



Enhancing Transparency in The Coconut Supply Chain Through a Software Engineering Approach

24-25J-313

Project Proposal Report

Vithanage H.D

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

Department of Information Technology

Sri Lanka Institute of Information Technology

Sri Lanka

August 2024

**A Novel Architectural Approach for Enhancing System
Availability and Plugin Management in a Plugin Architecture**

24-25J-313

Project Proposal Report

Vithanage H.D

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

Department of Information Technology


Sri Lanka Institute of Information Technology

Sri Lanka

August 2024

DECLARATION

We declare that this is our own work and this proposal does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any other university or Institute of higher learning and to the best of our 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.

Name	Student ID	Signature
Vithanage H.D	IT21308284	

The supervisor/s should certify the proposal report with the following declaration.

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

Signature of the Supervisor: Mr.Vishan Jayasinghearachchi	Date: 2024-08-23
Signature of the Co-Supervisor: Mr. Jeewaka Perera	Date:2024-08-23

ABSTRACT

This study aims at developing and assessing a novel architectural approach of a microkernel structure for increasing the system availability and the dynamic plugin management to be implemented in the coconut peat manufacturing process supply chain. This research seeks to solve pertinent problems in the SCM domain such as high availability of the system, real-time adaptability and the integration and incorporation of IoT and Blockchain technologies which enable transparency. The research problem focuses on the possible manner of implementing microkernel architecture, involving the issues of handling plugins and sustaining an operation with a constant change of workflows as well as various clients.

The study employs the design and implementation methodology, where the microkernel architecture is developed with an aim of modularity, flexibility, scalability and customizability. Key components include a Core System for the management of the overall workflow, Communication with Plug-Ins, Security System, and IoT Sensor & Blockchain integration among others, it has also a flexible Plug-In System for the control of several levels of coconut peat production process. The design also includes a ‘workflow definition tool’ which allows the definition of the workflow and a blockchain-based system for guaranteeing the transparency of each step in the supply chain.

The findings show that microkernel architecture can efficiently handle dynamic plugins, with the system showing easily resistance with any failed plugin and at the same time, can maintain high availability.

Therefore, the report proves the applicability of a microkernel architecture for increasing system availability and flexibility in SCM. This has presented this approach as a strong, elastic and clear system with capability of satisfying the complicated and dynamic need of the evolved industrial supply chain especially the coconut peat production.

Keywords: - supply chain, Coconut peat, microkernel architecture, dynamic plugin management, high availability

Table of Contents

ABSTRACT.....	4
LIST OF FIGURES.....	6
LIST OF TABLES	6
LIST OF ABBREVAITION.....	6
INTRODUCTION	7
BACKGROUND & LITERATURE SURVEY.....	8
Fundamental Concepts of Microkernel Architecture	8
System Availability in Microkernel Architecture.....	8
Plugin Management in Microkernel Architecture.....	8
Challenges & Considerations	9
Conclusion & Future work.....	9
RESEARCH GAP	10
Introduction	10
System availability in Microkernel architecture	10
Plugin Management in Microkernel architecture.....	10
RESEARCH PROBLEM	11
Research Question:.....	11
Explanation:	11
OBJECTIVES	12
Main Objective.....	12
Specific Objectives.....	12
METHODOLOGY.....	13
Architectural Trade-off Analysis	13
System Architecture	14
Proposed Approach	16
Technology Stack	16
Timeline & Work Breakdown Structure.....	17
Anticipated Conclusion.....	18
PROJECT REQUIREMENTS	18
Functional Requirements	18
User Requirements	19
System Requirements.....	19
Non-Functional Requirements	19
Use Cases	20
Test Cases.....	20
REFERENCES	22
APPENDIX.....	23

LIST OF FIGURES

Figure 1: [System Architecture](#)..... 14

Figure 2: [Timeline & Work Breakdown Structure](#) 17

Figure 3: [Timeline & Work Breakdown Structure](#) 17

LIST OF TABLES

Table 1: [Architectural Trade-off Analysis](#) 13

LIST OF ABBREVAITION

- SCM – Supply Chain Management
- ERP – Enterprise Resource Planning
- WBS – Work Breakdown Structure
- IOT – Internet of Things
- CAGR – Compound Annual Growth Rate
- IPC – Inter-Process Communication

INTRODUCTION

Coco-peat, also known as coir pith, is an adaptable byproduct of the coconut industry, known for its extensive use in horticulture as a soil substitute and soil conditioner. Also, its used as dry oil and lubricant absorbent in industrial settings and as high-water absorption material for animal bedding as well. This environmentally friendly product has earned significant traction in recent years due to its beneficial properties such as high-water retention, aeration, and biodegradability. Sri Lanka is recognized as the fourth-largest producer of coconuts globally and contributes greatly to this sector, with an annual production of approximately 2.8 to 3 billion coconuts [1]. A large portion of these coconuts is processed into value-added products like coco-peat, positioning Sri Lanka as a key actor in the global coco-peat market.

The global market for coco-peat is growing rapidly and reflecting the acceptance and demand in various agricultural applications. The market was valued at USD 2.27 billion in 2022 [2]. Projections indicate that this market size will reach USD 3.8 billion by 2031, driven by a compound annual growth rate (CAGR) of 4.4% [3]. This growth highlights the importance of coco-peat in sustainable agriculture and its capability to support global horticultural practices and fields.

In today's global economy, supply chain transparency and traceability are more important than ever, especially in the agricultural sector where customer demand for products supplied ethically and sustainably is rising. The processing of coconut peat, in particular, involves a number of parties, including exporters, clients, manufacturers, and local suppliers. For this reason, the coconut business is a good candidate for technological involvement to increase transparency and reliability. Conventional supply chain management techniques are frequently disjointed and do not integrate well enough to allow for real-time tracking and accountability. In order to solve these problems, this study suggests a novel software engineering strategy that focuses on implementing novel architecture approach based on microkernel architecture in an Enterprise Resource Planning (ERP) system for the coconut peat supply chain. This architecture improves system availability and optimizes plugin management.

Microkernel architecture or Plugin architecture has emerged as a major approach in modern software engineering specifically for the systems involving in high modularity, flexibility and customizability [4]. The microkernel's design principles separate the core functionality of the system from the addition functionalities of developers or end users in the form of extensions or plug-ins [4]. These plugins can dynamically add or remove based on the requirements. However, the architecture processes some challenges related to system stability and efficient plugin management. As for a supply chain the need for robust and adaptable system architecture is important. This proposal report will focus on improving the system availability and managing plugin effectively in microkernel architecture.

BACKGROUND & LITERATURE SURVEY

Fundamental Concepts of Microkernel Architecture

The microkernel architecture or plugin architecture is characterized by its simple core which handles only the most important functions to operate the system. Such as inter-process communication (IPC) and basic scheduling, and load handling. While the other additional functions/ services run in user space as independent plugins. This separation is crucial for achieving high system availability as it sets apart failures to individual plugins, thus preventing a single failure from crashing the whole system. Liedtke's seminal work on microkernel arguing for a design that minimizes the core's footprint to improve performance and reliability [5].

System Availability in Microkernel Architecture

System availability is an important metric in any system. It is the ability to remain operational and responsive. However, the dependency of the stable core may introduce risks. If the core fails, the entire system would be inoperable even the plugin. Several studies have shown that microkernel architecture inherently supports high availability. For instance, the paper "A microkernel architecture for distributed systems" highlights how the isolation system components in user space can prevent a system-wide failure. However, the study also shows the potential performance overhead introduced by frequent IPC which could impact system performance in high traffic environment [6].

Recent advancements have made cloud-native technologies such as containerization and orchestration tools like Kubernetes to further improve the system availability. These technologies can be used to ensure even in heavy load or partial system failures, the system availability by managing service redundancy and traffic load distribution [7].

Plugin Management in Microkernel Architecture

Management of the Plugins includes enabling the dynamic loading, execution and unloading in the architecture is crucial factor to consider as its directly related with process. This flexibility is very important for supply chain management where the workflow and process may frequently change due to varying client requirements. The process of loading, execution and unloading is managed through a well-defined plugin interface that ensures consistency across different plugins.

In his work "On microkernel construction" Liedtke describes the difficulties arising in microkernel-based systems due to IPC between different processes, with overarching stress on the efficient messaging schemas. A Microkernel must ensure that the IPC system offered incorporates the dynamic aspect of plugin loading and unloading the IPC system must be scalable in a way that it ensures seamless communication between the core and its plugins [5].

The use of formal methods to verify plugin interactions with the core system is another crucial factor. This verifies that the plugins do not introduce inconsistencies or security vulnerabilities into whole system [8].

Challenges & Considerations

While microkernel architecture offers many advantages if not managed efficiently, challenges will arise like communication overhead which can lead to performance bottlenecks. Also verifying the compatibility of plugins specifically when they are developed independently requires thorough testing and communication protocols. The research on “tightly coupled multi-robot architectures using microkernel based real-time distributed operating systems” also stresses the challenges of managing real-time limits in microkernel environments. Although the study focuses on robotics, the principals of real-time management are relevant to any system where timing and performance are important [9].

Conclusion & Future work

As per the literature review presented that microkernel architecture inherent modularity and flexibility are well compatible for systems requiring high availability and dynamic plugin management. However, addressing the challenges of communication overhead and ensuring real-time performance remains critical problems for future research. Further research could open the door to optimize the communication mechanisms within the architecture and developed standardized frameworks for plugin management that ensure computability and performance across different environments.

RESEARCH GAP

Introduction

The Microkernel architecture or Plugin architecture has gained major attention due to its modularity, minimalism and extensibility quality attributes. Despite its pros certain problems/ challenges persist, specifically in the area of system availability and plugin management. While existing literature provides a foundation for understanding microkernel architecture several gaps remain unfilled in the context of modern, dynamic and distributed computing environments.

System availability in Microkernel architecture

1. Redundancy and Failover mechanisms:

Although redundancy and failover have been considered with regard to distributed systems and cloud, there is no study that reviews how these strategies can be easily incorporated with the use of microkernel architecture. There is very little research addressing microkernel architecture, most work is done on monolithic or monolithic/hybrid systems where core services are not isolated as in microkernel architecture. Thus, there is a strong desire to find out how in its realization, a microkernel can provide native high availability with distributed redundancy without additional and significant overhead.

2. Health monitoring and Auto-Recovery:

Present-day publications on health monitoring and auto-recovery in such a structure seem to suffer from insufficient Advanced information that can be used for large scale realistic applications. These concerns are investigated in numerous works, often within more simplistic settings, where requirements such as real-time or high availability are not considered. Moreover, automated recovery in microkernel-based systems and in environments where dynamic plugins are utilized is an area of interest left unaddressed.

Plugin Management in Microkernel architecture

1. Dynamic Loading and Unloading:

While the dynamic management of plugins in microkernel systems has been explored in several works, the solutions presented in the process often fail to provide the necessary implementation details and frequently do not contain any real-life validation. The overwhelming majority of existing works concerns rather limited and relatively less dynamic plugin management when the plugins are loaded with an application and do not change frequently. There is a clear gap in research focused on dynamically loading, unloading and updating plugins in real-time especially environments where plugins must be replaced or modified frequently to meet changing requirements.

2. Performance Overhead:

Impact on the performance when managing a larger number of plugins in a microkernel system is not well clear. Some studies try to measure the overhead introduced by the plugin management, but often limited to specific case studies. There is a gap that remains particularly when traffic of the plugins is high and how to balance load or resources.

RESEARCH PROBLEM

Research Question:

How can a microkernel architecture be designed and implemented to enhance system availability and manage dynamic plugins effectively, while ensuring minimal performance overhead?

Explanation:

Microkernel architecture by design offers modularity, flexibility and customizability compared to monolithic architecture. But the dynamic nature of supply chain environment where services or processes frequently change establishes significant challenges to traditional microkernel implementations. This challenge includes maintaining system availability and efficiently managing plugins (which may need to be loaded, unloaded, or updated in real-time) while ensuring smooth performance.

In addition, the system is becoming more complex due to integrating with blockchain technology and IOT technology which complicates the architecture. Because of this it requires to find innovative solutions to maintain system integrity, security and maintain the performance without failures.

This research aims at presenting solutions to these problems by proposing a novel architectural approach to enhances system availability and plugin management within a microkernel system and the management of the plugins with focus on the dynamic and distributed architecture of the modern computing landscape. The aim is to put forward and discuss the architecture which should enable us to solve these complex issues, providing a robust, scalable and flexible solution.

OBJECTIVES

Main Objective

To design and implement a novel architecture that enhances the system availability and facilitates efficient plugin management in a distributed computing environment, while minimizing performance overhead and increasing the system stability.

Specific Objectives

1. Design and Development of the core system:

To design a core microkernel system that efficiently handles communication, traffic and system monitoring, ensuring high availability of the system even in the event of component failures.

2. Implementation of Dynamic plugin management:

To implement dynamic plugin management that allows for real-time loading, execution, unloading and updating of plugins with minimal performance impact to the system by ensuring that the system can adapt to changing requirements without downtime.

3. Integration of Modern technologies:

To integrate and explore the use of blockchain technology and IOT technology to enhance the transparency, security and efficiency of system operations.

4. Evaluation of System performance:

To analyze the performance of the proposed novel architecture, especially in distributed and real-time environment. Identifying potential bottlenecks and optimizing efficiency.

METHODOLOGY

Architectural Trade-off Analysis

Attribute	Monolithic Architecture	Microkernel Architecture	Microservices Architecture
Modularity	Low. All components are tightly integrated, making the system less modular.	Medium. Core functionalities are centralized, while plugins add modularity.	High. Each service is an independent module, allowing for high modularity.
Scalability	Low to Medium. Scaling requires scaling the entire application.	Medium. Core can be scaled independently of plugins, but scaling is limited to the kernel's capabilities.	High. Services can be independently scaled based on demand.
Maintainability	Low. Changes require rebuilding and redeploying the entire system.	Medium. Core and plugins can be maintained independently, but integration may become complex.	High. Independent services make maintenance easier and less disruptive.
System Availability	Medium. A failure in one part can potentially bring down the whole system.	High. The core can recover from plugin failures without affecting other parts of the system.	High. A failure in one service does not impact the availability of others.
Performance	High. Direct communication within a single process.	Medium. Some overhead due to inter-process communication between core and plugins.	Medium to High. Network latency might affect performance but can be optimized.
Complexity	Low. Simple structure but can become complex as the system grows.	Medium. Core-plugin interaction adds some complexity, but it's manageable.	High. Managing multiple services and their interactions adds significant complexity.
Deployment Flexibility	Low. Entire system must be redeployed for any change.	Medium. Plugins can be deployed or updated without affecting the core.	High. Each service can be deployed independently, allowing for continuous delivery.
Flexibility & Extensibility	Low. Hard to add new features without impacting the entire system.	High. Easy to add new plugins or replace existing ones without disrupting the core.	High. New services can be added or existing ones replaced with minimal impact.
Fault Isolation	Low. Failures can propagate across the system.	High. Plugin failures can be isolated from the core and other plugins.	High. Failures are contained within the affected service, not affecting others.
Development Speed	High initially, but slows down as the system grows.	Medium. Requires careful planning for core and plugin development.	Medium to Low. Initial setup is complex, but development speed increases with experience.
Technology Stack Flexibility	Low. Limited to the technologies chosen for the entire system.	Medium. Core uses a specific stack, but plugins could use different technologies.	High. Each microservice can use the technology stack best suited to its needs.

Attribute	Monolithic Architecture	Microkernel Architecture	Microservices Architecture
Security	Medium. Centralized security, but harder to manage as the system scales.	High. Core security is robust, and plugins can have separate security measures.	High. Each service can implement its own security, reducing attack surfaces.

Table 1 [4,10]

Summary

- **Monolithic Architecture:** Suitable for smaller or simpler systems where fast development and high performance are the main concern. But it suffers from low modularity, maintainability, and flexibility, making it less ideal for a complex, evolving supply chain scenario.
- **Microkernel Architecture:** Offers a good balance between modularity and performance. It allows for flexible and independent plugin management, making it a strong candidate for this supply chain scenario. However, it does introduce some complexity, particularly in managing core-plugin interactions.
- **Microservices Architecture:** Provides the highest modularity, scalability, and flexibility. It is well-suited for complex and distributed systems. But it also comes with high complexity in managing multiple services and their interactions, requiring a robust orchestration and monitoring infrastructure.

System Architecture

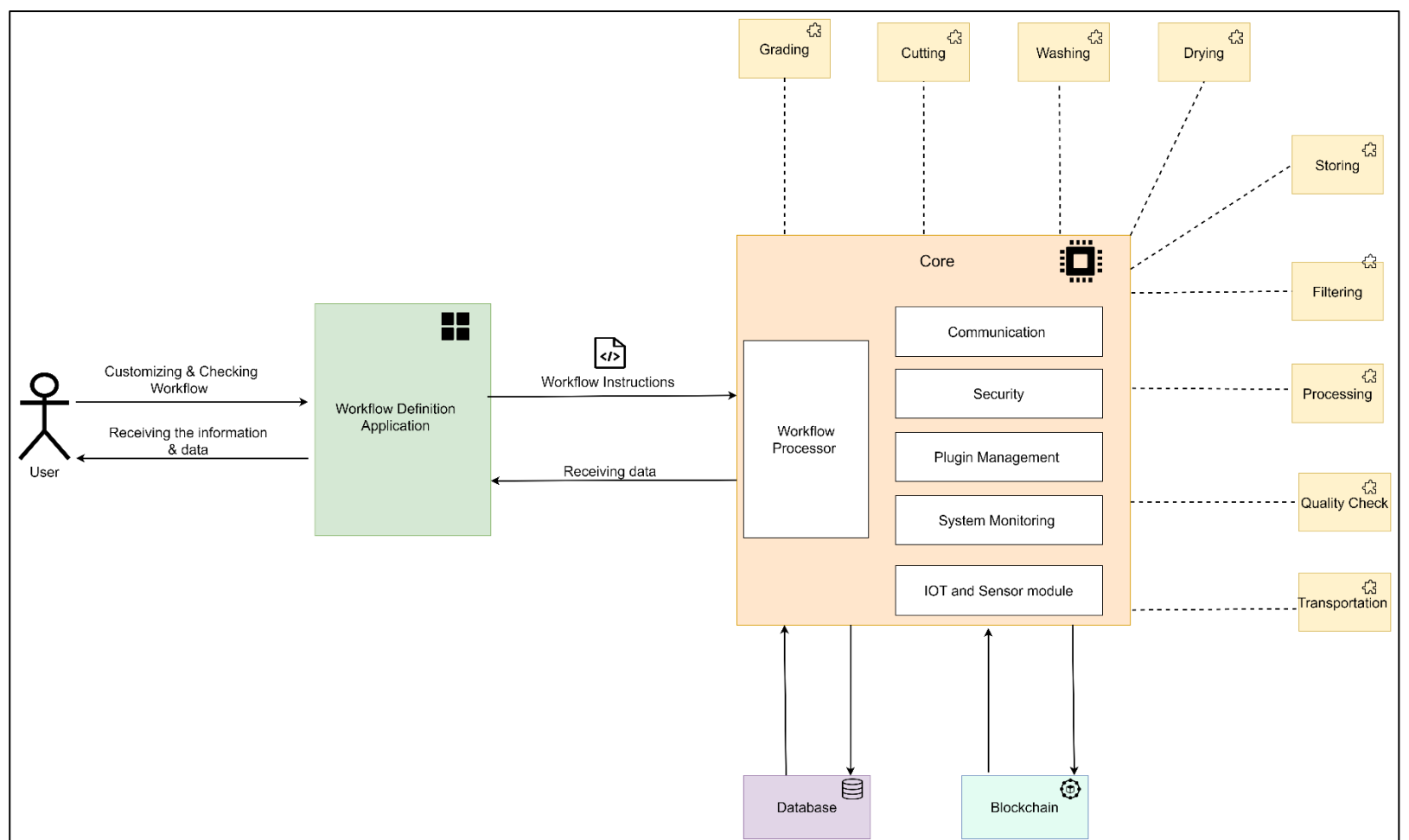


Figure 1

The methodology for this research project borders the approach to designing and implementing a microkernel-based system architecture that improves system availability and plugin management. The system diagram (as shown above) serves as the blueprint for achieving the main objective, which is to create a robust and flexible architecture capable of dynamically managing plugins in a supply chain system. This section details the tasks, technology stack, timeline, and anticipated outcomes of the project.

1. **Core System:**

- **Workflow Processor:** Responsible for executing workflows as defined by the Workflow Definition Application. It interprets the instructions given by the workflow definition application and communicates with plugins and coordinates their execution.
- **Communication Module:** Manages all interactions between the core system and the plugins. It handles API calls, ensures proper data flow, and maintains synchronization between various system components to provide smooth flow.
- **Security Module:** Confirms that all operations, especially those involving plugins, are conducted securely. It implements authentication, authorization, and encryption protocols.
- **Plugin Management:** This component dynamically loads, unloads, and manages plugins based on the requirements of the workflow. It ensures that only the necessary plugins are active, optimizing resource usage.
- **System Monitoring:** Continuously monitors the health and performance of both the core and plugins, providing real-time feedback and triggering recovery mechanisms in case of failures.
- **IoT and Sensor Module:** Interfaces with IoT devices to collect and process sensor data, which may be used by various plugins (not covered in this methodology).

2. **Plugins:**

- Represent various stages in the coconut processing workflow, such as **Grading, Cutting, Washing, Drying, Storing, Filtering, Processing, Quality Check, and Transportation**. Each plugin is responsible for a specific task and is controlled by the core system.

3. **External Components:**

- **Workflow Definition Application:** Allows users to define and customize workflows, which are then interpreted by the Workflow Processor.
- **Database:** Stores configuration data, plugin states, and logs, enabling persistent data management.
- **Blockchain:** Ensures transparency and verification across the supply chain (not covered in this methodology).

Proposed Approach

Project Execution Strategy

- **Phase 1: System Design and Architecture**
 - **Task 1:** Design the core system architecture, focusing on the Workflow Processor, Communication Module, Security Module, Plugin Management, and System Monitoring.
 - **Task 2:** Develop the plugin interface and standardize communication protocols between the core and plugins.
- **Phase 2: Implementation**
 - **Task 3:** Implement the core system components, ensuring that they can handle dynamic plugin management.
 - **Task 4:** Develop and integrate individual plugins for each stage of the coconut processing workflow.
 - **Task 5:** Implement the connection between external interface (e.g., the Workflow Definition Application) and ensure seamless integration with the core system.
- **Phase 3: Testing and Optimization**
 - **Task 6:** Conduct unit and integration tests for each core component and plugin, ensuring that they perform as expected.
 - **Task 7:** Optimize the system for performance, scalability, and security.
- **Phase 4: Deployment and Evaluation**
 - **Task 8:** Deploy the system in a simulated environment to evaluate its performance under various conditions.
 - **Task 9:** Collect feedback and refine the system based on real-world testing.

Sub-Tasks

- **Sub-Task 1:** Develop detailed API documentation for plugin communication.
- **Sub-Task 2:** Apply orchestration tools and containerization tools to ensure system stability and plugin management.
- **Sub-Task 3:** Establish a continuous integration/continuous deployment (CI/CD) pipeline to automate testing and deployment processes.

Technology Stack

- **Programming Language:** Golang (for core and plugin implementation)
- **Containerization:** Docker (for managing plugins as containers)
- **Orchestration:** Kubernetes or K3s (for managing core and plugin instances)
- **Database:** PostgreSQL or MongoDB (for storing configuration data and logs)
- **Security:** TLS/SSL for secure communication, OAuth 2.0 for authentication
- **Monitoring:** Prometheus and Grafana (for real-time monitoring and visualization)

Timeline & Work Breakdown Structure

Gantt chart:

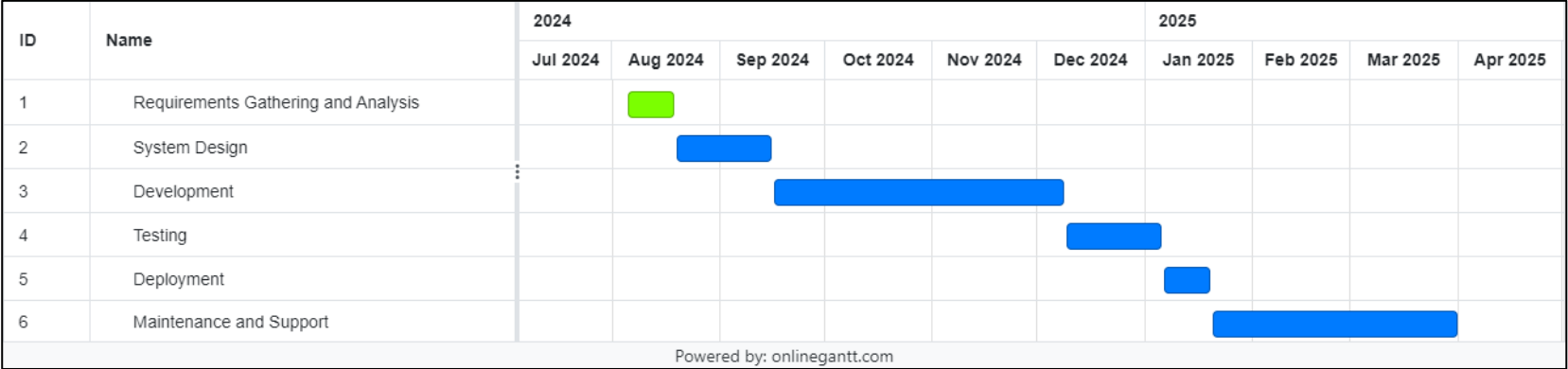


Figure 2

Workflow breakdown Structure:



Figure 3

Anticipated Conclusion

By the end of the project, the developed system will provide a robust novel architecture inspired by microkernel architecture capable of dynamically managing plugins, ensuring high system availability, and efficiently handling the coconut processing workflow. The real-world application of this system would be in automating and optimizing the coconut supply chain, reducing manual interventions, and enhancing overall transparency and efficiency.

The results will likely show the effectiveness of microkernel architecture in distributed systems, particularly in industries where dynamic processing and high availability are critical. The system could be adapted for other supply chains or industries with similar requirements.

PROJECT REQUIREMENTS

Functional Requirements

- **FR1: Core System Initialization**
 - The core system must initialize and load essential services, including inter-process communication (IPC) mechanisms, process scheduling, and system monitoring tools upon startup.
- **FR2: Plugin Management**
 - The core must dynamically load, and unload plugins based on system needs and user commands without requiring a system restart.
- **FR3: Plugin Communication**
 - The core must facilitate communication between plugins and between plugins and the core through a well-defined API.
- **FR4: Plugin Execution**
 - The core must execute plugins based on the workflow instructions, ensuring that each plugin receives the necessary input data and processes it accordingly.
- **FR5: Plugin Monitoring**
 - The core must monitor the status of each plugin, logging its performance and health status, and trigger auto-recovery mechanisms if a plugin fails.
- **FR6: Plugin Version Control**
 - The core must manage multiple versions of plugins, allowing users to switch between versions or revert to a previous version if necessary.
- **FR7: Error Handling**
 - The core must detect and handle errors within plugins, either by restarting the plugin, switching to a backup, or notifying the user of the issue.

User Requirements

- **UR1: Real-Time Feedback**
 - Users must receive real-time feedback on the status and performance of plugins, including notifications of any errors or failures.
- **UR3: Customizable Plugin Configuration**
 - Users must have the ability to configure plugins according to their needs, including setting parameters and defining execution conditions.
- **UR4: Access Control**
 - The system must provide role-based access control, ensuring that only authorized users can load, unload, or configure plugins.

System Requirements

- **SR1: Operating System Compatibility**
 - The core system must be compatible with major operating systems (Linux, Windows, macOS) and support both server and desktop environments.
- **SR3: Database Integration**
 - The core system must integrate with relational and non-relational databases to store configuration data, plugin states, and logs.
- **SR4: API Support**
 - The system must provide a RESTful API for external systems to interact with the core and manage plugins programmatically.

Non-Functional Requirements

- **NFR1: Performance**
 - The system must load, execute, and unload plugins with minimal latency, ensuring that plugin operations do not significantly impact overall system performance.
- **NFR2: Scalability**
 - The core system must be scalable, allowing it to manage an increasing number of plugins and users without deprivation in performance.
- **NFR3: Reliability**
 - The system must be highly reliable, with a target uptime of 99.9%, ensuring that the core and plugins are always available and functional.
- **NFR4: Security**
 - The system must secure all communications between the core and plugins using encryption and implement access controls to prevent unauthorized use.
- **NFR5: Usability**
 - The system's interface must be intuitive and easy to navigate, reducing the learning curve for new users and minimizing the need for extensive training.

- **NFR6: Maintainability**
 - The system must be modular and easy to maintain, with clear documentation and well-defined interfaces for plugins.

Use Cases

- **UC1: Load Plugin**
 - **Actors:** System Admin
 - **Description:** The user loads a plugin into the core system using the interface. The core checks the plugin's compatibility and loads it if valid.
 - **Preconditions:** The plugin file must be accessible and compatible with the core system.
 - **Postconditions:** The plugin is loaded and available for use.
- **UC2: Unload Plugin**
 - **Actors:** System Admin
 - **Description:** The user unloads a plugin from the core system. The core safely terminates the plugin's operations and releases any allocated resources.
 - **Preconditions:** The plugin must not be in use by any active processes.
 - **Postconditions:** The plugin is unloaded and no longer available.
- **UC3: Monitor Plugin**
 - **Actors:** System Admin
 - **Description:** The user monitors the status and performance of a plugin through the interface.
 - **Preconditions:** The plugin must be loaded and running.
 - **Postconditions:** The user receives real-time data on the plugin's performance and health.
- **UC4: Recover Failed Plugin**
 - **Actors:** System Admin
 - **Description:** The core system detects a plugin failure and automatically attempts to restart or switch to a backup plugin.
 - **Preconditions:** The plugin must be monitored by the core.
 - **Postconditions:** The plugin is either restarted or replaced with a backup.

Test Cases

- **TC1: Load Plugin Test**
 - **Objective:** Verify that the system correctly loads a plugin and makes it available for use.
 - **Input:** Valid plugin file.
 - **Expected Output:** The plugin is successfully loaded and listed in the interface.

- **TC2: Unload Plugin Test**
 - **Objective:** Verify that the system safely unloads a plugin and frees up resources.
 - **Input:** Command to unload an active plugin.
 - **Expected Output:** The plugin is unloaded, and resources are released.

- **TC3: Plugin Failure Recovery Test**
 - **Objective:** Test the system's ability to detect and recover from a plugin failure.
 - **Input:** Simulated plugin failure.
 - **Expected Output:** The system detects the failure, attempts recovery, and logs the event.

- **TC4: Performance Test**
 - **Objective:** Measure the system's performance when loading, executing, and unloading multiple plugins simultaneously.
 - **Input:** Multiple plugins with various loads.
 - **Expected Output:** The system maintains performance within acceptable limits.

REFERENCES

1. The Island, "From Tradition to Transformation: Sri Lanka's Coconut Export Revolution," The Island, 2023. [Online]. Available: <https://island.lk/from-tradition-to-transformation-sri-lankas-coconut-export-revolution/>. [Accessed: Aug. 20, 2024].
2. Sri Lanka Export Development Board, "Coconut Export Performance," Sri Lanka Export Development Board, 2024. [Online]. Available: <https://www.srilankabusiness.com/coconut/about/export-performance.html>. [Accessed: Aug. 20, 2024].
3. Sri Lanka Export Development Board, "Sri Lankan Cocopeat on the Global Stage: A Strategic Exploration of Local Dynamics, Market Entry, and Global Opportunities," Sri Lanka Export Development Board, 2023. [Online]. Available: <https://www.srilankabusiness.com/pdf/sri-lankan-cocopeat-on-the-global-stage-a-strategic-exploration-of-local-dynamics-market-entry-and-global-opportunities.pdf>. [Accessed: Aug. 20, 2024].
4. Mark Richards, "Microkernel Architecture," in *Software Architecture Patterns*, 2nd ed. Sebastopol, CA: O'Reilly Media, 2022. [Online]. Available: <https://learning.oreilly.com/library/view/software-architecture-patterns/9781098134280/ch04.html>. [Accessed: Aug. 20, 2024].
5. J. Liedtke, "On Microkernel Construction," in *Proc. SIGOPS '95*, ACM, 1995, pp. 237-249. [Online]. Available: http://crossmark.crossref.org/dialog/?doi=10.1145%2F224057.224075&domain=pdf&date_stamp=1995-12-03. [Accessed: Aug. 20, 2024].
6. T. Bopp and T. Hampel, "A Microkernel Architecture for Distributed Mobile Environments," in *Proc. 7th International Conference on Enterprise Information Systems (ICEIS 2005)*, Miami, FL, USA, 2005, pp. 151-156.
7. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *ACM Queue*, vol. 14, no. 1, pp. 70-93, Jan.-Feb. 2016. [Online]. Available: <http://queue.acm.org/detail.cfm?id=2898444>. [Accessed: Aug. 22, 2024].
8. Z. Qian, R. Xia, G. Sun, X. Xing, and K. Xia, "A measurable refinement method of design and verification for micro-kernel operating systems in communication network," *Digital Communications and Networks*, vol. 9, pp. 1070-1079, 2023.
9. G. Wang and M. Xu, "Research on tightly coupled multi-robot architecture using microkernel-based, real-time, distributed operating system," in *Proc. 2008 International Symposium on Information Science and Engineering (ISISE)*, Shanghai, China, 2008, pp. 978-0-7695-3494-7. DOI: 10.1109/ISISE.2008.80.
10. V. Benavente, C. Rodriguez, L. Yantas, R. Inquilla, I. Moscol, and Y. Pomachagua, "Comparative analysis of microservices and monolithic architecture," in *Proc. 14th IEEE Int. Conf. on Computational Intelligence and Communication Networks (CICN)*, Lima, Peru, 2022, pp. 177-182. DOI: 10.1109/CICN.2022.30.

APPENDIX