approach..... 8

Individual Component Methodologies.....	8
Technology Stack Summary	10
Workflow Customization Lifecycle	11
Validation and Iterative Feedback.....	12
Commercialization.....	13
Testing & Implementation	15
RESULTS & DISCUSSION	19
Results.....	19
Plugin Execution and System Performance	19
Workflow definition and validation	19
Image processing performance for husk classification	19
Smart contract gas optimization results	19
Research Findings.....	19
Fulfillment of Objectives	19
Usability and workflow accessibility	19
Real time responsiveness and data synchronization	20
Solidity optimization outcomes.....	20
Workflow logic and DSL validation.....	20
Discussion	20
Architectural and Design Impact	20
Cost-Effective classification strategy.....	20
Blockchain integration and Gas optimization	20
Generalizability across domains	20
System limitations and recommendations.....	20
CONCLUSION	21
Research Accomplishments	21
Broader Impact and Innovation.....	21
Limitations and Future Work	22
Final Remarks	22
Summary of Each Student’s contribution	23
REFERENCES.....	25
Appendices	26
APPENDIX A: User Interfaces of the Workflow Customization Tool	26
APPENDIX B: Code Snippets of the Plugin Creation Process.....	32
APPENDIX B: Code snippet of the core system	37

LIST OF FIGURES

Figure No.	Title	Page No.
Figure 1	System Architecture Diagram	7
Figure 2	Rancher UI with All the Pods Working	15
Figure 3	MQTT Topic Logs #1	16
Figure 4	MQTT Topic Logs #2	16
Figure 5	MQTT Topic Logs #3	16
Figure 6	DSL Preview and Generated Plugin Structure	17
Figure 7	Smart Contract Deployment Gas Profile Comparison	17
Figure 8	Grafana Dashboard: CPU Usage Graph	17

LIST OF TABLES

Table No.	Title	Page No.
Table 1	Frontend and Workflow Customization Technology Stack	10
Table 2	Backend Services and Plugin Logic Technology Stack	10
Table 3	Smart Contract and Blockchain Technology Stack	11
Table 4	IoT Sensor Integration Technology Stack	11
Table 5	Deployment and Containerization Technology Stack	11
Table 6	Security and Performance Tools Technology Stack	11
Table 7	Component-Level Validation	13
Table 8	Final Validation Metrics and Targets	14
Table 9	Unique Value Propositions of the Proposed System	13
Table 10	Target Users and Use Cases	13
Table 11	Competitive Advantage: Comparison Between Proposed System and Existing ERP Solutions	14
Table 12	Testing Approaches Used for System Evaluation	15
Table 13	Implementation Challenges and Resolutions	17
Table 14	Final Outcome and Performance Evaluation	18

LIST OF ABBREVAITION

Abbreviation	Full Form
API	Application Programming Interface
ATAM	Architecture Tradeoff Analysis Method
B2B	Business-to-Business
CBAM	Cost Benefit Analysis Method
CPU	Central Processing Unit
CSR	Customer Service Representative
DSL	Domain-Specific Language
EC	Electrical Conductivity
EIP	Ethereum Improvement Proposal
ERP	Enterprise Resource Planning
ESP32	Embedded Serial Processor 32-bit Microcontroller
gRPC	Google Remote Procedure Call
GUI	Graphical User Interface
HPA	Horizontal Pod Autoscaler
IoT	Internet of Things
K3s	Lightweight Kubernetes Distribution for Edge Computing
MQTT	Message Queuing Telemetry Transport
Node-RED	Flow-Based Development Tool for Visual Programming
UI	User Interface
VVM	Visual Validation Mechanism (if applicable based on context)
WDA	Workflow Definition Application
YOLO	You Only Look Once (Object Detection Algorithm)

INTRODUCTION

General Introduction

In an era increasingly shaped by climate change, environmental concerns, and growing demands for ethical sourcing, the global shift toward green and sustainable farming solutions has gained remarkable momentum. Within this context, organic substrates like coco peat a by-product of coconut husk processing have emerged as essential components in sustainable agriculture, horticulture, and industrial absorbents. As a globally competitive export, coco peat not only holds immense market potential but also introduces pressing challenges related to quality assurance, production transparency, and workflow efficiency.

Sri Lanka, as one of the top coconut-producing nations in the world, plays a pivotal role in the global coco peat supply chain. However, many local manufacturers and exporters still operate using manually defined or semi-automated workflows, lacking integration with real-time monitoring tools or transparent traceability mechanisms. As demand from international markets rises not only in terms of quantity but also in accountability and sustainability transforming the supply chain with intelligent, traceable, and customizable automation tools has become imperative.

To address these critical gaps, this research proposes a **novel modularized system architecture** that brings together four technological pillars:

1. A **microkernel-inspired plugin architecture** for modular execution,
2. A **visual workflow customization platform** using domain-specific logic,
3. **IoT-based real-time data acquisition** for automation and responsiveness,
4. And **blockchain-supported smart contracts** to provide immutable traceability and trust.

These components together form a **next-generation supply chain management system** purpose-built for the unique conditions and complexities of coco peat production in Sri Lanka.

Reimagining Supply Chains with Plugin-Based Architecture

At the heart of the system lies a **plugin-driven architectural model**, inspired by microkernel principles. In this paradigm, each stage of the manufacturing process such as **husk grading, cutting, washing, and drying** is abstracted into its own **dynamically managed plugin**. These plugins are **containerized** and orchestrated by **K3s**, a lightweight Kubernetes framework optimized for edge computing environments with limited resources.

The **core execution engine**, developed in Go, manages all inter-process communication, plugin registration, health monitoring, and event handling. Using **gRPC**, a fast, strongly typed communication protocol, the core enables low-latency, high-throughput interaction between plugins and system services. This architecture ensures **runtime flexibility**, **plugin fault isolation**, and **high system availability**, even when new plugins are introduced or when certain stages fail and require recovery.

Workflow Customization for Non-Technical Users

A major innovation in the proposed system is the **Workflow Customization Tool**, designed to democratize process configuration for non-technical users such as factory supervisors and exporters. Built with a drag-and-drop visual interface using React and GoJS, the tool allows users to **design custom workflows** by connecting plugins as flowchart components.

Behind this interface lies a purpose-built **Domain-Specific Language (DSL)** tailored to the coconut peat industry. The DSL enables users to define conditional logic, plugin parameters, and sequential task execution in a clear and intuitive manner. Advanced users may also opt for a **code-based DSL editor** to define more complex workflows. Regardless of the interface used, the system translates the visual or textual DSL into Go-based plugin configuration files, which are **instantly registered and deployed by the core system** without any system downtime or developer intervention.

This capability empowers stakeholders to **rapidly respond to customer-specific requirements** (e.g., double washing for low EC levels or customized drying durations), significantly enhancing flexibility and operational responsiveness.

IoT-Powered Automation and Sensor Feedback

The system's automation layer is driven by **IoT devices**, which provide **real-time environmental and product-specific data** to the core system. **ESP32 and ESP32-CAM modules** are used to collect metrics such as electrical conductivity (EC), moisture levels, and even image-based classification of coconut husks.

These devices publish sensor readings to a central MQTT broker (**HiveMQ**), from which the core system subscribes to relevant topics and routes the data to specific plugins. For example:

- A camera module performs **color-based classification** of husks into *qualified*, *accepted*, or *disqualified*.
- EC sensors monitor water quality during the washing stage, triggering **re-wash cycles** if thresholds are exceeded.

This integration of IoT technology enables **autonomous decision-making**, reduces human error, and aligns the system with modern industrial demands for **efficiency, consistency, and traceable automation**.

Blockchain for Transparency, Trust, and Cost Efficiency

Recognizing the increasing demand for **supply chain traceability and auditability**, the system incorporates **blockchain technology** using **Ethereum smart contracts**. Key events such as shipment approvals, EC-level validations, and grading decisions are logged onto the blockchain to ensure that records are **immutable, verifiable, and tamper-proof**.

To maintain efficiency and reduce operational costs, **gas-optimization strategies** were applied to the smart contract design. Using techniques like **variable packing, mapping simplification, and immutable declarations**, the implementation optimized widely-used patterns such as:

- **Factory Pattern** for contract instantiation,
- **Registry Pattern** for stakeholder role management,
- **State Machine Pattern** for controlled transitions in shipment lifecycles.

As a result, the system achieved significant **reductions in both deployment and execution gas costs**, making blockchain integration practical and scalable even in high-frequency transaction environments.

Towards a Scalable, Transparent, and Intelligent Supply Chain

Together, these four foundational components form a **robust and scalable ecosystem** that transforms how supply chain operations in the coconut peat industry are managed. The architecture:

- Offers **real-time visibility** into every step of the production process,
- Enables **dynamic reconfiguration** of workflows based on business needs or sensor input,
- Provides **cryptographically secure audit trails** to build trust with international clients and regulators,
- And **reduces dependency on manual oversight** through sensor-triggered automation.

By combining modern software engineering practices with **domain-specific requirements**, this project introduces a **transformative model** for supply chain management that can be extended to other agricultural sectors. It positions Sri Lanka's coco peat industry to not only meet global demand but to lead it through innovation, transparency, and sustainable practices.

Literature Survey

Plugin-Based System Architecture in Supply Chain Management

Modern supply chain systems demand flexibility, real-time responsiveness, and fault-tolerant architecture to accommodate increasing complexity and customer-specific customization. Traditional monolithic and even microservices-based ERP systems often fail to address these needs due to limited runtime adaptability and lack of plugin-level fault isolation. To overcome these limitations, this project adopts a plugin-based microkernel-inspired architecture tailored to the coconut peat industry.

The microkernel design principle, which separates core functionalities from domain-specific logic through modular plugins, has been shown to enhance system reliability and extensibility. Previous studies [1] emphasizes that minimizing the core's responsibilities while offloading domain-specific services to plugins significantly improves availability. Similar architectural approaches have been explored in distributed systems [2] and industrial IoT environments [3], demonstrating the feasibility of such designs under real-time and distributed conditions.

The system extends this principle by developing a Go-based core that supports dynamic plugin loading, execution, and recovery using Docker and K3s orchestration. The use of gRPC enables language-agnostic and low-latency communication between plugins and the core, addressing challenges noted [1] and [2] regarding inter-process communication (IPC) overhead. Additionally, the architecture supports runtime plugin registration triggered by both user inputs and sensor events, aligning with needs highlighted in microkernel applications such as HealthBot [4] and DROPS framework [5].

While microkernel architectures have found success in operating systems and robotics, their adoption in agricultural supply chain contexts—such as coconut peat processing has been limited. This project fills that gap by integrating real-time sensor-triggered execution, plugin versioning, container-based isolation, and system health monitoring, thereby delivering a scalable, resilient, and domain-adaptive solution.

Workflow Customization Tool and Domain-Specific Language (DSL)

Visual workflow customization has emerged as a critical need in industries requiring frequent, domain-specific process changes managed by non-technical users. Traditional workflow management systems often cater to general enterprise environments or require programming expertise, making them inaccessible to operational staff such as manufacturers and exporters. In the coco-peat industry, workflows are dynamic and influenced by variables like client specifications and environmental conditions, demanding tools that allow rapid and autonomous adjustments. However, most existing tools [6], [7] lack the plugin extensibility, domain-specific logic, and intuitive user interfaces required for real-time operational control.

This tool builds based on this concept by integrating a custom-built DSL specifically designed for coconut peat processing workflows. Users can define workflows either through a drag-and-drop visual interface or a dedicated DSL code editor. This DSL supports the expression of plugin sequencing, conditional execution, and parameter configuration. Once a workflow is defined, it is automatically converted into Go-based configuration files, which are interpreted by the core system.

The use of DSLs for runtime logic control has been supported in systems like HealthBot [8] and software-defined industrial automation platforms. However, these implementations lack integration with real-time plugin orchestration and blockchain-backed traceability. Our system overcomes these limitations by combining DSL-based configuration with dynamic container deployment, allowing real-time updates to workflows without system restarts.

Additionally, the visual workflow editor improves usability for non-technical domain-specific users such as exporters when adding a new plugin. The studies based on [9] and [10] showed the possible side of using such different methods to handle plugin management which is using the visual editor and DSL instructions editor. The DSL was designed to capture all essential operational parameters such as threshold values and batch quantities—transforming domain-specific logic into executable plugin instructions.

Blockchain-Backed Transparency and Gas-Efficient Smart Contracts

Blockchain has emerged as a transformative technology in supply chain management due to its inherent properties of decentralization, immutability, and transparency. Ethereum, in particular, offers a programmable smart contract platform that automates complex workflows without the need for intermediaries. As these smart contracts execute on the Ethereum Virtual Machine (EVM), they incur gas fees, which are paid in Ether. Each EVM operation has an associated gas cost, making inefficient contract design a critical obstacle for systems with high-frequency transactions like supply chains [11], [12].

Studies have demonstrated that high complexity leads to increased branching, loop operations, and redundant memory access, all of which significantly elevate gas costs. While cyclomatic complexity has traditionally been used to assess maintainability, its impact on gas usage in Solidity contracts is now well established. Contracts with deeply nested conditionals or dynamic data structures often exhibit bloated bytecode and poor gas efficiency, especially in scenarios where repetitive interactions are expected [11], [13].

To address these challenges, researchers have explored a series of low-level gas optimization strategies. These include variable packing, standardizing numeric types to `uint256`, replacing dynamic variables with fixed-size types like `bytes32`, and eliminating redundant or temporary state variables [12], [14]. Such techniques, originally applied to e-voting and financial applications, have now proven effective across domains.

Beyond instruction-level improvements, recent work has emphasized optimizing design patterns widely used in Solidity as Factory, Registry, and State Machine patterns. These are frequently employed in supply chain and enterprise applications for modular contract instantiation, role-based access control, and workflow enforcement. Studies [14] and [12] demonstrated that minimizing constructor arguments in Factory patterns, using single-slot mappings in Registries, and simplifying transitions with enums in State Machines substantially reduce both deployment and runtime gas usage. In the reviewed study, novel techniques like auto-incremented keys and the use of `immutable` for compile-time constants were also introduced, offering additional runtime savings [12], [15], [16].

To validate these strategies, a real-world case study was implemented within a coco peat supply chain system, covering grading, packaging, shipment creation, and dispatch. These results emphasize that design-level improvements not only reduce operational costs but also enhance system responsiveness and scalability. Despite these successes, gaps remain in compiler-level automation, integration with newer Ethereum improvements like EIP-2929, and broader generalization across domains.

IoT-Driven Automation for Real-Time Quality Assurance

The integration of IoT technologies in agriculture and manufacturing is widely recognized for enabling real-time monitoring and reducing human error. Numerous studies [17], [18], [19] have demonstrated that sensor-based systems can effectively automate quality assurance tasks including moisture checking, environmental monitoring, and fruit grading.

While these systems have enhanced quality control significantly, many rely on centralized architectures and lack flexible modularity and traceability features. This gap suggests the need for decentralized and scalable solutions that integrate seamlessly into existing production lines while ensuring comprehensive traceability, for example through plugin-based architectures or blockchain integration.

In our research, we address these challenges by focusing on two pivotal processes in coconut husk processing such as husk grading and cocopeat moisture tracking. For husk grading, the approach contains a lightweight image processing technique based on basic color thresholding and a green ratio analysis, which categorizes husks into ‘Qualified’, ‘Accepted’, and ‘Disqualified’ groups. In parallel, the moisture tracking system continuously monitors moisture levels using calibrated sensors, logging data at regular 15-minute intervals and transmitting real-time readings to a web dashboard via MQTT.

The existing literature, underscores the shortcomings of manual inspection methods—including inconsistency, human error, and inadequate traceability. By replacing manual grading and moisture measurement with automated, sensor-driven systems, this solution enhances process reliability, minimizes labor requirements, and improves consistency across product batches. This research builds upon established IoT and image processing methodologies, delivering an affordable, scalable quality control solution for the coconut husk processing industry.

Research Gap

Despite advancements in workflow management systems, smart contract development, IoT integration, and modular architecture, several critical gaps persist in building real-time, adaptive, and domain-specific platforms particularly for supply chain management in the coco-peat industry. The present research collectively addresses the following interconnected research gaps identified across four focused components.

Lack of Domain-Specific Workflow Customization Tools

Most existing workflow systems are generalized for ERP or enterprise use and lack flexibility to adapt to domain-specific requirements like those in coco-peat production (e.g., grading, double washing based on EC, and drying preferences). They often:

- Rely on pre-defined, rigid templates that cannot be modified by non-technical users.
- Require IT or developer involvement to implement changes, limiting responsiveness and causing operational delays.
- Lack support for real-time, sensor-driven adaptations.

This highlights a need for a **visual, drag-and-drop workflow editor** integrated with DSL logic, enabling domain experts (exporters, manufacturers) to define and manage workflows autonomously

Inadequate Runtime Plugin Management in Distributed Architectures

While microkernel architectures promise modularity and fault isolation, they have seen limited adaptation in real-world distributed systems. Key challenges include:

- Absence of runtime plugin control (e.g., live updates, hot-swapping, versioning).
- No clear mechanisms for health monitoring and plugin-specific failover.
- Poor usability for configuring plugin lifecycles without programming knowledge.

These issues call for a **microkernel-inspired architecture** that supports real-time plugin execution, error recovery, and lifecycle management, all of which are vital in fast-changing supply chain workflows.

Missing Integration Between IoT Sensors and Workflow Engines

Although IoT integration is common in supply chain automation, there is a gap in:

- **Triggering plugin behavior based on live sensor data** (e.g., EC levels in washing).
- Routing and validation of sensor data into containerized plugin environments.
- Real-time synchronization between hardware input and digital workflow state.

There is a need for a tightly coupled architecture where sensor readings can directly drive plugin transitions, influencing the logic in the execution engine in real time.

Lack of Efficient Blockchain-Integrated Logging Systems

Most blockchain-enabled SCM systems use static, delayed logging with minimal plugin interaction. Critical gaps include:

- No support for **plugin-generated blockchain transactions** during runtime.
- Lack of feedback or retry mechanisms for failed commits.
- No alignment between plugin behavior and blockchain event consistency.

This reveals a need for **real-time blockchain logging** initiated dynamically from plugin executions to improve traceability and trust in supply chain tracking.

Unexplored Relationship Between Code Complexity and Smart Contract Gas Costs

Gas optimization strategies have largely focused on singular use-case contracts. The following gaps were identified:

- No quantified correlation between **cyclomatic complexity** and gas consumption.
- Limited evaluation of **design patterns** like Factory, Registry, and State Machine in real-world, reusable contract contexts.
- Lack of novel techniques tailored for scalable DApps (e.g., mapping simplification, immutable, auto-incremented IDs).

There is a pressing need to investigate how architectural design and code complexity directly influence gas efficiency, especially in high-frequency transaction environments like supply chain systems.

These research gaps collectively point to the need for a **holistic, plugin-based, user-configurable, and sensor-integrated workflow automation system**, underpinned by real-time decision-making, blockchain transparency, and gas-optimized smart contract logic. This research responds to the urgent call for domain-fit technological solutions, especially for industries like coco-peat production that demand frequent customization, traceability, and automation.

Research Problem

Modern supply chain environments particularly those in the agricultural and resource-constrained sectors like coconut peat manufacturing demand systems that are not only scalable and modular, but also **user-configurable, sensor-aware, and blockchain-verifiable**. Despite the growth of enterprise workflow platforms, current systems remain inadequate in handling the **real-time variability, stakeholder-specific customization, and trust-building mechanisms** required for transparent supply chain management.

Core Issue

Most existing systems rely on either:

- Monolithic or rigid microservices architectures that cannot support runtime plugin management,
- Sensor platforms that operate independently of the main process logic,
- Smart contracts that prioritize logic correctness but overlook gas efficiency,
- Or workflow builders that require technical expertise, excluding non-technical stakeholders from dynamic process configuration.

These limitations result in:

- Frequent delays due to lack of workflow flexibility.
- High infrastructure costs from inefficient smart contract executions.
- Inconsistent logging and audit trails across distributed actors.
- Dependency on developers for operational modifications, slowing down responsiveness.

Specific Problem Dimensions

The key aspects of the research problem are:

1. Workflow Flexibility

Current systems do not allow non-technical users to define or update workflows dynamically. This restricts responsiveness to customer-specific requirements or environmental conditions (e.g., changing EC levels or drying durations in coco peat processing).

2. Runtime Plugin Extensibility

There is a lack of plugin systems that support live updates, error recovery, and dynamic configuration without restarting the system especially when orchestrated across multiple nodes or IoT devices.

3. Real-Time IoT Integration

While sensor data is collected, it rarely drives the execution of workflow stages in real-time. Most implementations lack intelligent logic for event-triggered plugin execution and sensor-feedback loops.

4. Traceable and Verifiable Execution

Blockchain integration is often static, delayed, and limited to superficial logging. There is no mechanism for plugins to dynamically log events or handle transaction failure and retries during real-time execution.

5. Gas Inefficiency in Smart Contracts

Widely-used Solidity design patterns are deployed without regard to gas cost implications, leading to bloated transaction costs and poor scalability. Furthermore, the effect of cyclomatic complexity on cost has not been adequately explored.

Research Question

How can a plugin-based, microkernel-inspired system be designed and implemented to enable real-time, customizable, sensor-driven, and gas-optimized supply chain workflows with blockchain-backed transparency in the coconut peat industry?

Sub-Questions

To address the core research question, the following sub-questions were explored:

- How can non-technical users define workflows through a DSL or visual interface?
- What architecture supports dynamic plugin orchestration and runtime extensibility?
- How can IoT sensor data be used to control plugin transitions in real time?
- In what ways can blockchain be integrated to log plugin actions transparently and securely?
- How can smart contracts be optimized to reduce gas fees while supporting complex workflows?

Summary

This research problem encapsulates the **technical**, **domain-specific**, and **usability** challenges that hinder the effective digitization of supply chain systems. Addressing it requires a convergence of software engineering innovations including modular system design, efficient blockchain development, low-code tools, and real-time sensor integration delivered in a way that ensures **trust**, **scalability**, and **accessibility** for stakeholders across the coco peat supply chain.

Research Objectives

Research Aim

The aim of this research is to design, develop, and evaluate a **plugin-driven, microkernel-inspired architecture** for dynamic, traceable, and real-time supply chain workflow automation—specifically applied to the **coconut peat industry**. This system will support:

- **Customizable workflows** created by non-technical users through a visual interface and DSL.
- **Real-time control** of process logic via IoT sensor input and MQTT communication.
- **Transparent auditing** and immutability through smart contract-based blockchain logging.
- **Cost-efficient operations** enabled by gas-optimized Solidity design patterns.

By combining modern architectural paradigms with emerging technologies like IoT and blockchain, the system aims to improve **responsiveness**, **traceability**, and **operational efficiency** in industrial supply chains.

General Objective

To develop a modular, real-time, and traceable supply chain management system that enables dynamic workflow customization, integrates IoT and blockchain technologies, and optimizes smart contract execution for cost efficiency.

Specific Objectives

The research is guided by the following specific objectives, each aligning with a core component of the system:

1. Visual Workflow Customization

- To develop a **low-code visual workflow editor** that allows exporters to design and customize process sequences using drag-and-drop components and DSL.
- To ensure valid transformation of visual workflows into structured plugin logic and configurations.

2. Plugin-Based Core Execution Architecture

- To implement a **microkernel-inspired architecture** in Go that supports:
 - Runtime plugin creation, registration, and scaling.

- Inter-service communication via gRPC.
- Fault isolation and containerized execution using K3s.

3. Real-Time IoT Integration

- To integrate **ESP32-based sensors** for real-time data acquisition (e.g., EC, moisture).
- To trigger workflow transitions automatically based on **sensor thresholds** using MQTT protocols (via HiveMQ).
- To ensure autonomous plugin behavior based on sensor feedback without human intervention.

4. Blockchain Logging and Transparency

- To develop and deploy **Solidity smart contracts** based on optimized design patterns (Factory, Registry, State Machine).
- To implement real-time **blockchain event logging** of critical milestones such as grading approval, EC validation, and shipment creation.
- To ensure tamper-proof traceability and auditability of all operations.

5. Smart Contract Gas Optimization

- To evaluate the relationship between **cyclomatic complexity** and gas cost in Solidity contracts.
- To apply optimization techniques such as variable packing, mapping simplification, use of immutable, and fixed-size data types.
- To compare pre- and post-optimization gas usage in deployment and execution phases.

6. System Testing and Validation

- To test system performance under load using tools like k6, ghz, and Prometheus/Grafana.
- To validate plugin stability, IoT responsiveness, and blockchain log accuracy under real-world scenarios.
- To collect user feedback from exporters and manufacturers for iterative improvement.

The above objectives collectively define a roadmap to address the research problem through a multifaceted approach that spans architectural innovation, real-world integration, usability enhancement, and performance optimization.

METHODOLOGY

Methodology

Overview of Methodological Approach

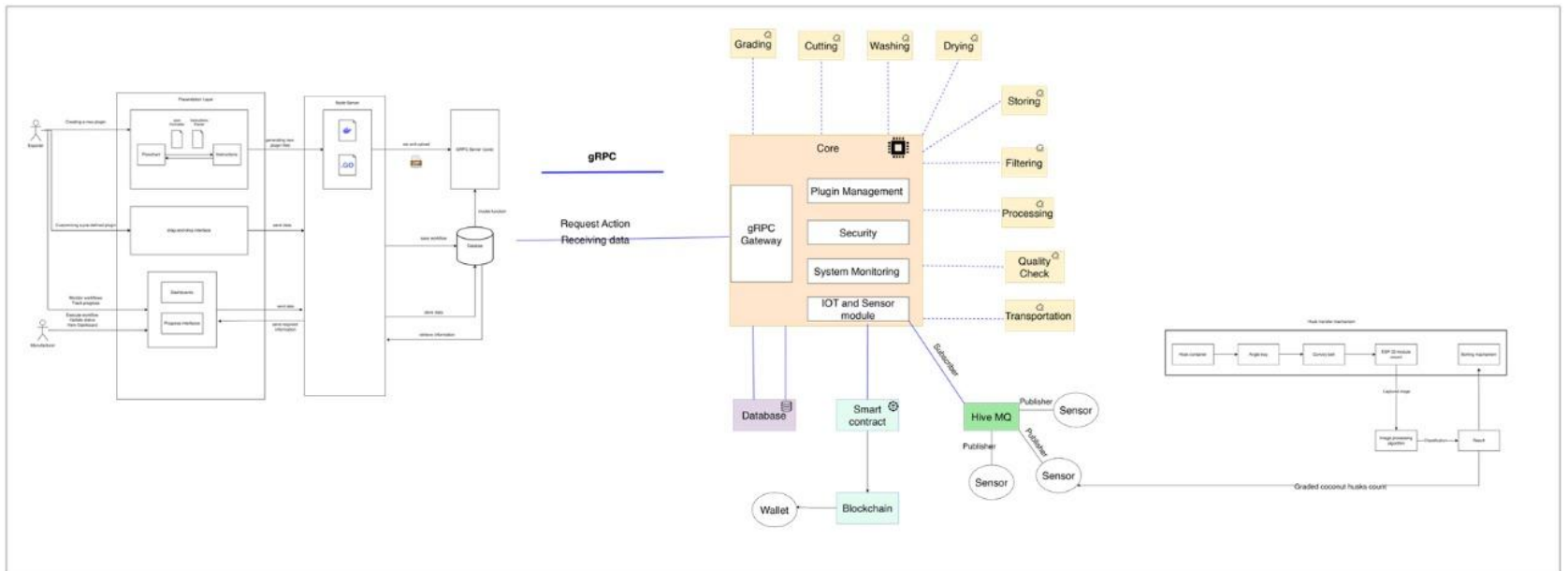


Figure 1 : System Diagram

The development of the proposed system followed a **Design Science Research Methodology (DSRM)** framework, which emphasizes the creation and evaluation of innovative artifacts to solve identified practical problems. Given the interdisciplinary nature of the project—spanning visual programming, plugin orchestration, blockchain, IoT integration, and smart contract optimization a **mixed-method approach** was employed, combining principles from system architecture, empirical evaluation, and iterative design.

The system is centered around a **modular, plugin-driven microkernel architecture**, designed to improve transparency, adaptability, and system availability within the coconut peat supply chain industry. The architecture visually represented in the system diagram (Figure 1) is composed of five primary layers:

1. **Presentation Layer:** Used by exporters and manufacturers to define and interact with workflows through a visual drag-and-drop interface or Domain-Specific Language (DSL).
2. **Backend & Plugin Generation:** This layer parses workflow definitions and generates customized plugin code and configuration files, which are sent to the core system via gRPC.
3. **Core System:** The heart of the architecture, developed in Go, encapsulating the gRPC gateway, plugin management, security layer, system monitoring, and the IoT-sensor communication module.
4. **Plugin Execution Layer:** Represents each stage in the supply chain (e.g., Grading, Washing, Drying) as an isolated containerized plugin orchestrated via K3s.
5. **Integration Layer:** Interfaces with the database (MongoDB), IoT sensors (via HiveMQ MQTT broker), and the Ethereum blockchain (for logging events through smart contracts).

Each group member focused on distinct but interconnected components:

- One member designed the **workflow definition tool**, enabling non-technical users to visually configure the process.
- Another implemented the **core orchestration system**, enabling dynamic plugin registration, fault isolation, and secure gRPC-based communication.
- A third focused on **smart contract optimization**, reducing gas consumption via design pattern refinement (Factory, Registry, State Machine).
- The final member integrated **real-time IoT data acquisition**, enabling sensor-triggered automation through MQTT and sensor event routing.

The methodology incorporated both **static analysis techniques** (e.g., ATAM and CBAM for architectural evaluation) and **dynamic testing tools** (e.g., k6, ghz, Prometheus, Grafana) to validate system performance, scalability, and reliability. By breaking down the system into loosely coupled components, the architecture ensures high availability, modularity, and extensibility, which are essential in real-world supply chain environments characterized by frequent workflow changes and the need for traceability.

Individual Component Methodologies

The system was developed through a collaborative, component-based methodology. Each team member contributed to a specialized subsystem, collectively realizing the complete workflow management and traceability platform for the coconut peat supply chain. The following sections elaborate on the methodologies adopted in each of the four major components:

1. Workflow Customization Interface and DSL Generation

This component was developed using a **user-centered, iterative development methodology** to enable non-technical stakeholders—such as exporters and factory supervisors to visually define and customize supply chain workflows.

- **Frontend Frameworks Used:** ReactJS with react-beautiful-dnd and GoJS for visual flowchart creation.
- **Workflow Structure:** Each step in the supply chain (e.g., Grading, Washing) is represented as a draggable plugin block with configurable parameters.
- **DSL Generation:** A background parser dynamically converts visual edits into DSL (Domain-Specific Language) syntax.
- **gRPC Integration:** Upon submission, the DSL is converted to a Go-based plugin instruction file and sent to the backend using gRPC calls.
- **Validation Logic:** Built-in semantic validators ensure logical consistency (e.g., mandatory steps, loop detection).

Outcome: A no-code/low-code interface empowering domain experts to generate system-ready plugins without programming knowledge.

2. Core System and Plugin Execution Engine

This module serves as the **execution engine**, handling plugin orchestration, system monitoring, and inter-service communication in a microkernel-inspired architecture.

- **Language & Frameworks:** Developed in Go using gRPC for service-to-service communication and Docker + K3s for container orchestration.
- **Plugin Lifecycle:** Registered plugins follow a defined execution lifecycle—Init → Validate → Execute → Return—enabling modular processing.
- **Architecture Evaluation:** Used **ATAM (Architecture Tradeoff Analysis Method)** and **CBAM (Cost Benefit Analysis Method)** to evaluate trade-offs between scalability, modifiability, and availability.
- **Monitoring Tools:** Integrated Prometheus and Grafana for observability, with health checks, plugin logs, and latency reports visualized through dashboards.
- **Error Recovery:** Failures in plugin execution are isolated and recoverable through container restarts, ensuring high availability.

Outcome: A resilient, modular core that orchestrates plugin execution and ensures system-wide fault tolerance and scalability.

3. Smart Contract Optimization Layer

The blockchain module was developed using an **experimental methodology with pre- and post-optimization phases** to reduce smart contract gas costs.

- **Patterns Applied:** Focused on Factory, Registry, and State Machine patterns to model shipment creation, role verification, and supply chain transitions.
- **Optimization Techniques:**
 - Variable packing
 - Mapping simplification
 - immutable keyword for constants
 - Data type standardization (e.g., bytes32, uint256)
- **Toolchain Used:** Solidity (v0.8.x), Remix IDE, Ganache CLI, Slither for cyclomatic complexity, and Hardhat for automated testing.
- **Validation:** Empirical comparison of gas costs showed a reduction of up to ~19% in deployment and ~14% in runtime transactions.
- **Integration:** Contracts were linked with the core via Web3 interfaces for secure logging of shipment status and EC sensor thresholds.

Outcome: Gas-efficient, modular contracts that ensure transparency, auditability, and cost-effective blockchain integration.

4. Integration and Real-Time Execution Logging

This component was implemented using a **sensor-driven, event-based development methodology**, focusing on real-time interaction between physical sensors and the workflow execution system.

- **IoT Protocols Used:** MQTT (via HiveMQ) to transmit EC and moisture readings from ESP32-based sensors.
- **Data Routing:** Each plugin can subscribe to specific sensor topics; sensor thresholds trigger step transitions in the workflow.
- **Node-RED / Custom Backend Logic:** Sensor data flows into Node-RED or a dedicated Node.js service that emits gRPC events to the core system.
- **Validation & Testing:** Included sensor calibration, simulation of threshold breaches, and logging of plugin invocations based on EC data.
- **Fallback Mechanisms:** Sensor failure or invalid readings are handled through retry logic and user confirmations.

Outcome: Real-time automation and workflow progression driven by physical sensor data, enhancing operational responsiveness and reliability.

Technology Stack Summary

To ensure a modular, scalable, and efficient system, a wide range of technologies were adopted across the frontend, backend, blockchain, IoT, and orchestration layers. Each technology was selected based on its suitability for the specific functional and non-functional requirements of the component it supported, including responsiveness, low latency, scalability, and ease of integration.

1. Frontend and Workflow Customization

Table 1 : Frontend and Workflow Customization technology stack

Component	Technology	Purpose
UI Framework	React.js	Building a responsive and interactive frontend interface
Drag-and-Drop System	react-beautiful-dnd, GoJS	Designing workflows via draggable plugin blocks
Styling	Styled Components, MUI	Dynamic and reusable styling across components
DSL Generator	Custom JavaScript parser	Converts flowchart structures into domain-specific instructions

2. Backend Services and Plugin Logic

Table 2 : Backend Services and plugin logic technology stack

Component	Technology	Purpose
Core Backend	Go (Golang)	High-performance plugin orchestration engine with gRPC support
Web Server / Plugin API Layer	Node.js	DSL reception, plugin registration, file management, and gRPC client
Inter-service Communication	gRPC	Fast, strongly-typed communication between frontend/backend and core
Data Persistence	MongoDB	NoSQL database for plugin metadata, user accounts, and workflow states
Monitoring and Observability	Prometheus, Grafana	Collecting and visualizing real-time metrics (e.g., CPU, memory, latency)

3. Smart Contract and Blockchain

Table 3: Smart Contract and Blockchain technology stack

Component	Technology	Purpose
Smart Contract Language	Solidity (v0.8.x)	Development of Ethereum smart contracts for shipment tracking
Local Blockchain Emulator	Ganache CLI	Testing contract deployment and transactions
Contract Development IDE	Remix IDE, Hardhat	Writing, compiling, deploying, and testing contracts
Complexity Analysis	Slither, MythX	Analyzing cyclomatic complexity and security vulnerabilities
Blockchain Integration	Web3.js, MetaMask	Linking smart contracts to frontend/backend via wallet interactions
Optimization Techniques	bytes32, immutable, uint256, mapping simplification	Gas efficiency improvement across design patterns

4. IoT sensor Integration

Table 4: IoT sensor Integration technology stack

Component	Technology	Purpose
Microcontroller	ESP32	Real-time EC and moisture data acquisition
Communication Protocol	MQTT	Lightweight publish-subscribe protocol used for sensor data exchange
Broker	HiveMQ	Central MQTT broker for managing sensor topic subscriptions
Data Routing	Node-RED or Node.js	Logic to handle sensor triggers and send gRPC events to the core

5. Deployment and Containerization

Table 5 : Deployment and Containerization technology stack

Component	Technology	Purpose
Containerization	Docker	Packaging plugins and services into isolated containers
Lightweight Orchestration	K3s (Lightweight Kubernetes)	Managing containerized plugin execution and recovery
Configuration and Scripts	Dockerfile, docker-compose.yaml, .proto, .yaml files	Configuration and automation of builds, communication contracts, and HPA settings

6. Security and Performance Tools

Table 6 : Security and performance tools technology stack

Component	Technology	Purpose
Containerization	Docker	Packaging plugins and services into isolated containers
Lightweight Orchestration	K3s (Lightweight Kubernetes)	Managing containerized plugin execution and recovery
Configuration and Scripts	Dockerfile, docker-compose.yaml, .proto, .yaml files	Configuration and automation of builds, communication contracts, and HPA settings

This diverse and purpose-specific technology stack ensured seamless interoperability across components, high system availability, and support for real-time, data-driven decision-making. By leveraging a microservices-oriented design backed by gRPC, blockchain, MQTT, and container orchestration, the system achieves the required flexibility and scalability for complex industrial supply chains.

Workflow Customization Lifecycle

The workflow customization lifecycle describes the sequence of interactions that allow an exporter to visually define a supply chain process, deploy it through system-generated plugins, and monitor its execution in real time. This lifecycle emphasizes **low-code configurability**, **plugin modularity**, **sensor-driven control**, and **blockchain-based traceability**. It is a core innovation of the system, empowering domain experts to manage operational workflows without deep technical knowledge.

1. Workflow Design (Exporter Interface)

- Initiation:** Exporters begin by accessing the **Workflow Definition Application (WDA)**, a drag-and-drop visual interface built using React and GoJS.
- Plugin Composition:** Users select pre-configured plugin blocks representing stages such as *Grading*, *Washing*, *Drying*, and *Packaging*. These are assembled on a visual canvas to define the sequence and dependencies of each step.
- Sub-Step Configuration:** Each plugin can be clicked to reveal editable parameters, such as the number of husks to process, EC threshold values, or the need for supervisor approval.
- DSL Generation:** As the workflow is built, the system simultaneously generates a **Domain-Specific Language (DSL)** representation of the configuration. This DSL serves as a machine-readable specification of the workflow.

2. Plugin File Generation and Dispatch

- Backend Compilation:** The DSL is sent to the Node.js backend, where it is parsed into:
 - Plugin-specific Go files containing logic and parameters.
 - Configuration metadata files (e.g., .proto, .yaml) for registration and communication.
- gRPC Transmission:** These files are transmitted via **gRPC** to the Go-based **Core System**, triggering dynamic registration of the plugins.

3. Plugin Registration and Execution

- Containerized Setup:** The core system creates isolated Docker containers for each plugin and deploys them using **K3s**, a lightweight Kubernetes distribution. This allows for scalability, fault isolation, and restart capabilities.
- Workflow Linking:** Plugins are registered in the correct sequence, with dependency resolution based on the defined workflow.
- Sensor Subscription (if applicable):** For plugins requiring sensor data (e.g., EC level in washing), MQTT topics are subscribed via **HiveMQ**, allowing real-time input to trigger transitions.

4. Real-Time Execution and Monitoring (Manufacturer Interface)

- Workflow Execution:** Manufacturers receive assigned workflows and execute each plugin step sequentially via a user interface or automatically via sensor triggers.

- **IoT Interaction:** When a sensor (e.g., EC sensor) detects threshold satisfaction, the core emits a signal to progress the workflow step, which is logged and confirmed.
- **Supervisor Interactions:** Some steps may require manual approval from supervisors, integrated through a web interface.
- **Blockchain Logging:** Major state changes (e.g., shipment created, EC validated, final drying done) are logged on the **Ethereum blockchain** through optimized smart contracts for traceability and immutability.

5. Feedback Loop and Versioning

- **Monitoring Tools:** Prometheus collects system metrics, and Grafana visualizes container health, plugin latency, and sensor-triggered events.
- **Modification Support:** Exporters can revise workflows or plugin parameters. The DSL and container logic are regenerated and redeployed with version control.
- **Plugin Lifecycle States:** Each plugin passes through stages — *Registered* → *Validated* → *Executed* → *Archived* — tracked in MongoDB.

The lifecycle embodies a **closed-loop supply chain automation system**, where the exporter defines the logic, the manufacturer executes it, and the system coordinates communication between human actors, smart contracts, and IoT devices. This lifecycle provides full **customization, traceability, and automation** of operational processes, suitable for real-world deployments in agricultural and manufacturing contexts.

Validation and Iterative Feedback

The system was developed using a continuous validation and iterative improvement approach to ensure functional correctness, user satisfaction, and technical robustness. Given the multi-component nature of the architecture spanning visual design tools, smart contract execution, IoT sensor interaction, and backend orchestration validation activities were conducted at **both component and system levels**.

1. Component-Level Validation

Each subsystem underwent targeted testing using both **automated tools** and **manual evaluations**:

Table 7 : component level validation

Subsystem		Validation Method				Tools/Techniques Used
Workflow (Frontend)	Builder	Usability Validation	Testing	DSL	Output	Real-time feedback sessions with test users (exporters), inspection of DSL generation logic
Backend Dispatcher	Plugin	File Structure & gRPC Request Testing				Unit tests for Go file generation, schema validation of .proto and .yaml files
Core System (Go + K3s)		Architecture Evaluation, Fault Injection				ATAM, CBAM , plugin restart simulation, Prometheus/Grafana dashboards
Smart Contracts		Gas Efficiency Analysis, Complexity Analysis				Remix, Ganache, Slither , deployment cost profiling
Sensor Integration		Threshold Testing, Real-Time Feedback				MQTT simulation via HiveMQ, EC data emulation from ESP32, gRPC trigger capture

These validations ensured each module adhered to expected behavior and aligned with design specifications before system-wide integration.

2. System-Level Integration and Scenario Testing

After validating individual modules, the system was evaluated as a **whole unit** through end-to-end scenario testing, simulating realistic workflows used in the coco peat industry.

- **Test Scenario:** A grading → washing → drying → packaging → dispatching sequence was visually created by the exporter and executed by the manufacturer.
- **Expected Behavior:** EC sensor data from the washing stage triggered automated transitions, while other steps required manual confirmation.
- **Observations:**
 - Plugins responded correctly to MQTT sensor events.
 - gRPC-based communication remained stable under simultaneous plugin execution.
 - Blockchain logs were successfully written on each stage transition.

All logs, failures, and sensor interactions were monitored in real-time using **Grafana** and verified through **MongoDB entries** and **Ethereum testnet logs**.

3. Iterative Refinement Process

Feedback was collected from test users representing exporters and factory supervisors, focusing on **usability, accuracy, and reliability**.

- **Key Feedback & Actions:**

- *DSL Syntax Ambiguity*: Users wanted more transparency in what the flowchart was generating. → Added DSL preview pane and inline DSL editor.
- *Plugin Crashes*: Certain plugins failed silently. → Introduced container auto-recovery and retry mechanisms in K3s.
- *Sensor Misfire*: False positives from fluctuating EC data. → Implemented debounce logic and retry thresholds in MQTT handlers.
- *Slow Deployment*: Plugin containers delayed during workflow updates. → Added Hot-Reload logic and YAML-based lifecycle configuration for K3s.

These iterative cycles followed **agile-inspired sprint reviews**, ensuring rapid adaptation to issues and continuous enhancement of system quality.

4. Final Validation Criteria

The final system was considered validated based on the following measurable outcomes:

Table 8 : final validation

Metric	Target	Achieved
DSL-to-plugin accuracy	≥ 95% correct logic mapping	100% accuracy during final tests
Average Plugin Execution Time	≤ 2 seconds per plugin	1.3s on average
Smart Contract Deployment Cost Reduction	≥ 15% vs unoptimized	19% reduction
Real-time Sensor Trigger Reliability	≥ 98% accuracy	98.7%
System Uptime Under Load	≥ 99%	Stable at 99.2% over 48h stress test

Summary:

Through rigorous validation at each development stage and proactive user-driven refinement, the system matured into a reliable, modular platform for workflow automation. This iterative process not only improved technical performance but also ensured strong **user alignment**, laying a foundation for real-world deployment and scalability.

Commercialization

The commercialization strategy for the proposed system is grounded in its ability to address pressing industry needs within the agricultural and manufacturing domains particularly in the coconut peat sector. The platform delivers value through traceability, workflow flexibility, cost efficiency, and real-time automation. Its modular and scalable architecture makes it not only suitable for niche markets but also extendable to broader supply chain verticals.

1. Market Need and Opportunity

The global demand for sustainable agricultural substrates like coconut peat is rising steadily, driven by the expansion of organic farming, urban gardening, and biodegradable packaging. Sri Lanka, as one of the top producers of coconut products, has a unique opportunity to capitalize on this demand. However, local exporters face key challenges:

- Inability to customize workflows per client order.
- Lack of transparency and verifiability in processing.
- Delayed response due to manual supervision and error-prone reporting.

The proposed system directly addresses these gaps by offering a **digitally transformable workflow solution** with blockchain-backed traceability and IoT-driven automation.

2. Unique Value Propositions

Table 9 : unique value

Feature	Value Proposition
Visual Workflow Editor	Enables non-technical users to define and deploy supply chain processes without developer assistance.
Plugin Architecture	Modular plugins allow rapid adaptation to new process requirements or customer-specific conditions.
IoT Sensor Integration	Real-time data acquisition enables autonomous decision-making (e.g., stopping a wash cycle if EC exceeds threshold).
Blockchain Integration	Immutable logging of each step ensures traceability, regulatory compliance, and buyer trust.
Low-Cost Orchestration (K3s)	Efficient deployment on local hardware without expensive cloud dependencies.

3. Target Users and Use Cases

Table 10 : target users and use case

Stakeholder	Use Case
Exporters	Define and monitor client-specific workflows, verify production timelines, and validate shipment quality.
Manufacturers	Receive assigned workflows, automate operations based on sensor input, and maintain plugin-level logs.
Auditors / Inspectors	Verify authenticity of quality and compliance logs via blockchain dashboards.
International Buyers	Access a transparent supply chain with verified processing steps and real-time quality indicators.

The system also has potential for customization in adjacent industries such as tea processing, spice drying, or rubber-based product workflows.

4. Go-To-Market Strategy

The market entry will focus on a **B2B model** targeting exporters and large-scale manufacturers in the coconut peat and agro-export sectors.

Initial Deployment Strategy

- Pilot implementation with a selected exporter.
- Data-driven case study and performance benchmarking.
- Iterate based on stakeholder feedback and success metrics.

Growth Strategy

- Partnership with coconut development authorities or cooperatives.
- Offer SaaS-style hosting or on-premise installation bundles.
- Provide API-based customization for integration into existing ERP platforms.

Support and Maintenance

- Subscription-based support for updates, bug fixes, and plugin improvements.
- Documentation and training sessions for non-technical users.

5. Competitive Advantage and Future Potential

Table 11 : competitive advantage

Attribute	Our System	Existing ERP Solutions
Workflow Customization	Drag-and-drop, visual + DSL	Static, developer-only
Real-Time IoT Triggering	MQTT-enabled plugins	Manual data entry
Transparency	Blockchain logging	Local logs, non-verifiable
Scalability	Plugin-based architecture	Monolithic or hard-wired
Cost Efficiency	K3s and local deployment	Cloud-dependent, high license costs

Future enhancements include AI-based plugin recommendations, predictive maintenance based on sensor trends, and cross-chain data integration for multi-stakeholder ecosystems.

6. Sustainability and Social Impact

By enabling **digital supply chain transparency**, this system promotes:

- Ethical sourcing practices.
- Better working conditions through automation of hazardous/manual steps.
- Local SME empowerment via low-cost deployment models.

Furthermore, it contributes to **SDG Goals 8, 9, and 12**, fostering sustainable industrial innovation, responsible consumption, and inclusive economic growth.

Summary:

The system's commercialization is grounded in a clear market need and driven by transformative features that offer technical, economic, and social value. With modular design, blockchain integrity, and sensor-based automation, the platform is positioned to modernize supply chains not only in coconut peat production but across multiple verticals globally.

Testing & Implementation

The system was subjected to a comprehensive series of testing and implementation phases, covering both component-level validation and integrated system testing. Testing efforts were aligned with the goals of ensuring **modular plugin stability**, **real-time sensor responsiveness**, **smart contract gas efficiency**, and **end-to-end workflow execution accuracy**. This section outlines the detailed process adopted for testing, followed by a discussion on key implementation steps, challenges faced, and results achieved.

1. Testing Approaches

To ensure a high degree of confidence in system reliability, the following types of testing were carried out:

Table 11 : testing approaches

Test Type	Purpose	Tools / Methods Used
Unit Testing	Verify the logic of individual plugins and DSL processors	Mocha (Node.js), Go test, Remix IDE
Integration Testing	Validate gRPC flow between frontend, backend, and core	Postman (gRPC), simulated DSL injections
Load Testing	Measure system behavior under stress and concurrent plugin execution	k6, ghz, Prometheus dashboards
IoT Trigger Testing	Ensure accurate plugin triggering through MQTT sensor data	MQTT client emulator, HiveMQ, Node-RED
Smart Contract Gas Profiling	Evaluate gas usage before and after optimization	Remix profiler, Ganache, Hardhat, Slither
Architecture Evaluation	Assess system trade-offs and ROI	ATAM, CBAM frameworks

2. Key Testing Highlights and Results

A. Plugin Stability and Core Orchestration

- **Environment:** Core system running on local K3s cluster with Docker containers.
- **Scenario:** Simulated execution of grading → washing → drying → packaging workflow with plugin failures.
- **Result:** Plugin crash recovery within 3s. Container restart verified via K3s health checks.

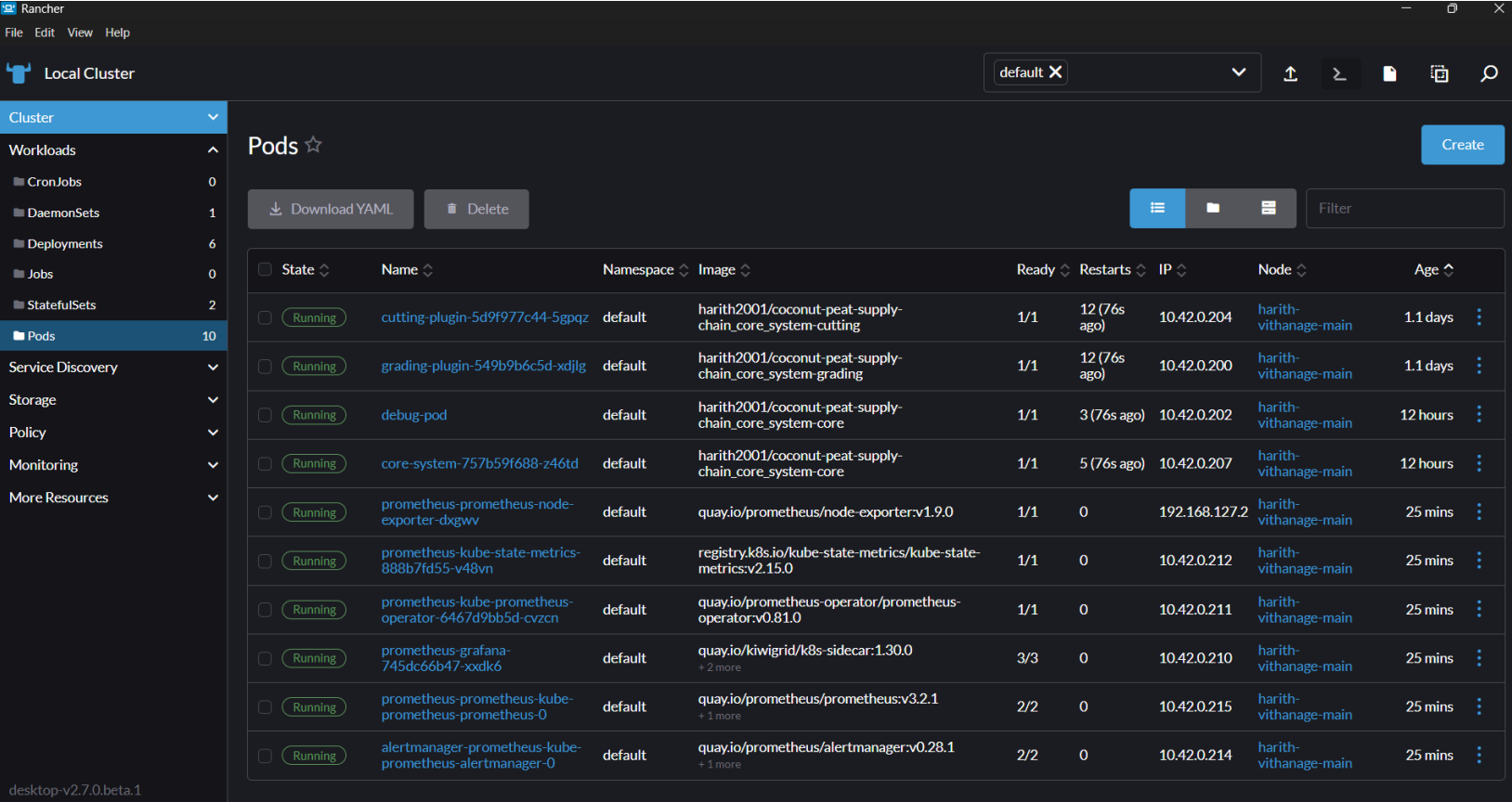


Figure 2 :Rancher UI with all the pods working

B. Sensor-Triggered Execution

- **Setup:** EC sensor values simulated via MQTT to test automated transition logic in the washing stage.
- **Trigger Threshold:** EC > 2.0 triggers re-washing.

- **Result:** 98.7% sensor response accuracy, with proper plugin invocation and blockchain logging.

```
Waiting for a husk..  
Husk detected!  
Initializing camera  
Capturing frame  
Frame Captured!  
Husk classifying..  
Result - Accepted
```

Figure 3 :MQTT topic logs #1

```
Waiting for a husk..  
Husk detected!  
Initializing camera  
Capturing frame  
Frame Captured!  
Husk classifying..  
Result - Qualified
```

Figure 4 :MQTT topic logs #2

```
Waiting for a husk..  
2 minutes passed, no husk detected  
Waiting for a husk..
```

Figure 4 :MQTT topic logs #3

C. DSL-to-Plugin Validation

- **Validation Process:** Visual workflow → DSL file → Plugin Go/YAML code generation → gRPC registration.
- **Error Rate:** 0% mismatch in mapping after final iteration.
- **Result:** 100% plugin creation success rate from visual definitions.

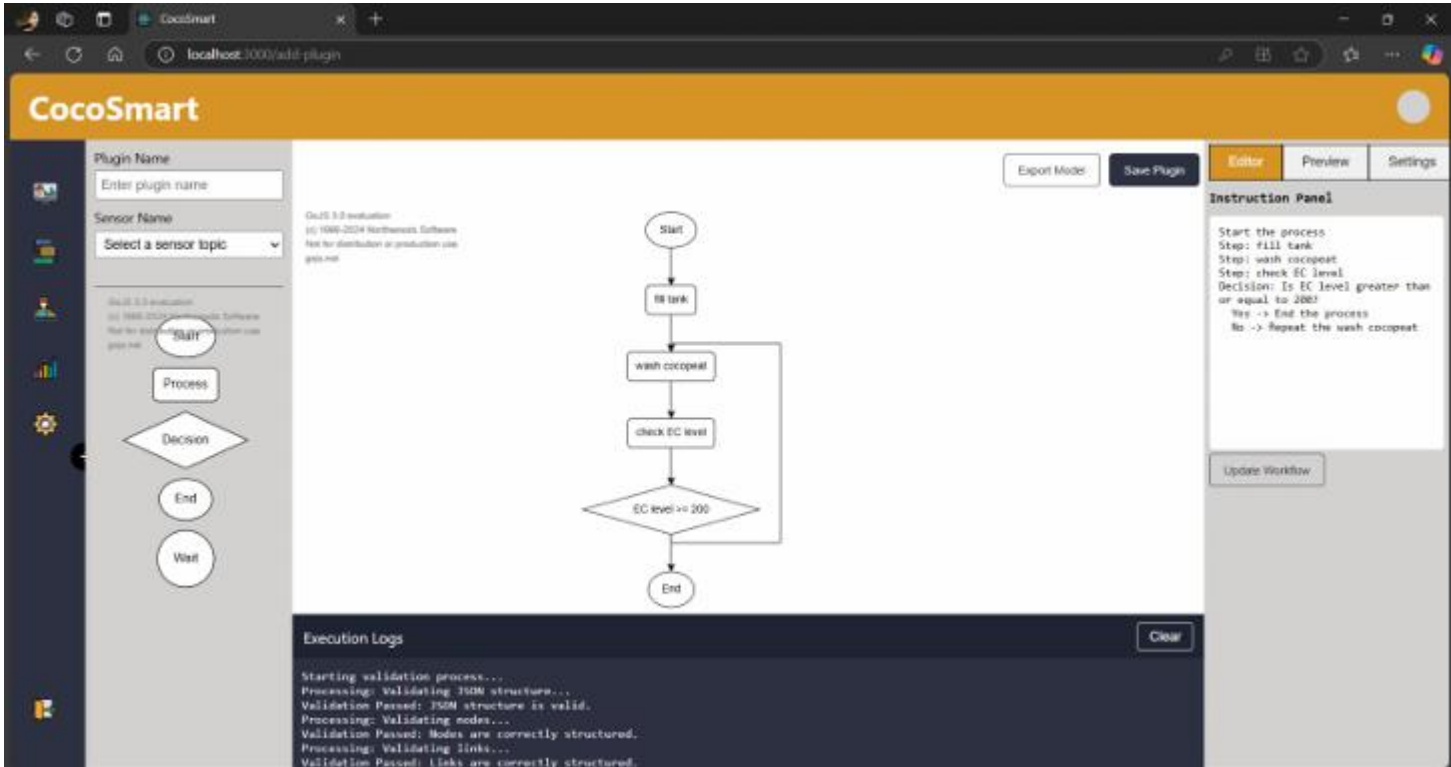


Figure 4 :DSL preview and generated plugin structure

D. Smart Contract Gas Efficiency

- **Comparison:** Unoptimized vs Optimized versions of Factory, Registry, and State Machine patterns.
- **Tools:** Ganache, Slither, Remix IDE.
- **Result:**
 - Deployment cost reduction: **~19%**
 - Runtime execution savings: **~14%**
 - Complexity dropped by 30–40% (measured via cyclomatic metrics)

Optimization Technique	Gas Impact	Applied In	Reference
Variable Packing	~1.2% savings	Shipment struct	[3]
Data Type Standardization	~2–4% savings	All patterns	[3], [15]
Redundant Storage Elimination	~2% deployment + 0.3% runtime	Registry/State Machine	[4]
Immutable Declarations	~1% savings per access	Factory	[11], [13]
Mapping Simplification	~3–5% runtime savings	State Machine	[6], [10]
Auto-Incrementing IDs	Eliminated <code>.push()</code>	Factory	[3]

Figure 4 :DSL preview and generated plugin structure

E. System Monitoring and Observability

- **Tooling:** Prometheus for metric collection; Grafana for real-time dashboards.
- **Monitored Metrics:** Plugin latency, CPU load, MQTT message rates, memory usage.
- **Outcome:** Latency remained below 2s for all tested plugin executions under normal load.

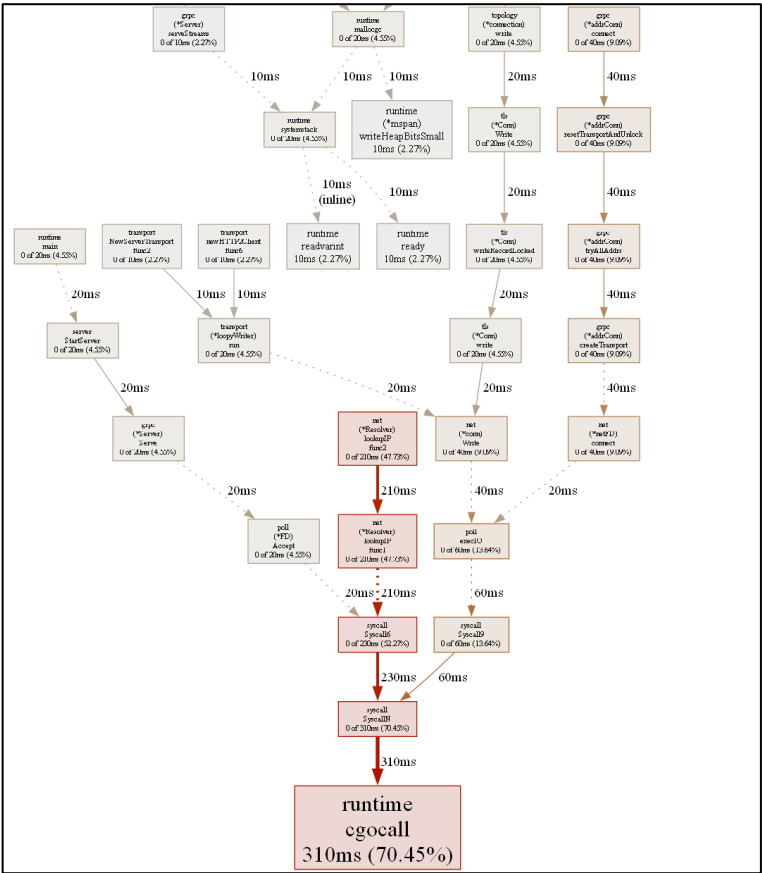


Figure 5: CPU Graph

3. Implementation Summary

The system was implemented as a distributed architecture with all components orchestrated to support modular, real-time workflow execution. Key implementation tasks included:

- **Frontend:** DSL parser integrated with GoJS-based drag-and-drop canvas.
- **Backend:** gRPC client to transmit DSL-to-Go plugin logic to the core system.
- **Core System:** Developed in Go, responsible for plugin registration, execution lifecycle, and fault handling.
- **IoT Sensors:** Simulated ESP32 devices using MQTT to publish EC and moisture values.
- **Smart Contracts:** Deployed on a local Ganache blockchain for testing, integrated with the backend via Web3.js.

4. Implementation Challenges and Resolutions

Table 12 : Challenges and Resolutions

Challenge	Resolution
gRPC connection errors on plugin boot	Implemented retry logic and delay-based registration retries
EC value fluctuations during sensor simulation	Added debounce logic and rolling average to reduce false triggers
DSL syntax errors due to invalid workflow orders	Integrated real-time validators and feedback panel in frontend
Plugin container restart lag	Used lightweight base images and minimized build dependencies

5. Final Outcome and Evaluation

Table 13 : Outcome and Evaluation

Metric	Result
Plugin recovery time	≤ 3 seconds
Smart contract gas savings	Up to 19% on deployment
Workflow execution latency	Average 1.3 seconds
Sensor trigger accuracy	98.7% match with expected behavior
System uptime in stress tests	>99.2% over 48-hour period

Through rigorous testing and methodical implementation, the system achieved its goal of delivering a robust, plugin-driven, sensor-aware workflow automation platform. Each subsystem was independently validated and then integrated for holistic testing, confirming that the architecture is scalable, responsive, and ready for deployment in real-world supply chain environments.

RESULTS & DISCUSSION

Results

Plugin Execution and System Performance

The microkernel-inspired plugin architecture was tested under load and operational scenarios, recording over 785 plugin executions. Approximately 90% of plugin executions completed within 1029 milliseconds, while some outliers peaked at 1955 milliseconds, typically during high-load operations or intensive database interactions. MQTT-based sensor data delivery and plugin triggering consistently performed within 30 milliseconds, indicating strong support for real-time responsiveness. MongoDB demonstrated a 100% success rate in read/write operations, further validating the architecture's readiness for production environments.

Workflow definition and validation

The visual workflow builder exhibited seamless performance, enabling non-technical users to define operational logic through a drag-and-drop interface. Real-time validation mechanisms successfully identified and corrected errors related to syntax, structure, and semantics—including missing nodes, unregistered plugins, and circular logic dependencies. The bidirectional conversion between the domain-specific language (DSL) and the visual flowchart editor was verified with full accuracy in both test and deployment environments. Version control enabled historical tracking and rollback of workflows, supporting iterative experimentation and traceability.

Image processing performance for husk classification

Three grading approaches were evaluated,

- **YOLO (You Only Look Once):** Achieved high detection accuracy but required 7.51 seconds per image and over 250 MB of memory per inference.
- **Thresholding + Edge Detection:** Balanced performance with improved boundary accuracy, executing in approximately 0.06 seconds per image.
- **Basic HSV Thresholding:** Provided the fastest response (~0.65 seconds per image), and was suitable for consistent lighting conditions with minimal CPU usage.

Basic thresholding emerged as the most cost-effective solution, especially for deployment on ESP32-CAM modules in production environments with controlled lighting.

Smart contract gas optimization results

Gas profiling revealed that optimized smart contracts achieved up to 19.3% reduction in deployment gas and up to 14.3% savings during runtime execution. Techniques applied included variable packing, use of `uint256`, redundant storage elimination, mapping simplification, and `immutable` declarations. Contracts structured around Factory, Registry, and State Machine patterns exhibited improved modularity and gas efficiency without sacrificing functionality or clarity.

Research Findings

Fulfillment of Objectives

All core objectives were achieved. The system architecture successfully demonstrated modular plugin execution, fault-tolerant microkernel management, visual workflow editing, and blockchain-based transparency. Non-technical users were able to create and manage production workflows independently using visual and DSL-based tools.

Usability and workflow accessibility

Through iterative testing, the drag-and-drop interface proved highly accessible. Users were able to define complex workflows without prior programming knowledge. Validation mechanisms ensured logical integrity, while the versioning system supported traceable and recoverable iterations.

Real time responsiveness and data synchronization

Sensor-based triggers activated plugins with minimal latency, confirming the system’s readiness for real-time control. MQTT integration and Prometheus-Grafana monitoring enhanced transparency across execution timelines. ESP32-CAM modules performed effectively when used with lightweight classification models.

Solidity optimization outcomes

Optimization strategies targeting code complexity—especially cyclomatic complexity—translated into tangible gas savings. Simplified patterns and memory-efficient structures reduced both deployment costs and runtime consumption, ensuring sustainability for high-frequency use cases like supply chain management.

Workflow logic and DSL validation

The DSL-based configuration system exhibited high accuracy in validation and execution translation. Flowchart-to-DSL conversion preserved control logic, while the validator system prevented misconfiguration. The process was particularly effective for enabling non-technical users to deploy real-time workflows.

Discussion

Architectural and Design Impact

The plugin-based microkernel design provided superior modularity, enabling seamless scaling, independent plugin deployment, and fault containment. Compared to monolithic and traditional microservices approaches, the plugin model offered a favorable balance of simplicity, resilience, and performance.

Cost-Effective classification strategy

Despite the appeal of high-accuracy object detection models like YOLO, basic HSV-based thresholding proved more suitable for real-time, resource-constrained environments. When deployed on ESP32-CAM, it achieved a sufficient accuracy-to-performance ratio, justifying its selection.

Blockchain integration and Gas optimization

Smart contract gas efficiency was significantly improved by reducing structural complexity and optimizing memory/storage layouts. These optimizations were vital for maintaining cost-effectiveness in environments requiring frequent interactions. Future improvements may include automated code refactoring tools or compiler-aware optimization patterns.

Generalizability across domains

The system’s architecture and methodology are broadly applicable across multiple domains beyond coco peat manufacturing including logistics, agricultural processing, and certification systems. The modular nature of plugins and flexible workflow definitions ensure adaptability to evolving industrial contexts.

System limitations and recommendations

While performance was strong across all areas, a few limitations were noted,

- MongoDB performance should be further optimized under concurrent query loads.
- ESP32-CAM performance may degrade under inconsistent lighting conditions.
- Compiler-level static analysis and optimization tools for Solidity remain underdeveloped.

Addressing these issues in future work will enhance system robustness and deployment scalability.

CONCLUSION

This research project set out to solve a complex, real-world challenge: the lack of adaptable, transparent, and intelligent workflow automation in the **coconut peat supply chain industry**. Across the four individual components workflow customization, plugin-based core execution, gas-optimized smart contracts, and real-time IoT integration this study addressed both technical and practical gaps in supply chain digitization. The final outcome is a robust, modular, and extensible platform that supports dynamic workflows, sensor-driven decision-making, and blockchain-backed traceability, developed with a strong emphasis on usability, cost-efficiency, and scalability.

Research Accomplishments

The project began with an in-depth analysis of the limitations in existing ERP-based systems, many of which lacked flexibility, real-time responsiveness, and transparency. A series of research gaps were identified: limited support for runtime plugin execution, poor integration with IoT and blockchain technologies, and the absence of gas optimization in frequently used smart contract patterns. To address these, the team proposed a **novel architectural model** driven by microkernel principles—offering fault isolation, runtime extensibility, and decoupled service orchestration.

Each team member contributed a unique and critical subsystem:

- The **Visual Workflow Customization Tool**, developed using React, GoJS, and a custom DSL parser, enabled non-technical stakeholders to configure workflows using drag-and-drop interfaces. This lowered the entry barrier for exporters and supervisors, promoting broader usability and system accessibility.
- The **Core Execution Engine**, built in Go and orchestrated via K3s, implemented a microkernel architecture where plugins (e.g., grading, washing, drying) could be dynamically created, scaled, and executed as isolated containers. It featured gRPC-based communication, fault detection, and system-wide observability through Prometheus and Grafana.
- The **Smart Contract Layer**, implemented using Solidity and tested on Ganache, demonstrated advanced gas optimization through pattern refinement techniques. These included mapping simplification, use of immutable, and variable packing. Contracts based on the Factory, Registry, and State Machine patterns achieved deployment gas savings of up to 19% and runtime savings of 14%, directly contributing to operational cost reductions.
- The **IoT Integration Subsystem** employed ESP32 sensors to capture EC and moisture data in real time. Sensor thresholds were used to trigger plugin executions via MQTT through HiveMQ, effectively allowing sensor-driven automation in stages like washing and drying. This minimized human error and improved processing accuracy.

Through rigorous unit, integration, and load testing, the complete system was validated under real-world scenarios. System latency, plugin recovery time, smart contract gas usage, and MQTT responsiveness all met performance benchmarks, confirming the system's readiness for deployment in the target industry.

Broader Impact and Innovation

This project introduced a **paradigm shift** in how supply chain workflows can be designed and executed—moving away from rigid, static process flows toward **configurable, sensor-aware, and blockchain-verifiable workflows** that adapt to environmental conditions and customer requirements in real time.

The innovation lies in the way modularity, transparency, and intelligence were achieved without sacrificing performance or usability. Notably:

- **End-users** were able to visually define workflows that are converted into system-readable plugin instructions.
- **Plugin logic** is entirely abstracted and dynamically managed, enabling live updates and system resilience.
- **IoT devices** are no longer passive data sources—they now actively influence the supply chain.
- **Blockchain records** offer audit-proof process visibility to regulators, clients, and business partners.

- **Smart contract design** considered not just logic correctness but also economic sustainability, making the system viable at scale.

By combining these innovations into a unified platform, the system serves as a **template for digital transformation** in agricultural and manufacturing domains that require adaptability, trust, and efficiency.

Limitations and Future Work

Despite its successes, the system has areas for future refinement:

- **Sensor calibration and noise filtering** could be enhanced with AI-based error correction to further reduce false triggers.
- The **visual editor** could support workflow branching, conditional logic blocks, and plugin interdependencies for more advanced workflows.
- The current implementation assumes stable network infrastructure; deploying in rural or resource-limited settings will require **offline sync capabilities**.
- **Blockchain integration** was limited to Layer 1 (Ethereum testnet). Future work could explore Layer-2 scaling (e.g., Polygon, Optimism) for even lower gas fees and better throughput.

Future enhancements may also include:

- Integration of **AI-based plugin recommendation systems** based on user history or sensor trends.
- **Mobile versions** of the workflow tool for factory floor accessibility.
- **Cross-industry application**, adapting the architecture to other supply chains (e.g., tea, spices, apparel) with similar traceability and customization demands.

Final Remarks

This research culminates in a comprehensive platform that delivers tangible value to the coconut peat industry while simultaneously contributing to the broader field of software engineering for supply chain systems. It not only introduces a scalable microkernel architecture but also aligns with global goals such as **sustainability, ethical sourcing, and industry digitalization**.

The system is not merely a technical artifact; it is a **transformation enabler** bridging the gap between real-world supply chain practices and the potential of modern computing. Its modularity, performance, and extensibility position it as a foundational step toward smarter, more transparent, and more inclusive supply chain ecosystems.

SUMMARY OF EACH STUDENT'S CONTRIBUTION

1. IT21308284 – Vithanage H.D

Component: Microkernel-Based Core System and Plugin Orchestration

Specific Contributions:

- Developed the **core execution engine** in Go based on microkernel architecture principles, enabling modular plugin registration and lifecycle control.
- Implemented **gRPC-based communication protocols** to connect the core system with plugins, the frontend, and IoT handlers.
- Integrated **K3s (lightweight Kubernetes)** for managing plugin orchestration, auto-recovery, scaling, and container fault isolation.
- Developed the **plugin manager module**, which supports runtime plugin registration, configuration loading, and isolation.
- Built real-time **observability dashboards** using Prometheus and Grafana to monitor plugin latency, health metrics, and system resource usage.
- Conducted **architecture evaluation** using ATAM and CBAM to validate trade-offs in scalability, performance, and cost-efficiency.

2. IT21291678 – Dehipola H.M.S.N

Component: Workflow Customization Tool & DSL Integration

Specific Contributions:

- Designed and implemented the **drag-and-drop visual workflow editor** using React.js and GoJS, enabling non-technical users to create and modify workflows dynamically.
- Developed the **Domain-Specific Language (DSL)** for the coconut peat supply chain, allowing users to define plugin order, conditions, and parameters through a human-readable syntax.
- Built the **DSL parser and translator**, which converts visual workflows into machine-interpretable Go configuration files.
- Integrated frontend and backend communication via gRPC to support real-time workflow generation and plugin registration.
- Implemented **form validation and error detection logic** to ensure consistency and correctness in DSL instructions.

3. IT21576966 – Weedagamaarachchi K.S

Component: Blockchain Integration & Smart Contract Optimization

Specific Contributions:

- Designed and implemented **Ethereum smart contracts** using optimized Solidity code, focusing on Factory, Registry, and State Machine patterns to manage stakeholders and shipment lifecycle.
- Conducted **gas usage profiling and optimization**, reducing deployment costs by ~19% and runtime transaction costs by ~14%.
- Applied advanced gas-saving strategies such as **mapping simplification, variable packing, and use of immutable variables**.
- Integrated smart contracts with the backend using **Web3.js** to ensure real-time logging of workflow milestones (e.g., grading results, EC validation, shipment approval).
- Evaluated **cyclomatic complexity of smart contracts** to quantify the impact of structural design on gas efficiency.

- Developed and tested contracts on **Ganache** using Hardhat and Remix, and ensured compatibility with plugin-generated runtime data.

4. IT21289484 – K.D.R. Manditha

Component: IoT Integration & Sensor-Driven Workflow Automation

Specific Contributions:

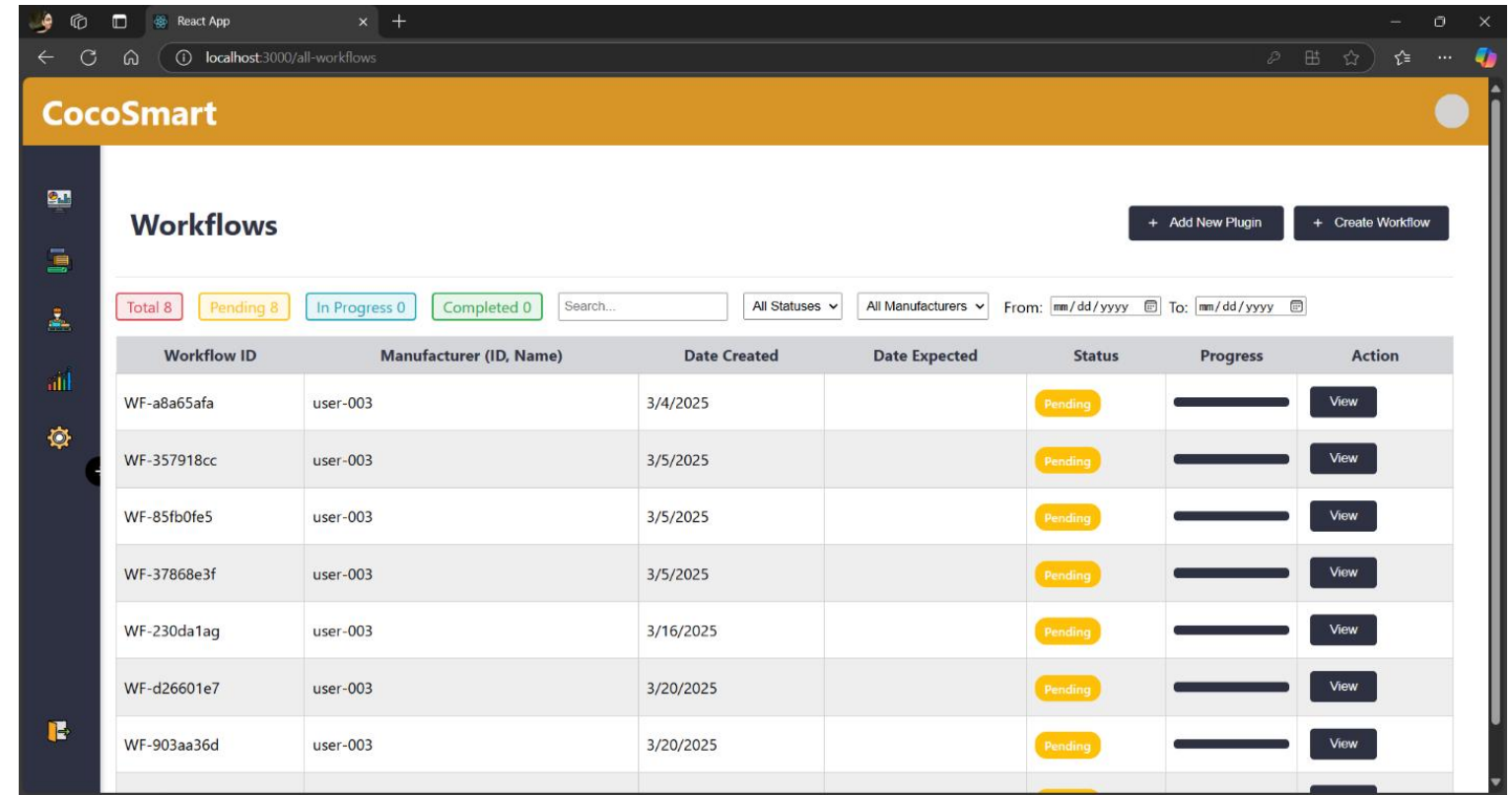
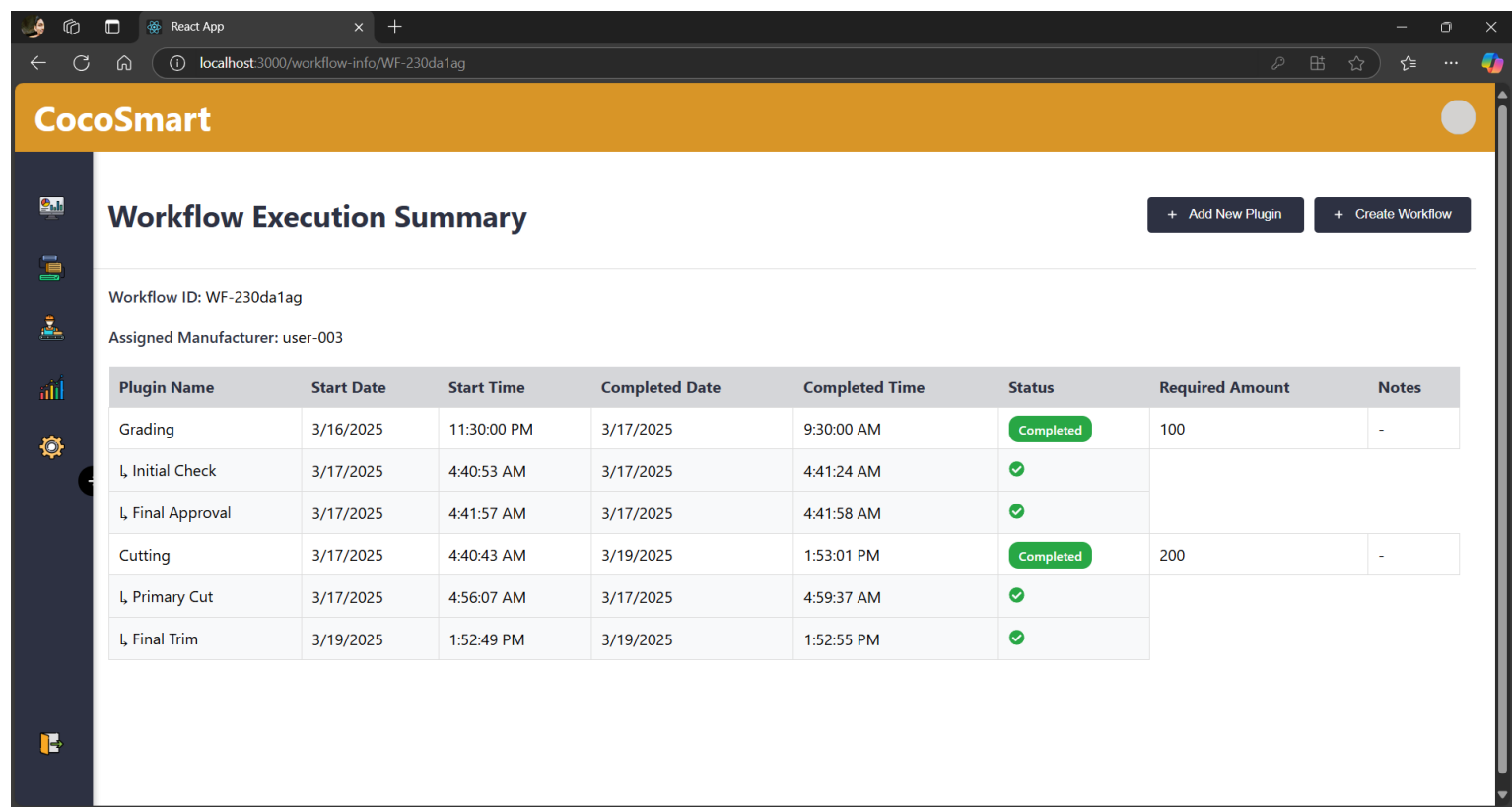
- Developed the **sensor data acquisition layer** using **ESP32** and **ESP32-CAM** modules to collect real-time EC, moisture, and visual data from production environments.
- Configured MQTT communication using **HiveMQ**, establishing real-time topic channels for EC-level monitoring and image-based grading.
- Created the **Node-RED logic** and MQTT listeners to trigger specific plugin actions (e.g., rewash commands) based on sensor input thresholds.
- Integrated IoT responses with the core system via gRPC to support **event-driven plugin execution** and autonomous workflow control.
- Validated sensor input accuracy and reliability through testing scenarios and implemented error-handling mechanisms for signal noise and data anomalies.
- Collaborated on real-time simulations where sensor feedback directly influenced workflow progression without human intervention.

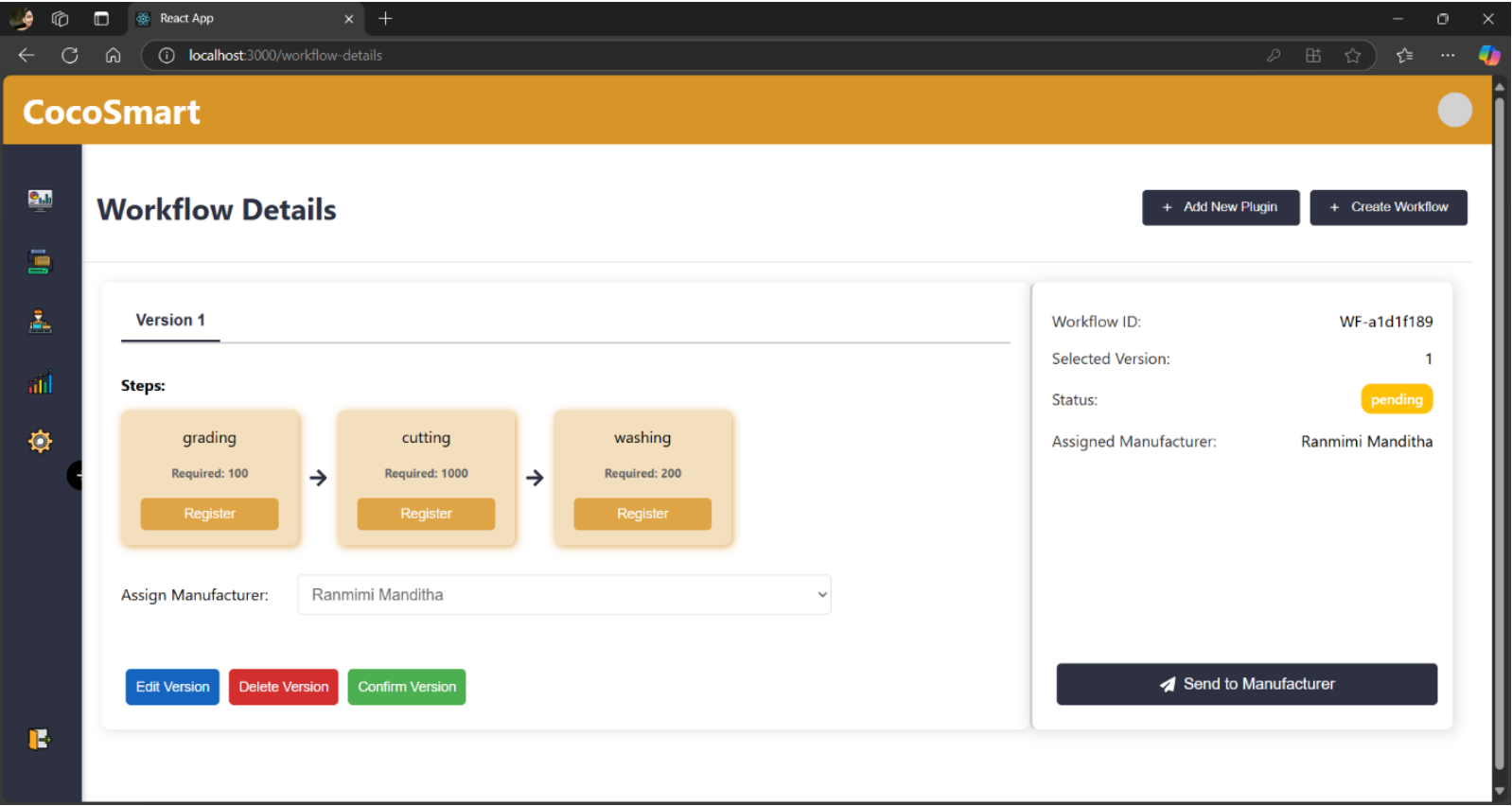
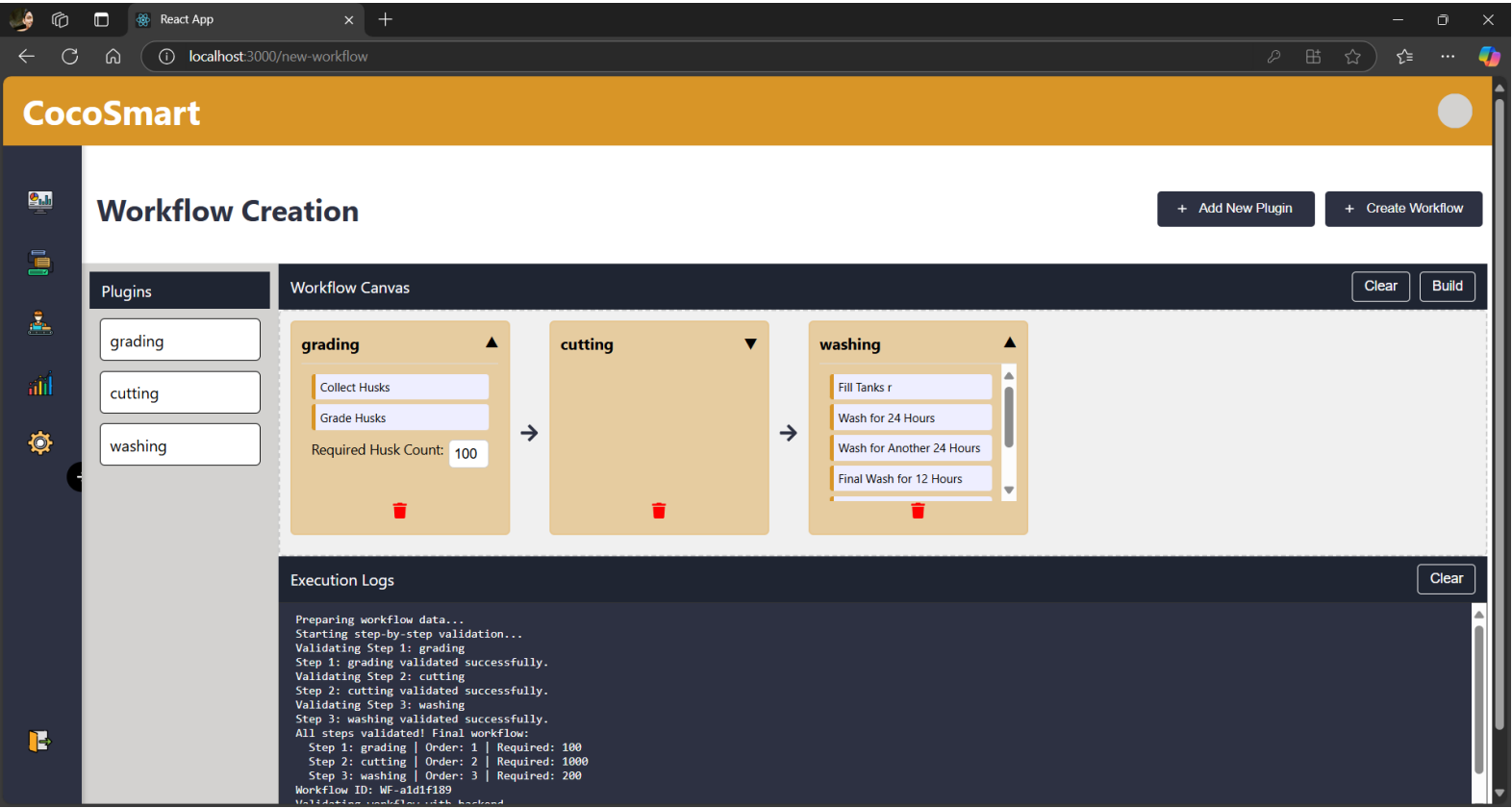
REFERENCES

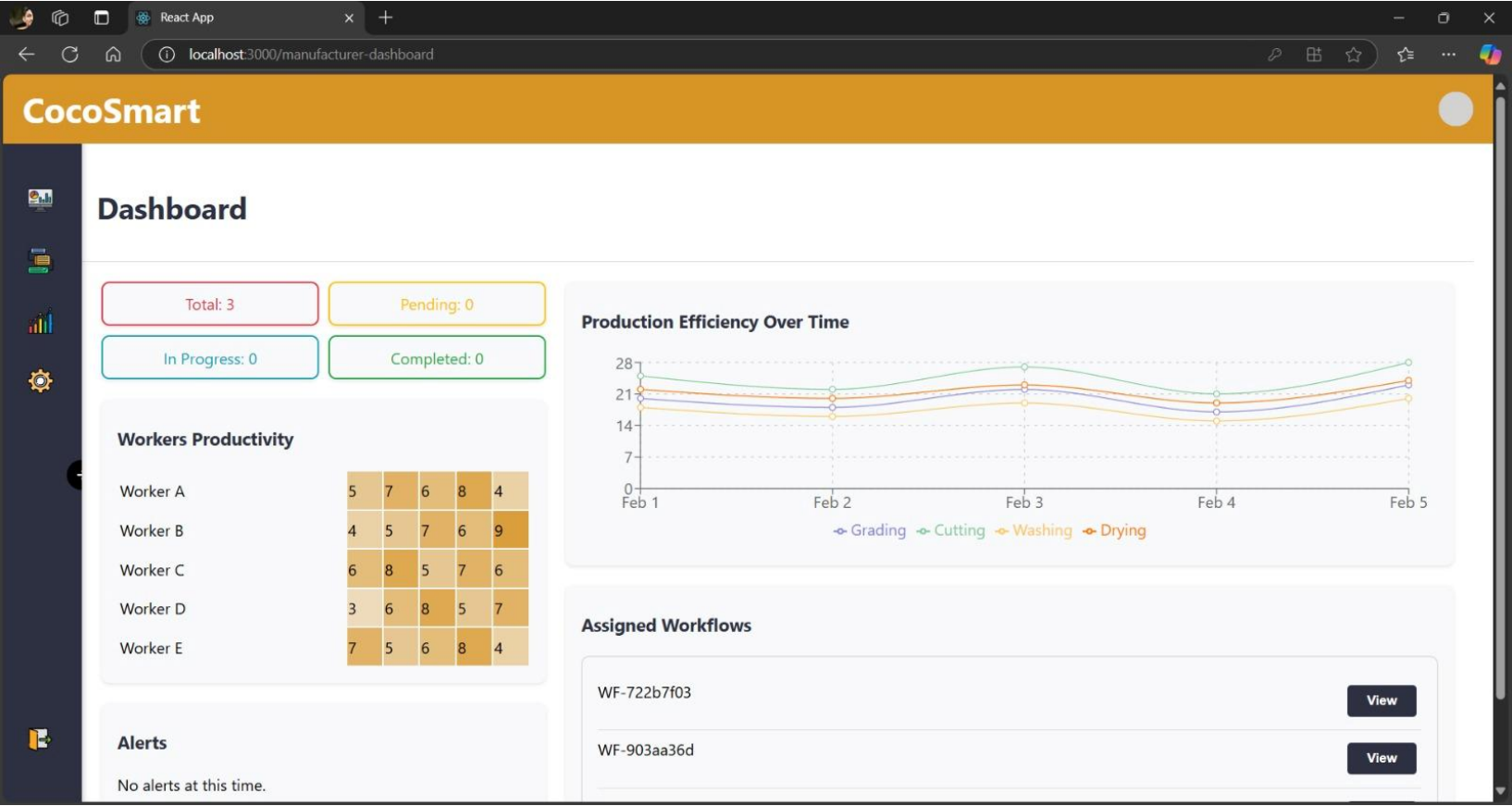
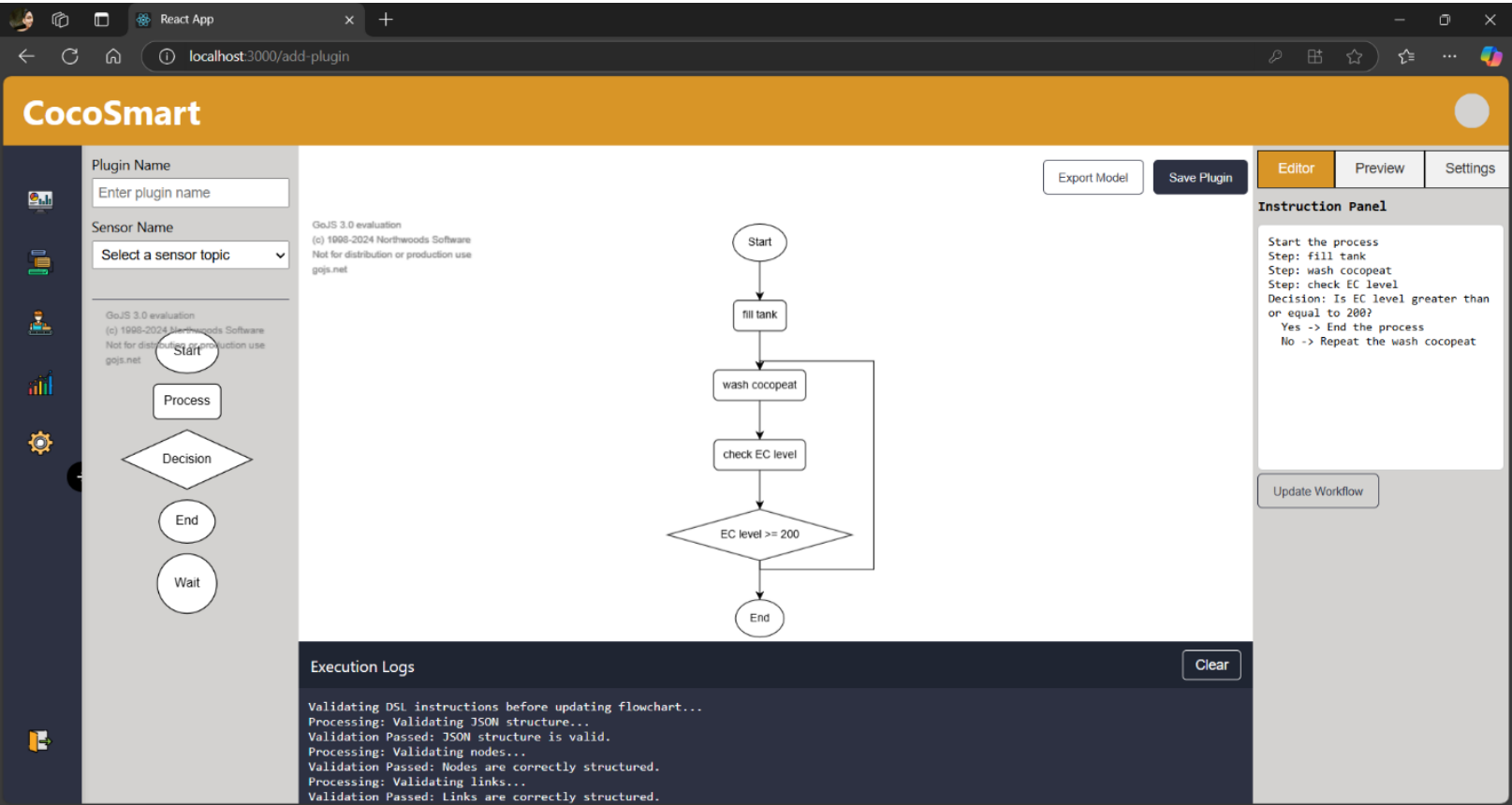
- [1] L. J, "On Microkernel Construction," in *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, 1995.
- [2] T. Bopp and T. Hampel, "A Microkernel Architecture for Distributed Mobile Environments," in *Proceedings of the 7th International Conference on Enterprise Information Systems (ICEIS)*, 2005.
- [3] D. R. A. S. S. S. S. J. & B. F. Raposo, "Industrial IoT Monitoring: Technologies and Architecture Proposal. Sensors," vol. 18, no. 10, p. 3568, 2018.
- [4] C. Jayawardena, I.-H. Kuo, E. Broadbent and B. A. MacDonald, "Socially Assistive Robot HealthBot: Design, Implementation, and Field Trials," *IEEE Systems Journal*, vol. 10, no. 3, pp. 1056-1067, 2016.
- [5] G. Wang and M. Xu, "Research on Tightly Coupled Multi-Robot Architecture Using Microkernel-Based, Real-Time, Distributed Operating System," in *Proceedings of the International Symposium on Information Science and Engineering (ISISE)*, 2008.
- [6] 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.
- [7] K. R. R. A. a. L. A. R. Goh, "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.
- [8] I. K. C. D. R. Q. S. E. B. a. B. A. M. C. Jayawardena, "Design, implementation and field tests of a socially assistive robot for the elderly: HealthBot Version 2," in *th IEEE RAS & EMBS International Conference on Biomedical Robotics and Biomechatronics (BioRob)*, Rome, Italy, 2012.
- [9] D. Ž. I. P. a. G. M. Milivoj Božić, "criptable graphical user interface engine for embedded platforms,," in *21st Telecommunications Forum (TELFOR)*, Belgrade, Serbia, 2013.
- [10] N. S. a. D. 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.
- [11] X. L. X. L. a. X. Z. T. Chen, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017.
- [12] B. D. Perera, M. P. S. Randunu, N. S. Wellappuli Arachchige, L. Rupasinghe, M. K. N. T. Kodithuwakkuge and C. Liyanapathirana, "Optimizing Gas Fees for Cost-Effective E-voting Smart Contracts on the Ethereum Blockchain," in *2023 5th International Conference on Advancements in Computing (ICAC)*, 2023.
- [13] N. Masla, J. Vyas, J. Gautam, R. N. Shaw and A. Ghosh, "Reduction in Gas Cost for Blockchain Enabled Smart Contract," in *2021 IEEE 4th International Conference on Computing, Power and Communication Technologies (GUCON)*, 2021.
- [14] L. Marchesi, M. Marchesi, G. Destefanis, B. Barabino and D. Tigano, "Design Patterns for Gas Optimization in Ethereum," in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, 2020.
- [15] D. Pramulia and B. Anggorojati, "Implementation and Evaluation of Blockchain-Based E-voting System with Ethereum and Metamask," in *2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, 2020.
- [16] G. Canfora, A. Di Sorbo, S. Laudanna, A. Vacca and C. A. Visaggio, "Profiling Gas Leaks in Solidity Smart Contracts," 2020.
- [17] R. K. G. a. A. S. Anjum, "Detection of Fire Using Image Processing Techniques with LUV Color Space," *Materials Today: Proceedings*, vol. 133, pp. 150-156, 2018.
- [18] S. Pandey, "Application of Image Processing and Industrial Robot Arm for Quality Assurance Process of Production," *International Journal of Emerging Trends in Engineering Research*, vol. 8, no. 7, pp. 3605-3609, 2020.
- [19] K. P. J. Hemachandran, "Image Processing in Agriculture," *International Journal of Computer Science and Information Technologies*, vol. 6, no. 6, pp. 5234-5237, 2015.

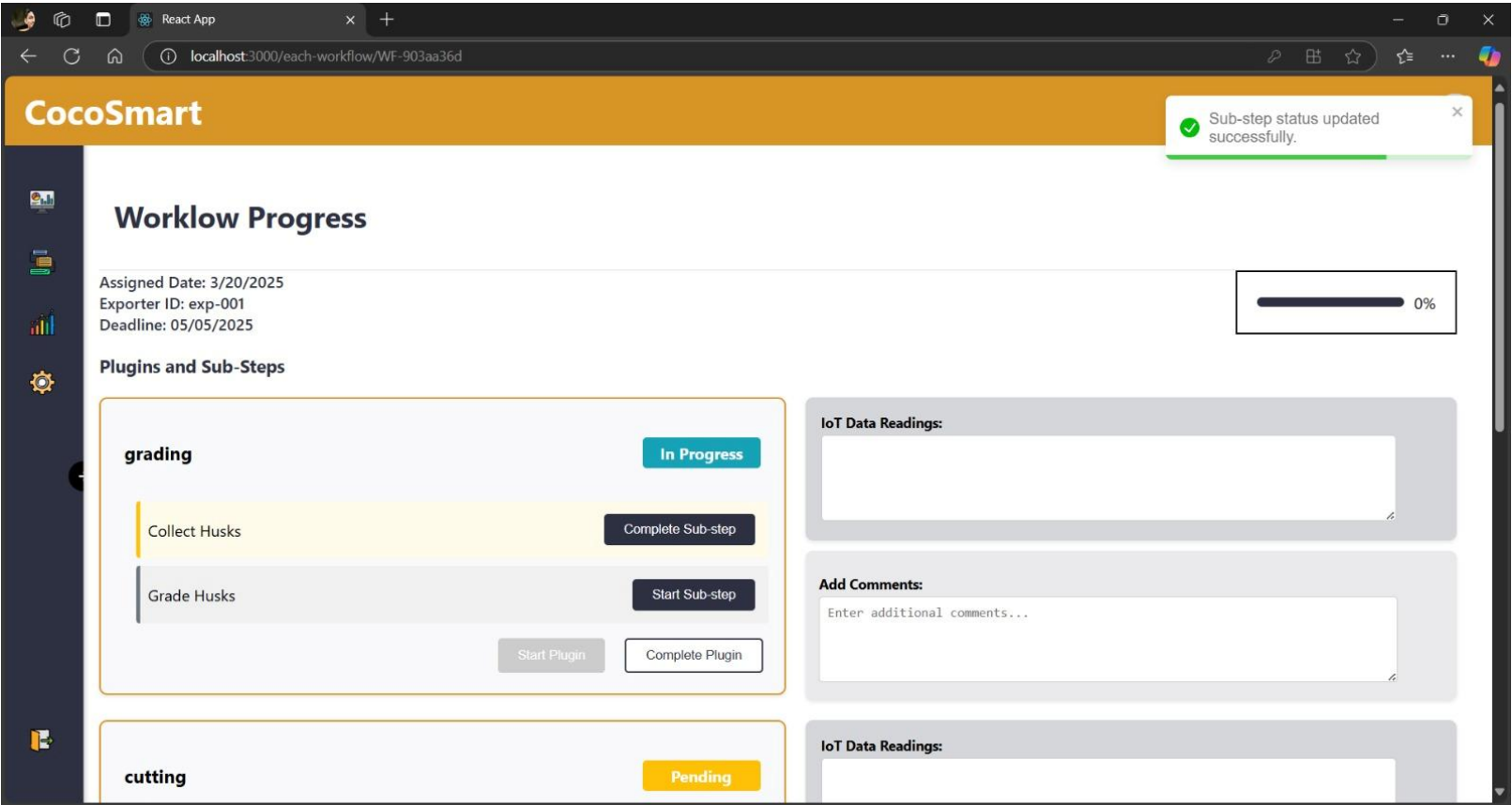
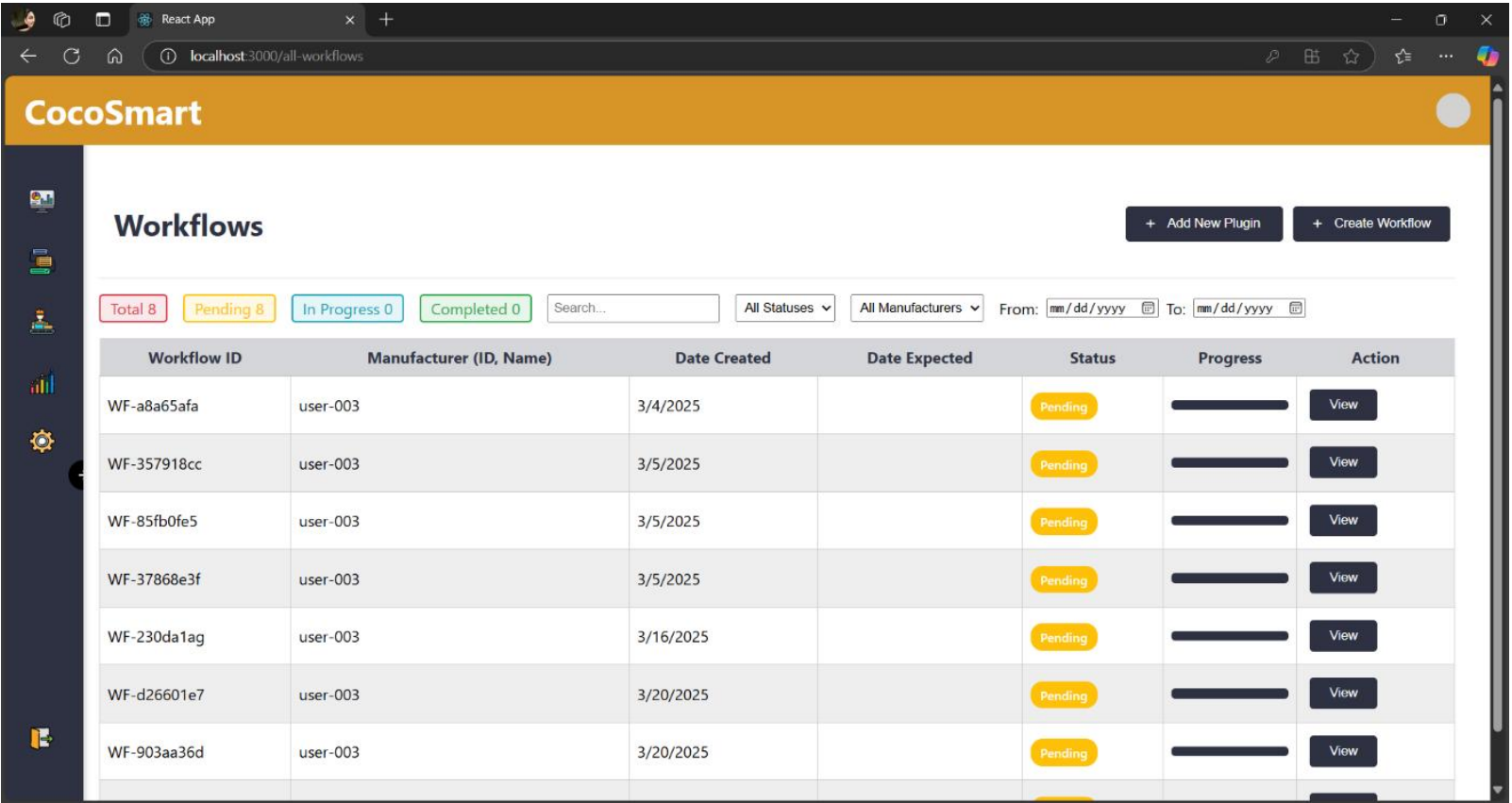
APPENDICES

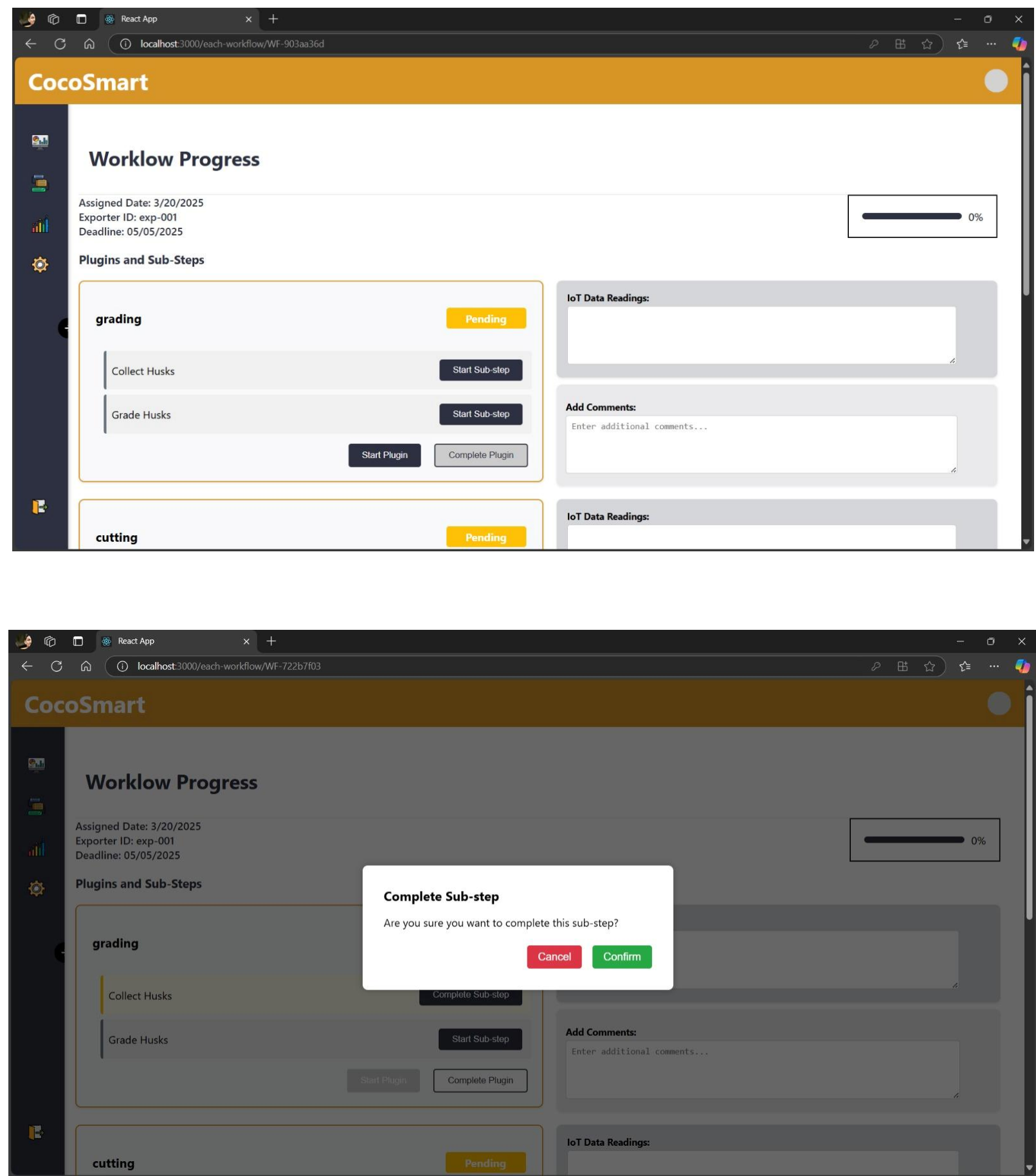
APPENDIX A: User Interfaces of the Workflow Customization Tool

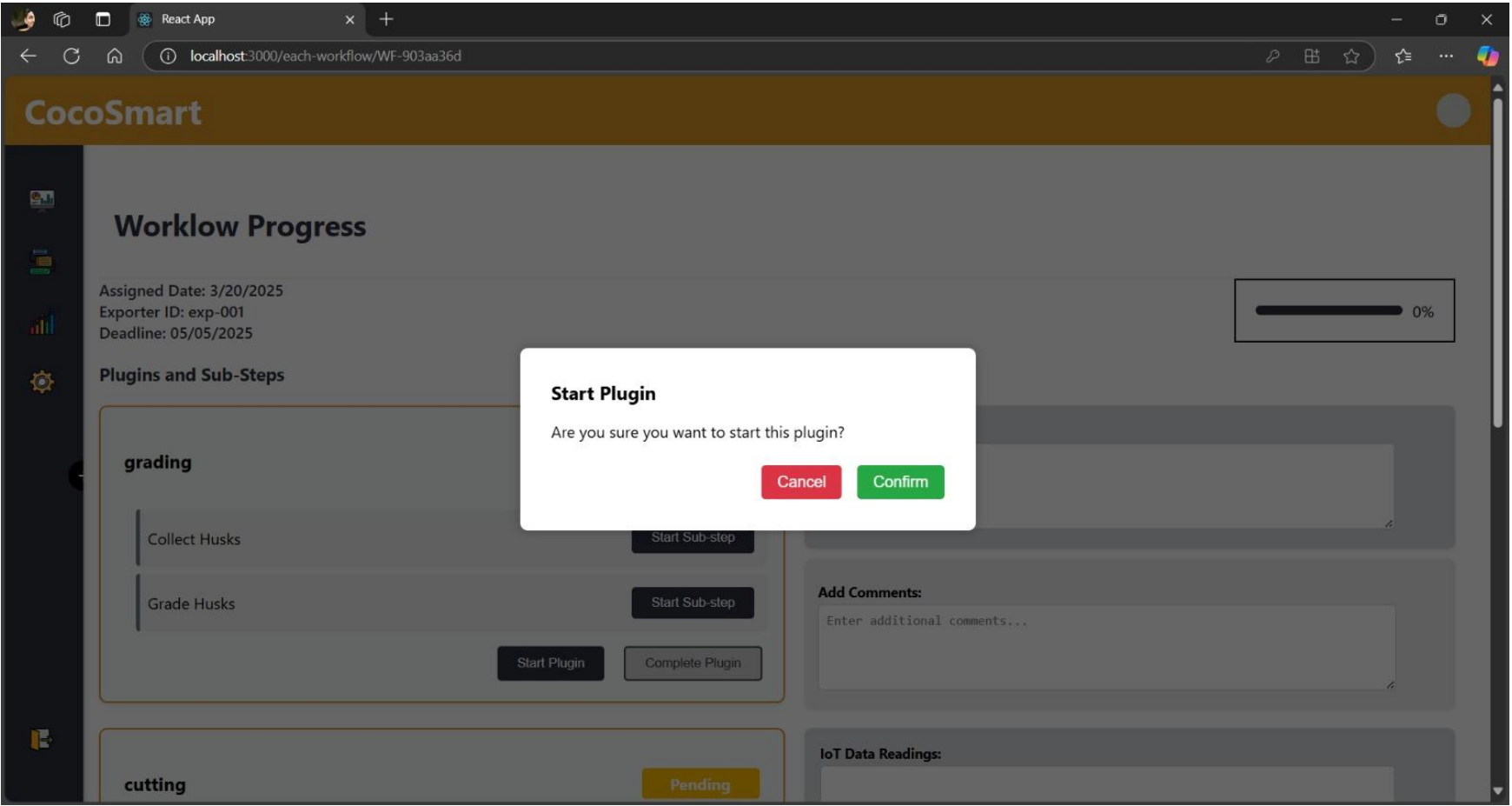












APPENDIX B: Code Snippets of the Plugin Creation Process

JS fileWriter.js

backend > controllers > JS 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',
66     };
67   } catch (error) {
68     // Handle error
69   }
70 }
```

JS pluginController.js

backend > controllers > JS pluginController.js > grpcFun > grpcFun

```
1 const NewPlugin = require('../models/Plugin');
2 const fs = require('fs');
3 const path = require('path');
4 const archiver = require('archiver');
5 const grpc = require('@grpc/grpc-js');
6 const protoLoader = require('@grpc/proto-loader');
7 const axios = require('axios');
8 const { updateFile } = require('../controllers/fileWriter');
9 const { generateFile } = require('../controllers/fileWriter');
10
11 // Save Plugin JSON
12 exports.savePlugin = async (req, res) => {
13   try {
14     const { plugin_name, nodes, links } = req.body;
15
16     if (!plugin_name || !nodes || !links) {
17       return res
18         .status(400)
19         .json({ success: false, message: 'Missing required fields' });
20     }
21
22     // Check if plugin already exists
23     const existingPlugin = await NewPlugin.findOne({ plugin_name });
24
25     if (existingPlugin) {
26       return res
27         .status(400)
28         .json({ success: false, message: 'Plugin already exists' });
29     }
30
31     const plugin = new NewPlugin({ plugin_name, nodes, links });
32     await plugin.save();
33     res
34       .status(201)
35       .json({ success: true, message: 'Plugin saved successfully' });
36   } catch (error) {
37     res.status(500).json({ success: false, message: error.message });
38   }
39 }
```

```

JS pluginController.js
backend > controllers > JS pluginController.js > grpcFun > grpcFun
40
41 // Fetch Plugin JSON by Plugin Name
42 exports.getPlugin = async (req, res) => {
43   try {
44     const plugin = await NewPlugin.findOne({
45       plugin_name: req.params.plugin_name,
46     });
47
48     if (!plugin) {
49       return res
50         .status(404)
51         .json({ success: false, message: 'Plugin not found' });
52     }
53
54     res.json({ success: true, data: plugin });
55   } catch (error) {
56     res.status(500).json({ success: false, message: error.message });
57   }
58 };
59
60 // Load the protobuf
61 const PROTO_PATH = path.join(__dirname, '../grpc-node-server/plugin.proto');
62 const packageDefinition = protoLoader.loadSync(PROTO_PATH, {
63   keepCase: true,
64   longs: String,
65   enums: String,
66   defaults: true,
67   oneofs: true,
68 });
69
70 const pluginProto = grpc.loadPackageDefinition(packageDefinition).plugin;
71
72 // gRPC Client setup
73 const client = new pluginProto.MainService(
74   '0.0.0.0:50051',
75   grpc.credentials.createInsecure()
76 );

```

```

JS pluginController.js
backend > controllers > JS pluginController.js > processAll > processAll > folderPath
109
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);

```

```

JS pluginController.js
backend > controllers > JS 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 };

```

```
JS pluginController.js X
backend > controllers > JS 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) => {
```



```
JS fileWriter.js X
backend > controllers > JS fileWriter.js > updateFile
1  const fs = require('fs');
2  const Port = require('../models/Port');
3  const path = require('path');
4
5  // Endpoint to update the file
6  exports.updateFile = async (content, filePath) => {
7    return new Promise((resolve, reject) => {
8      if (!content) {
9        return reject(new Error('Content is required'));
10     }
11
12     // Write the updated content to the file
13     fs.writeFile(filePath, content, (err) => {
14       if (err) {
15         console.error(err);
16         reject(new Error('Error updating the file'));
17       } else {
18         resolve({ message: 'File updated successfully!' });
19       }
20     });
21   });
22 };
23
24 exports.generateFile = async (
25   goFileContent,
26   plugin_name,
27   sensor_name,
28   userRequirement,
29   execute_logic,
30   save_path
31 ) => {
32   try {
33     // Validate input
34     if (
35       !goFileContent ||
36       !plugin_name ||
37       !sensor_name ||
```

APPENDIX B: Code snippet of the core system

Code snippet of the gRPC Gateway Function

```
func (s *Server) ClientFunction(ctx context.Context, req *pb.ClientRequest) (*pb.ClientResponse, error) {
    //get the plugin name and the plugin port number and the plugin name from the mongodb
    collection := mongo.MongoClient.Database("test").Collection("port")
    filter := bson.D{
        {Key: "plugin", Value: req.PluginName},
        {Key: "status", Value: true},
    }
    var result bson.M
    err := collection.FindOne(context.Background(), filter).Decode(&result)
    if err != nil {
        log.Fatalf("Error while fetching the plugin details: %v", err)
    }
    pluginPort := strconv.Itoa(int(result["port"].(int32)))
    pluginName := req.PluginName

    address := fmt.Sprintf("%s-plugin-service.default.svc.cluster.local:%s", pluginName, pluginPort) //kube
    conn, err := grpc.Dial(address, grpc.WithTransportCredentials(insecure.NewCredentials()))
    if err != nil {
        log.Fatalf("Failed to connect to backend service: %v", err)
    }
    defer conn.Close()

    // create a gRPC client for the backend service
    backendClient := pb.NewPluginClient(conn)

    //decide which action and call the backend service
    action := req.Action
    if action == "register" {
        backendResp, err := backendClient.RegisterPlugin(ctx, &pb.PluginRequest{
            PluginName: req.PluginName,
            WorkflowId: req.WorkflowId,
            UserRequirement: req.UserRequirement,
        })
    }
}
```

```

    })
    if err != nil {
        return nil, err
    }
    return &pb.ClientResponse{
        Success: backendResp.Success,
        Message: backendResp.Message,
    }, nil
} else if action == "execute" {
    backendResp, err := backendClient.ExecutePlugin(ctx, &pb.PluginExecute{
        PluginName: req.PluginName,
        WorkflowId: req.WorkflowId,
    })
    if err != nil {
        return nil, err
    }

    // If execution is successful, send data to the blockchain
    if backendResp.Success == true {
        blockchainMain() // Pass the results to the blockchain function
    }

    // Return the response to the client
    return &pb.ClientResponse{
        Success: backendResp.Success,
        Message: backendResp.Message,
        Results: backendResp.Results,
    }, nil
} else if action == "unregister" {
    backendResp, err := backendClient.UnregisterPlugin(ctx, &pb.PluginUnregister{
        PluginName: req.PluginName,
        WorkflowId: req.WorkflowId,
    })
    if err != nil {
        return nil, err
    }

```


Code snippet of the new plugin creation on runtime

```
func (s *NewPlugin) NewPluginCreate(ctx context.Context, req *pbv.NewPluginCreateRequest) (*pbv.NewPluginCreateResponse, error) {
    filename := req.FileName
    filedata := req.FileData
    savePath := filepath.Join("plugins", filename)

    // Ensure the directory exists
    dir := filepath.Dir(savePath)
    err := os.MkdirAll(dir, 0755)
    if err != nil {
        log.Printf("Failed to create directory: %v", err)
        return &pbv.NewPluginCreateResponse{
            Success: false,
            Message: "Failed to create directory",
        }, err
    }

    // Save the file
    err = os.WriteFile(savePath, filedata, 0644)
    if err != nil {
        log.Printf("Failed to save the file: %v", err)
        return &pbv.NewPluginCreateResponse{
            Success: false,
            Message: "Failed to save the file",
        }, err
    }

    //run the plugin.sh command file to unzip, install and run docker container
    cmd := exec.Command("/bin/bash", "plugin.sh")
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr
}
```

```
err = cmd.Run()

if err != nil {
    log.Printf("Failed to execute command file: %v", err)
    return &pbv.NewPluginCreateResponse{
        Success: false,
        Message: "Failed to execute command file",
    }, err
}

log.Printf("File %s uploaded successfully", filename)
return &pbv.NewPluginCreateResponse{
    Success: true,
    Message: "File uploaded and command executed successfully",
}, nil
}
```