UNIVERSIDAD DE LOS ANDES

FACULTY OF ENGINEERING

Department of Systems and Computing Engineering

Information Technology and Software Construction Group (TICSw)

Architectural Foundations for Multistage Task Orchestration in Engineering Educational Systems

Esteban González Ruales

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Systems and Computing Engineering at Universidad de los Andes

December 25, 2025

Advisor: Prof. Camilo A. Escobar-Velásquez, Ph.D.

Abstract

This thesis presents the design of a distributed, fault-tolerant, and extensible system to orchestrate multistage task execution in heterogeneous environments. The system is motivated by the need for an infrastructure capable of processing dynamically routed tasks, initiated by users and subject to variation in outcome and resource requirements, within educational and experimental contexts. A message-driven architecture is proposed in which stateless processing nodes consume tasks from a message broker, retrieve associated artifacts, execute them in isolation, and forward results based on a predefined routing structure.

The system is implemented using an actor-based concurrency model, enabling strong fault isolation, structured recovery, and scalable execution through parallel disposable workers. Key architectural principles include the separation of orchestration and execution logic, the asynchronous transmission of messages between components, and the strict isolation between processing tasks. The platform currently supports containerized workloads and provides extension points for integrating new task types by modifying only the executor logic, without altering the orchestration layer.

Contents

List of Figures

Acknowledgments

I am deeply grateful to my family and my girlfriend for their constant support, patience, and encouragement throughout this journey. My sincere thanks also go to my thesis advisor, whose guidance and constructive input shaped this work and contributed significantly to my growth as an engineer. I also thank my friends, colleagues, and all those who gave their time, advice, and motivation along the way. Whether through technical guidance, thoughtful feedback, or simple words of encouragement, each contribution helped me move forward and overcome various challenges.

Disclaimers

This thesis acknowledges the use of artificial intelligence tools (e.g., ChatGPT) as a support mechanism during initial drafts, specifically to structure and clarify the author's own ideas. In some sections, particularly Chapter 3, the tool was also used to assist in researching, synthesizing, and drafting comparative descriptions of existing orchestration frameworks (e.g., Celery, Airflow, Argo Workflows, Temporal, and Conductor OSS). All content was critically reviewed, adapted, and integrated by the author. No material was included without full intellectual oversight and authorship responsibility.

Chapter 1

Introduction

Contemporary engineering and computer science education is increasingly grounded in project-based, experiential learning. Students are expected to not only design and model solutions, but also test, validate, and iterate on them using real or simulated execution environments. This pedagogical shift places significant demands on academic infrastructure, which must support both the complexity of modern technical tasks and the scalability required to serve growing student populations.

As part of this evolution, assignments often involve the processing of student-submitted artifacts, such as control models, software scripts, or test configurations, across a variety of processing environments. These environments may be physical, involving hardware devices or instrumentation, or virtual, involving emulated systems or automated testing frameworks. Task processing is rarely a one-step operation; instead, it often involves multiple stages, including but not limited to validation, simulation, and final execution. Furthermore, steps and processing requirements vary depending on the nature of the task and the risks or constraints associated with the target environment.

This increasing diversity in task workflows and processing contexts presents new challenges for academic institutions. Manual processes, informal task handoffs, and limited visibility into the state of submissions all contribute to inefficiencies and bottlenecks. As technical education continues to demand a more robust infrastructure for managing and processing student work, there is a growing need to explore architectural approaches that support automation, transparency, and extensibility across heterogeneous systems.

1.1 Problem Statement

In many university-level engineering programs, students are required to submit practical assignments that involve the execution of their work on physical laboratory infrastructure, such as robotic arms, 3D printers, or electrical grid simulators. These systems often represent real-world industrial setups and are used to provide hands-on experience with automation, control systems, or model validation.

The current workflow for the execution of these assignments typically relies on laboratory assistants who manually collect, review, and manage student submissions. This process often involves validating whether a submission adheres to format or safety constraints, simulating it to detect potential errors, and finally deploying it to the actual physical device. As a result, this process is time-consuming, labor-intensive, and highly dependent on the availability of technical personnel.

This dependency presents several challenges. First, it limits the scalability of the class by allowing only a limited number of tasks to be processed within a given period, which delays student feedback and reduces opportunities for iteration and learning. Second, human error or oversight during validation can result in the execution of faulty or unsafe instructions, potentially damaging expensive lab equipment. Third, it lacks traceability and transparency for students, who often have limited visibility into the status of their submissions or the reason for a failed execution.

Additionally, as engineering education increasingly incorporates a broader range of processing environments, including both physical systems and software-based platforms, the diversity of task workflows continues to grow. Some assignments may require multiple validation stages, such as format parsing and simulation through a digital twin, before physical deployment. Others may be executed directly in a virtualized environment without simulation, depending on the risk and nature of the task. This variability increases the complexity of managing submissions and requires more flexible and adaptive workflows.

The variability in task types, the need for multistage processing, and the integration with heterogeneous environments all contribute to a complex operational problem. There is a clear need to rethink how task submissions are received, processed, validated, and executed in a way that is flexible, transparent, and reliable while reducing dependency on manual interventions and minimizing the risk of failure or damage during execution.

1.2 Thesis Objectives

The broader vision of this project is to develop a comprehensive system capable of receiving, validating, simulating, and executing student-submitted tasks across various physical and virtual environments, with minimal human intervention. This system would ideally support integration with multiple types of laboratory equipment and software platforms, automate task lifecycle management, and provide real-time feedback to students and instructors.

However, the development of such a complete system is beyond the scope of a single academic project. Therefore, this thesis focuses on the design and evaluation of the architectural approach that underpins such a system. The aim is to explore how task workflows can be modeled, coordinated, and adapted to heterogeneous environments through a flexible and extensible architecture. Rather than implementing all integration points or user interfaces, this thesis focuses on the core logic that governs task progression, error handling, and modularity.

1.2.1 General Objective

Explore and evaluate a system architecture capable of managing multistage task workflows across heterogeneous execution environments, with a focus on flexibility, reliability, and applicability in educational contexts.

1.2.2 Specific Objectives

1. Analyze the functional and architectural requirements involved in automating the processing of student-submitted tasks that may involve multiple processing stages.

2. Propose a system design that enables modular processing flows, supports task chaining, and can adapt to different processing contexts (e.g., physical and virtual environments).

3. Implement a prototype that simulates task processing across multiple conceptual stages, reflecting common execution flows found in educational use cases.

4. Explore mechanisms for handling diverse task workflows, where different types of submissions follow different processing chains based on their processing needs.

5. Incorporate fault tolerance and error recovery strategies, including handling message delivery issues, node-level failures, or connectivity problems with external systems.

6. Evaluate the behavior of the prototype under simulated conditions, measuring task progression, system responsiveness, and support for extensibility and failure recovery.

7. Identify the system limitations and potential for future integration with real infrastructure and user interfaces, as well as its suitability for broader application in academic environments.

1.3 Document Structure

This document is organized into seven chapters, each addressing a specific aspect of the research and development process.

- **Chapter 1 – Introduction:** Introduces the context and motivation behind the work, outlines the problem statement, defines the objectives of the thesis, and explains the scope and structure of the document.

- **Chapter 2 – Context:** Provides an overview of the academic environment in which this work takes place. It describes the current task execution workflows used in laboratory settings, the infrastructure involved, and the limitations of manual processes, leading to the motivation for an improved system.

- **Chapter 3 – Related Work:** Reviews relevant literature and existing systems to establish the context of this work. It also identifies key ideas and gaps that inform the direction of the proposed approach.

- **Chapter 4 – System Requirements and Design Considerations:** Defines the functional and quality requirements of the system, outlines architectural constraints, and discusses the primary design challenges considered during development.

- **Chapter 5 – System Design and Architecture:** Details the proposed architecture, including its components, task flow design, error recovery mechanisms, and extensibility model.

- **Chapter 6 – Implementation:** Presents the prototype in a representative deployment, detailing the hardware setup, selected workloads related to the academic context, and key implementation constraints.

- **Chapter 7 – Evaluation and Results:** Presents the methodology used to evaluate the prototype, the setup for testing, the metrics observed, and a discussion of the system's performance, resilience, and potential for integration with real environments.

- **Chapter 8 – Conclusions and Future Work:** Summarizes the contributions of the thesis, discusses key lessons learned, and outlines limitations and opportunities for future development, including integration with physical and software infrastructure and user-facing components.

Chapter 2

Context

This chapter provides the contextual foundation for the thesis by describing the educational environment in which the proposed system is conceived. It also outlines how task execution is currently handled in courses that make some use of engineering laboratories, the types of infrastructure involved, and the limitations of the existing workflow. This context is essential for understanding the challenges that the system aims to address and the rationale behind the architectural decisions presented in later chapters.

2.1 Overview of the Academic Environment

This section describes the academic environment in which the proposed system is developed. It also highlights the pedagogical value of project-based learning and the importance of integrating hands-on execution environments into engineering education.

2.1.1 Educational Setting

The system proposed in this thesis is being developed within the broader academic context of the Faculty of Engineering at the Universidad de los Andes. Although the project originated from task execution needs identified in a course that utilizes a robotic arm in the Industrial Engineering department, the scope of the project extends to multiple programs, including Systems and Computing Engineering, Electrical Engineering, and potentially others. The goal is to support various practical assignments, whether they require physical infrastructure, virtual environments, or both.

Several engineering courses already incorporate practical components into their curricula, allowing students to apply theoretical concepts through assignments that involve simulation, model execution, or interaction with physical systems. However, the current execution processes across these courses are highly dependent on the availability of teaching assistants and laboratory personnel. This dependency often leads to delays, inconsistencies in execution outcomes, and difficulties in maintaining a uniform workflow between students and staff. In addition, automation efforts for these workflows have stalled due to budgetary and resource constraints.

This thesis addresses these limitations by focusing on the architectural design of a system intended to coordinate task processing in a modular, scalable, and automated way. By enabling workflows that can adapt to different environments and reduce the dependence on manual intervention, the system aims to serve as a foundation for future development and integration of course workflows.

2.1.2 Importance of Project-Based Learning and Practical Assignments

Project-Based Learning (PBL) is a student-centered instructional approach in which learners gain knowledge by working for an extended period to investigate and respond to complex, real-world problems. In engineering education, PBL has been widely adopted as an experiential pedagogy that moves away from rote memorization and towards active problem solving and design experiences (Lavado-Anguera et al., 2024).

At the Universidad de los Andes, the Faculty of Engineering has made a sustained effort to promote project-based learning across its academic programs. Many undergraduate and graduate courses incorporate project-driven practical assignments as a core component of their assessment strategies. Although implementation levels may vary by department, PBL is increasingly seen as a preferred instructional model to align classroom learning with real-world engineering contexts. The university encourages this approach through curricular design, course guidelines, and institutional support mechanisms aimed at fostering hands-on experiences.

In a PBL environment, students take the lead in their learning while instructors act as facilitators or coaches, a shift that has been highlighted as effective in developing the skills that engineers need in the 21st century (Lavado-Anguera et al., 2024). PBL is inherently multidisciplinary and anchored on real-world context; notably, pioneering programs (e.g., the Aalborg model in Denmark) have shown that project-based education can yield better learning outcomes than traditional lecture-based methods due to the high level of student involvement and ownership in the learning process (Lara-Bercial et al., 2024).

Pedagogical Benefits of PBL

A growing body of research indicates clear pedagogical advantages of incorporating PBL in engineering curricula. Students in PBL settings tend to show improved academic achievement and understanding of course material. A recent systematic literature review found that PBL improves academic performance and knowledge retention compared to traditional methods, while also keeping students more attentive and engaged in class (Lavado-Anguera et al., 2024). Importantly, learning through projects helps cultivate higher-order thinking skills; for example, one study reported that the introduction of industry-oriented design projects led to a significant gain in problem solving and critical thinking abilities for engineering students (with measured improvements of 25% in problem solving and 30% in critical thinking) (Mahtani et al., 2024). Such hands-on projects also tend to increase student motivation and satisfaction. In the same study, students reported greater enjoyment and interest when tackling authentic engineering problems, reflecting a deeper engagement in the learning process (Mahtani et al., 2024). Similarly, a 12-year longitudinal analysis at a European university compared graduates from a PBL-based engineering program to those from a traditional program. The findings showed that the PBL-trained students perceived themselves to have a stronger grasp of technical concepts and better problem solving skills, and remained more motivated during their studies (Lara-Bercial et al., 2024). These pedagogical benefits suggest that PBL not only helps students learn engineering content more effectively, but also encourages active learning behaviors (inquiry, collaboration, reflection) that enhance their overall educational experience.

Real-World Relevance and Skill Development

Beyond improving academic metrics, PBL is highly valued for its role in developing practical skills and real-world readiness among engineering students. Engineering projects in the curriculum simulate professional practice that require teamwork, communication, and project management, all skills essential in industry. The literature shows that PBL allows students to develop both technical competencies and crucial non-technical skills (such as teamwork, leadership, and communication) that are key to becoming successful professionals (Lavado-Anguera et al., 2024). For example, PBL alumni in the 12-year study above not only excelled academically, but also reported greater adaptability to the workplace and confidence in applying their skills in new contexts (Lara-Bercial et al., 2024). This indicates that PBL experiences help smooth the transition from school to work by mirroring real engineering challenges and workflows. Institutional data from Worcester Polytechnic Institute (WPI) further reinforce the real-world impact of PBL: in a survey of more than 2,200 alumni, 94% of the respondents said that WPI's project-based curriculum enhanced their ability to develop ideas and 93% reported that it improved their ability to work effectively in teams (Worcester Polytechnic Institute, 2024). The vast majority also credited PBL projects with improving their communication skills and even strengthening personal attributes such as resilience and confidence (Worcester Polytechnic Institute, 2024). These outcomes highlight that through well-designed projects, students practice solving open-ended problems, working in diverse teams, and communicating results, just as professional engineers.

The emphasis on project-based practical learning at the Universidad de los Andes aligns with global educational trends and supports the development of engineering graduates who are both technically competent and professionally prepared. This institutional focus provides the foundation for many of the technical and operational challenges addressed in this thesis, especially those related to scalable and reliable processing of student-generated tasks that go towards the completion of academic projects.

2.2 Current Task Execution Workflow

This section describes how student submissions are currently handled in courses that involve practical assignments requiring execution in either virtual or physical environments. Although there is institutional interest in expanding the use of such workflows, the current execution model remains largely manual, relying heavily on faculty and teaching assistant availability. The following subsections describe the submission process and the steps involved in processing tasks within university laboratories.

2.2.1 Handling of Student Submissions

Currently, students submit their practical assignments through Bloque Neon, the university's learning management platform. This platform is used not only for submitting coursework, but also for sharing grades, receiving feedback, and managing general course logistics. Once submissions are made, course staff manually download files from the platform and execute them locally or in designated execution environments, depending on assignment requirements.

Submissions can vary widely in format. Students may submit compressed files containing complete projects, source code files, machine instructions, or configuration models, depending on the nature of the assignment. There is no standardized file format or submission metadata and naming conventions are not enforced centrally. Although some professors define their own naming rules to maintain order and facilitate grading, these practices vary by course and are not universally adopted across the faculty.

Although there is an intention to improve and standardize the submission process, including the development of a dedicated user interface or potential integration with Bloque Neon, these efforts are still in the early planning stages. Currently, all interactions with submissions are manual and the system lacks automation or structured task metadata, making consistent handling more difficult across courses and staff members.

2.2.2 Steps Performed in Laboratories

Once student submissions are downloaded from Bloque Neon, they are reviewed and processed by course staff, primarily teaching assistants (TAs) or laboratory personnel, depending on the nature of the assignment. Tasks that require physical execution, such as those that involve specialized hardware such as the robotic arm, are typically handled by laboratory personnel with access to the necessary equipment. In contrast, tasks designed for virtual execution environments may be run directly by TAs or instructors on their own machines.

Currently, no formal validation step precedes the execution of submissions. Tasks are executed as-is, and the students receive feedback based on the performance of their submission. Grading is performed manually according to rubrics defined by the course staff. These rubrics may assess correctness, functionality, adherence to specifications, or the ability of the system to perform a specific task (e.g., completing a movement on a robotic arm or passing a set of automated tests).

Interaction with physical equipment is done using the appropriate interfaces provided by the hardware. For virtual environments, tasks may be run as scripts or test suites, depending on the course. Feedback to students varies according to the type of assignment: in some cases, students receive detailed comments, while in others, only rubric scores are returned. This variability in grading feedback reflects the decentralized and manual nature of current processes.

Execution delays are common and derive primarily from the time required to download, review, and process a large number of submissions manually. In cases where student submissions are incomplete or fail to run, staff must often contact students for corrections, introducing additional delays and inconsistencies in turnaround times. This highlights the need for greater automation, validation support, and standardization across task execution workflows.

2.3 Infrastructure and Execution Environments

The execution of student tasks in engineering courses requires the use of specialized infrastructure, either in the form of physical devices housed in university laboratories or virtual environments tailored to specific technologies. Although some infrastructure is currently in use, most workflows remain manually operated and lack automation, standardization, and integration. This section provides an overview of the types of infrastructure currently available or envisioned, highlighting their relevance to the educational process and the challenges associated with their use.

2.3.1 Physical Infrastructure in Laboratories

Within the Faculty of Engineering at the Universidad de los Andes, several pieces of physical infrastructure have the potential to support task execution. These include a robotic arm, a test bench for an electrical grid, and 3D printers. Of these, the robotic arm is the only system that has seen some level of use within a course setting, although even this integration has been limited and largely manual.

Physical infrastructure is distributed across different laboratories, each associated with the department that owns and maintains the corresponding equipment. Access to these systems is typically restricted to laboratory personnel and direct student interaction with the hardware is not currently allowed. The limited number of devices available, along with logistics and staffing constraints, imposes a hard ceiling on the number of tasks that can be executed and evaluated using physical infrastructure.

Due to the heterogeneity of the systems involved and the lack of standardized procedures or interfaces, interactions with physical devices vary by department and are handled internally by trained staff. The specific methods used to load, execute, and observe tasks on each system are not fully documented or standardized, and setup or teardown times are currently unknown. These limitations highlight the need for more structured and scalable execution workflows, particularly if the use of physical systems is to be expanded across courses.

2.3.2 Conceptual Considerations of Virtual Environments

In parallel with physical systems, several courses within the engineering curriculum incorporate virtual environments to support the execution and evaluation of student tasks. These environments vary significantly depending on the nature of each course. For example, some classes require students to submit test cases for web applications, interface testing across mobile devices, or other system-level behaviors that must be executed in isolated environments. In some envisioned use cases, tasks such as file parsing or structural validation would serve as intermediate steps prior to physical execution, particularly when malformed or unsafe instructions could lead to equipment misuse or damage. For instance, a poorly formatted file intended for a robotic arm or electrical system may not only fail during execution, but could also result in hardware-level faults if not tested in a simulated or digital twin environment first.

Although some of these environments are already used in practice, the workflows are predominantly manual. For example, in the case of automated testing with Cypress, students submit their test cases and course staff are responsible for manually running them in their local environments. This approach introduces multiple points of inconsistency, particularly due to variations in how student and staff machines are configured.

One of the key challenges in working with virtual environments is the lack of standardization across platforms. Differences in operating systems, installed dependencies, library versions, and project configurations frequently lead to compatibility issues that affect the ability to execute and evaluate student submissions uniformly. This lack of environmental parity results in additional work for the staff, who often must debug or adapt each submission before it can be tested correctly.

Although there is no formal system for managing these virtual workflows at the faculty level, there is a clear opportunity to improve consistency through standardization and automation. The architectural design presented in this thesis aims to support extensibility by allowing future integrations with a wide range of virtual environments. Rather than limiting the system to predefined tools or technologies, the goal is to enable flexible coupling with whatever technologies individual courses require, supporting both present and future educational needs.

2.4 Limitations of the Existing Workflow

Although the Faculty of Engineering at the Universidad de los Andes possesses both physical and virtual infrastructure that could support a variety of tasks, current workflows for processing student submissions are limited by a lack of standardization, automation, and scalability. The process remains fragmented across departments and highly dependent on human intervention, which introduces inefficiencies and potential risks. This section outlines the primary limitations that affect current workflows, which the proposed architectural system aims to address.

2.4.1 Dependence on Human Availability

One of the most pressing limitations of the current approach is its complete dependence on faculty and support staff to process each task manually. After students submit their assignments through Bloque Neon, it falls on instructors, teaching assistants (TAs), or laboratory personnel to download each submission, manually verify its readiness, and execute it on the intended environment, be it a local machine, a simulator, or a physical device.

This model is inherently fragile. When staffing levels fluctuate due to academic workload, budgetary restrictions, or the unavailability of trained personnel, processing throughput suffers. In fact, some educational programs that intended to introduce physical infrastructure or automated task workflows have been paused or delayed because there was no sustainable operational model to support them. The process cannot scale without consuming significantly more staff time, which becomes unfeasible as course sizes grow or as more courses attempt to incorporate hands-on elements into their curricula.

2.4.2 Delay in Feedback and Limited Throughput

In traditional workflows, feedback loops are slow and capacity is severely limited. Because staff must handle every submission individually, students may have to wait several days, or in some cases longer, to receive evaluation results, especially when assignments require interaction with physical infrastructure that only a limited number of people can access. In addition, the number of submissions that can be processed within a given timeframe is directly constrained by the availability and responsiveness of staff; this creates a hard ceiling on throughput that limits the scalability of this approach.

These delays affect the operational flow of the course and increase the administrative burden on staff. In cases where submissions contain structural issues, misconfigured files, or errors that prevent successful execution, staff members must contact students to request corrections or clarifications. This manual communication further extends the feedback cycle and can create uneven processing times between students. The lack of a streamlined system for pre-checking or automating common failure points contributes to these inefficiencies and makes it difficult to support larger cohorts or expand task-based practical assignments into more courses without overextending staff resources.

2.4.3 Risk of Equipment Misuse or Damage

Another critical limitation is the lack of intermediate validation or simulation steps before the execution of a task on physical infrastructure. As it stands, staff often run submitted files directly on hardware such as robotic arms or electrical systems without prior automated checks. Although some instructors may conduct informal manual reviews, there is no structured validation pipeline to assess the format, safety, or logical correctness of a task before execution.

This approach introduces significant risk. Poorly structured or incomplete submissions could cause runtime errors, improper device behavior, or even physical damage to the infrastructure. For example, if a robotic arm receives a command sequence with undefined positions or timing mismatches, it could collide with itself, exceed physical boundaries, or enter an unexpected state. As more devices are incorporated into coursework, the potential for damage due to unvalidated input increases, highlighting the urgent need for a safe and automated pre-execution review layer.

2.4.4 Lack of Transparency and Standardization

Currently, there is no universal convention that governs how students should name, structure, or document their submissions. Some instructors enforce local conventions, such as requiring students to name files in a specific way or follow a folder structure, but these rules vary widely and are often informal. As a result, faculty often spend time reformatting or interpreting submissions before they can be executed.

Furthermore, the system lacks traceability, that is, the ability to track the progress of a submission throughout its life cycle. Once a student submits a file, there is no shared record of whether it has been received, validated, simulated, executed, or processed in any way. Feedback is often provided manually and outside the system (e.g., through Bloque Neon comments or email), with no linkage to execution outcomes or error logs. This fragmentation not only hinders the student's learning experience but also makes it difficult for staff to coordinate workflows, debug failures, or provide consistent grading.

This lack of transparency also means that systemic issues, such as recurring submission errors or hardware failures, go unnoticed across iterations. Without centralized data on submission failures or performance anomalies, it is challenging to improve processes over time or to provide insights that could help students prepare better assignments.

2.5 Vision for an Improved System

Given the limitations of the current workflow, including manual processing, dependency on staff availability, inconsistent file handling, and lack of validation before execution, there is a clear opportunity to envision a more robust, automated, and scalable system. Such a system would aim not only to alleviate the operational bottlenecks observed across courses, but also to lay a technical foundation for the broader integration of physical and virtual infrastructure into the educational process.

The vision outlined in this section reflects a modular, extensible, and infrastructure-agnostic approach. Although full implementation is beyond the scope of a single academic project, this thesis focuses on establishing the architectural foundation necessary for such a system to evolve. The following subsections present key characteristics of the envisioned system, which guide the design choices explored throughout this work.

2.5.1 Automation of Task Workflows

A central objective of the envisioned system is to reduce the dependency on manual intervention by automating the submission-to-execution pipeline. Rather than relying on teaching assistants or laboratory personnel to download, verify, and run each task individually, the system would coordinate task progression across multiple stages in a semi-autonomous or fully autonomous manner. This includes automatically receiving student submissions, routing them through defined validation and execution steps, and collecting results or logs for review.

Automation would not eliminate human oversight altogether, but would instead shift the role of staff from manual operators to reviewers and decision-makers. Tasks that fail at a certain stage, such as validation or simulation, could be flagged for manual review, while others could complete processing without intervention. This automation layer would enable higher throughput, reduce processing delays, and make task-based practical assignments more sustainable for larger class sizes or multi-course deployment.

2.5.2 Integration of Simulation and Validation Steps

To address the risks associated with the execution of tasks on physical infrastructure, the system should support intermediate processing stages that serve as validation checkpoints. These steps would enable automatic parsing, format checking, and logical verification of a submission before it reaches the execution phase. For example, a model destined for a robotic arm could be parsed for compliance with safety constraints, simulated in a digital twin environment, and only then, queued for physical execution.

These validation and simulation steps not only safeguard equipment but also increase confidence in the execution process by identifying structural or semantic errors early. The inclusion of such stages would help standardize task workflows across courses while maintaining flexibility to accommodate varying requirements. Courses that rely exclusively on virtual environments could bypass hardware simulation, whereas hardware-integrated courses could configure multiple safety layers before reaching the device.

2.5.3 Extensibility towards Multiple Environments and Task Types

A defining feature of the proposed system is its ability to support a wide range of tasks and execution environments without being tightly coupled to any particular technology or platform. This is especially important given the heterogeneity of educational tools and course designs across departments. Some tasks may involve Python scripts, automated tests, or configuration files for virtual systems, while others may require instructions for robotic arms, electrical grids, or 3D printers.

Rather than prescribing fixed technologies, the system should be designed to integrate new processing environments through a modular architecture.

This could be achieved via plugin mechanisms or dynamically loaded components that define how specific tasks are interpreted and processed. Such extensibility would make the system sustainable over time and adaptable to the evolving needs of the faculty. It would also allow instructors to continue to use domain-specific tools while benefiting from a unified task execution framework.

Chapter 3

Related Work

This chapter presents a review of systems, approaches, and concepts that relate to the larger problem addressed in this thesis. The purpose is to provide context, identify prior efforts in adjacent areas, and explore ideas that inform or contrast with the direction taken in this work. Rather than offering an exhaustive survey, the focus is on highlighting relevant characteristics, limitations, and insights that contribute to understanding the space in which this project is located.

3.1 Project-Based Learning

As described in 2.1.2, various engineering courses at the Universidad de los Andes are structured around project-based practical assignments. These assignments often require students to submit executable artifacts, such as code, control models, or instruction files intended for deployment in virtual or physical environments. Although this thesis does not aim to assess the effectiveness of project-based learning (PBL) as a pedagogical model, it acknowledges that such instructional formats are a central component of the academic context in which the challenges addressed in this work emerge.

Portions of this chapter, particularly the comparative analysis of general-purpose orchestration frameworks, were developed with the assistance of OpenAI's ChatGPT. The tool was used to explore technical documentation and synthesize draft content. The author performed the formulation, interpretation, and integration into the thesis.

Reliance on execution-based assignments introduces the need for reliable and efficient infrastructure to manage and evaluate student work. Unlike traditional assignments that can be reviewed statically, these tasks require dynamic evaluation in controlled environments. The characteristics of project-based and hands-on learning formats, highlighted in 2.1.2, therefore serve as the foundation for understanding the operational and technical difficulties explored in the remainder of this thesis.

3.2 Educational-Oriented Orchestration Platforms

Several platforms in the field of educational technology have explored how to orchestrate tools, resources, and activities across learning environments. Among these, GLUE!-PS and Toccata stand out for their contributions to the coordination of distributed educational processes. Although they were developed for different contexts (GLUE!-PS for the deployment of learning designs across virtual learning environments (VLEs) and Toccata for real-time orchestration in classroom settings), both systems reflect a growing recognition of the need to manage complexity and flow in educational scenarios.

GLUE!-PS focuses on bridging the gap between learning design authoring tools and virtual learning platforms. It achieves this by introducing a multi-layered integration model in which learning designs created in different specification languages can be deployed uniformly across multiple environments without modifying the tools themselves (Prieto et al., 2011). This flexibility allows teachers to reuse and scale learning scenarios across platforms (Prieto et al., 2011). Similarly, Toccata addresses the fragmentation of classroom digital activities by offering teachers a centralized interface to manage group tasks, assign digital resources, and monitor real-time progress (Lachand et al., 2018). It supports coordination and awareness in live teaching sessions and helps structure the delivery of activities (Lachand et al., 2018).

Despite their value, these systems are focused on pedagogical orchestration, not technical execution workflows. GLUE!-PS enables teachers to deploy and instantiate abstract learning designs, but does not handle the processing of student-submitted artifacts. It does not integrate with simulation layers, parsing tools, or hardware infrastructure (Alario-Hoyos et al., 2012). Likewise, Toccata enables the management of classroom interactions and provides teachers with visibility over student engagement, but does not extend to orchestrating back-end systems that process technical tasks (Lachand et al., 2018). Neither system is concerned with how files are validated, routed, or executed once submitted; nor do they support integration with heterogeneous environments, such as virtual test benches or physical robotics labs.

This distinction is central to the positioning of the system proposed in this thesis. Although GLUE!-PS and Toccata reflect mature efforts to coordinate educational activity flow, they do not address the orchestration of task-processing pipelines. The problem this thesis targets is downstream from pedagogical design; it begins once a student submits a task and focuses on ensuring that the task is processed through the appropriate stages. As such, the work presented here complements rather than competes with prior orchestration platforms by introducing architectural mechanisms that manage processing logic across diverse environments, with the possibility of integrating physical infrastructure and supporting API-based status tracking.

3.3 Workflow Orchestration and Task Queuing Frameworks

Modern software systems frequently rely on workflow orchestration and task queuing frameworks to manage the execution of automated processes across distributed components. These frameworks enable the coordination of tasks that may be executed asynchronously, retried in case of failure, or chained together into larger processing pipelines. Originally designed for use cases such as data processing, background job scheduling, and microservice communication, these tools have become central to the development of scalable and fault-tolerant systems.

Given that this thesis investigates the coordination of multi-stage task flows, existing orchestration frameworks offer valuable insight into how complex processes are managed in practice. This section explores a selection of widely adopted task orchestration and queuing tools, highlighting their capabilities, architectural assumptions, and relevance to the broader problem space addressed in this work.

3.3.1 Task Queuing Frameworks: Celery

Celery is a distributed task queue framework widely used in Python-based systems to execute background jobs asynchronously. It supports task scheduling, retries, and chaining through simple APIs, and is commonly used in web and backend applications to decouple workload execution from request handling (Celery Project, 2025).

Although Celery is robust for stateless, predefined operations, such as data processing or asynchronous notifications, it assumes that workflows are known in advance and defined programmatically. It is not designed to handle tasks whose processing paths vary based on content or external validation, nor does it natively support file-based submission workflows or integration with heterogeneous environments.

As such, Celery provides useful architectural ideas in task decoupling and worker coordination, but it lacks the domain awareness and dynamic routing capabilities needed for systems that process student-submitted artifacts through multi-stage workflows.

3.3.2 DAG-Based Workflow Orchestration: Apache Airflow and Argo Workflows

Apache Airflow and Argo Workflows are orchestration frameworks based on Directed Acyclic Graphs (DAGs), where each node represents a task and the edges represent the dependencies between them. Airflow is commonly used to automate data pipelines and ETL processes (Apache Software Foundation, 2025), while Argo is designed to manage containerized workflows in Kubernetes environments (Argo Project, 2025).

These systems are well-suited for scenarios where workflows are predefined, repetitive, and composed of discrete steps that follow a known execution order. They support scheduling, retries, and monitoring, and offer visualization tools that make pipeline management accessible and auditable.

However, both frameworks rely on workflows being fully specified before execution. They are less flexible in handling cases where the task flow must adapt based on the content or outcome of previous steps. Furthermore, their design focuses on cloud-native and service-based environments, and they do not provide built-in support for heterogeneous infrastructure or dynamic integration with domain-specific processing environments.

Although these systems offer valuable insights into dependency management and step coordination, their assumptions around static workflows and container-based infrastructure may limit their applicability in more dynamic or domain-specific contexts.

3.3.3 Code-Defined and Dynamic Orchestration: Temporal and Conductor OSS

Temporal and Conductor OSS are orchestration frameworks that define workflows through code or state machines. Temporal supports durable workflows written in languages like Go or Java (Temporal Technologies, Inc., 2025), while Conductor OSS uses JSON-based definitions to sequence tasks across distributed services (Conductor OSS, 2025). Both are designed for microservice-based, cloud-native environments, offering advanced control over retries, branching, and long-running execution.

While powerful, these systems assume that workflows are composed of stateless service calls, and they operate within API-driven ecosystems. They are not tailored for coordinating file-based submissions, heterogeneous processing environments, or workflows that must adapt dynamically based on content validation or execution outcomes. Additionally, they do not address educational needs such as traceability, feedback loops, or infrastructure diversity.

Their models offer valuable insights into workflow resilience and dynamic coordination, but their assumptions make them less applicable in domains where execution involves diverse formats, environments, or domain-specific constraints.

3.4 Opportunistic Infrastructure for Task Execution

UnaCloud is an academic Infrastructure-as-a-Service (IaaS) platform developed at the Universidad de los Andes to support opportunistic cloud computing. Its goal is to make use of underutilized computing resources, such as idle laboratory computers, to offer virtual machines or containers to execute resource-intensive workloads (Arévalo, 2013; Montañez, 2019; Sekler, 2016; Urazmetov, 2011).

The platform has evolved to support high-performance computing tasks, container deployment, and domain-specific applications in areas such as sequence alignment and simulation (Arévalo, 2013; Montañez, 2019; Sekler, 2016; Urazmetov, 2011). Its architecture enables users to deploy virtualized environments with minimal administrative overhead, making it a flexible and cost-efficient infrastructure provider for academic projects (Arévalo, 2013; Montañez, 2019; Sekler, 2016; Urazmetov, 2011).

While UnaCloud excels in infrastructure provisioning, it does not provide mechanisms for task-level coordination, submission-aware workflows, or dynamic multi-stage processing. There is no orchestration of different processing steps; instead, users are expected to configure and execute their workflows within the virtual environments they deploy.

In this thesis, UnaCloud is not considered a competing solution, but rather a potential execution backend. Certain processing nodes, such as simulation or test environments, could be deployed on virtual machines provisioned by UnaCloud. This possibility falls outside the current scope, but represents a viable direction for future work, particularly in scenarios where lightweight orchestration must scale across distributed academic resources.

3.5 Prior Implementation

This thesis builds upon work previously conducted by the author during his undergraduate studies. That earlier project aimed to create a system capable of orchestrating distributed task execution for laboratory environments at the Universidad de los Andes, using a message-driven architecture that incorporated RabbitMQ and FTP-based file storage (González, 2024). The goal was to allow students to submit tasks remotely, which would be routed through a global processing component and executed across distributed clusters (González, 2024).

Although the system demonstrated basic task distribution and results reporting, it suffered from critical architectural and operational limitations. The overall structure lacked formal modularity and component isolation. The messaging and processing logic were tightly coupled and manually configured, with limited support for extensibility, fault isolation, or dynamic task types. The deployment was manual, with poor encapsulation of responsibilities, no abstraction over execution environments, and offered minimal scalability.

This thesis presents a fundamental re-engineering of the system. It introduces a layered and component-based architecture that emphasizes message-driven coordination, actor-based concurrency, and fault isolation. Each processing node now follows a clearly defined structure, internal responsibilities are modularized using the actor model, and task routing is abstracted through formalized message schemas and broker-based coordination. The architecture is designed to support fault-tolerant execution, dynamic task orchestration, and clean subsystem isolation, addressing the weaknesses of the earlier design and allowing for a broader scope of application and evaluation.

The systems reviewed in this chapter offer valuable capabilities in distributed task execution, workflow orchestration, and educational process coordination. However, each was developed to address a specific class of problems that do not fully align with the requirements defined in this thesis.

General-purpose orchestration frameworks, such as Celery, Airflow, Argo, Temporal, and Conductor OSS, are effective in managing jobs across cloud-native infrastructures, but they operate under assumptions that limit their applicability in heterogeneous or stateful processing contexts. These systems typically assume predefined workflows, stateless task invocation, and homogeneous or containerized execution environments. As such, they are ill-suited for scenarios where tasks must be dynamically routed based on their content or intermediate outcomes, particularly when the processing involves specialized infrastructure or physical components.

Conversely, educational platforms such as GLUE!-PS and Toccata focus on coordinating learning activities and pedagogical workflows, but lack runtime orchestration, artifact-level evaluation, or support for integration with distributed execution environments. Their abstraction level does not address the technical challenges posed by dynamic, multi-stage task execution.

This reveals a clear opportunity: currently, no system is designed to coordinate multistage workflows initiated by student-submitted tasks, where the execution path may evolve dynamically and span both virtual and physical infrastructure. To explore this space, the author previously developed a prototype system during their undergraduate studies. Although that system offered an initial attempt at remote orchestration and distributed execution, it exhibited architectural limitations that restricted its extensibility and robustness. The current thesis builds upon the insights gained from that early effort, proposing a formally restructured architecture that addresses the identified shortcomings and meets the specific demands of dynamic educational task orchestration.

Chapter 4

System Requirements and Design Considerations

This chapter defines the requirements and constraints that guide the design of the system proposed in this thesis. The functional requirements that the system must fulfill and the quality attributes that are important to ensure its effectiveness, robustness, and adaptability are discussed here. In addition, this chapter outlines key assumptions, relevant use cases, and the architectural challenges that arise from supporting diverse tasks, multistage workflows, and heterogeneous execution environments. These considerations provide the foundation for the architectural approach discussed in the following chapters.

4.1 Functional Requirements

This section outlines the core functional requirements of the system, derived from the analysis of existing workflows, the challenges discussed in previous chapters, and the intended use of the system within educational and experimental settings. Each requirement is followed by a brief rationale that connects it to the underlying motivations.

1. **Task Ingestion and Identification**
   Requirement: The system must be able to receive new tasks and assign a unique identifier to each.

Rationale: To support traceability, progress tracking, and modular coordination, tasks must be individually addressable throughout their lifecycle.

2. **Processing Flow Management**
Requirement: The system must coordinate the movement of tasks through multiple processing stages and dynamically determine their routing based on task type, content, system conditions, or stage outcomes.
Rationale: Task workflows may vary depending on their characteristics. The system must support both flexible progression between stages and the selection of appropriate processing units at runtime.

3. **Processing Stage Output Management**
Requirement: The system must store the results produced at each processing stage in a structured and accessible manner.
Rationale: Task outcomes may need to be consumed by downstream stages, reviewed by operators, or analyzed for feedback or grading. Persistent storage ensures that intermediate and final results are available when needed.

4. **Task State Introspection**
Requirement: The system must ensure that the current state of each task is maintained consistently and can be known at any point during its lifecycle, including its progression across processing stages and substage transitions.
Rationale: Accurate and up-to-date task state information is necessary to enable monitoring, support decision-making, and provide the basis for routing logic, recovery strategies, or external system interaction.

These functional requirements define the operational scope of the system's distribution layer and guide the development of architectural components that ensure consistent, flexible, and traceable task processing. Although additional functionality may be desirable in extended implementations, such as integration with user interfaces or administrative tooling, this set represents the core capabilities required to address the challenges and design goals outlined within the scope of this project.

4.2 Quality Attributes

Although several quality attributes were identified as relevant to the system's behavior, a focused analysis was conducted to determine which ones are most central to the objectives of this work. Among the attributes considered, performance and scalability naturally arose as important concerns. However, given the nature of this system as a coordination and distribution layer, rather than an execution engine, performance was not deemed core to the architecture proposed in this thesis. Task execution time ultimately depends on the behavior and responsiveness of the processing work performed by distributed nodes, which is beyond the scope and control of this thesis.

However, scalability was included as a secondary attribute due to potential variability in deployment scenarios. The system must be able to coordinate a small number of processing nodes or scale to manage significantly more, depending on how the environment evolves. Although the actual performance of the full system under load depends on individual node behavior, it is still possible to test the scalability of the distribution logic itself by simulating a high number of nodes with fixed response characteristics.

As a result of this analysis, fault tolerance and integrability were identified as the most critical quality attributes for the system. The ability to recover from failures in individual task stages, node unavailability, or even complete system failure, and the flexibility to incorporate diverse task types and specialized execution environments, are fundamental to ensure system robustness, applicability, and long-term sustainability.

4.2.1 Fault Tolerance

Fault tolerance is a central quality attribute for a system that operates in a distributed and potentially unreliable execution context. Since task processing is delegated to external nodes, some of which may be manually deployed, under-provisioned, or prone to failure, the system must be capable of detecting, isolating, and recovering from disruptions at multiple levels.

At the system level, fault tolerance involves identifying dropped messages, unreachable nodes, and stalled workflows, and responding through strategies such as retries, timeouts, task redelivery, or fallback routing. At the node level, the system must ensure that communication protocols are robust and that node processes respond reliably to coordination signals, even when internal task-specific logic fails.

Although the exact logic used by nodes to process tasks may vary depending on the execution environment, the base node implementation and its interaction with the system fall within the scope of this work. As such, the architecture must support both centralized and distributed mechanisms for error detection and recovery. In this context, fault tolerance underpins system dependability and its suitability for autonomous operation in educational and research environments.

4.2.2 Integrability

Integrability refers to the system's ability to incorporate new processing logic, task types, and execution environments without requiring modification of core coordination mechanisms. Unlike interoperability, which focuses on communication between systems, integrability is concerned with architectural adaptability, supporting internal growth, variation, and modularity.

This quality is essential in settings where workflows evolve over time. In educational institutions, for example, new laboratory equipment, software frameworks, or analysis tools may be introduced frequently, each requiring different types of validation and execution steps. A highly integrable system allows for the introduction of new processing nodes, such as those that execute robotic simulations, automated tests, or digital twins, without disrupting the existing pipeline.

The system must offer well-defined extension points for registering new task types, assigning processing rules, and delegating execution responsibilities. By decoupling task-specific logic from the routing and coordination core, the system becomes easier to maintain, extend, and adapt to future academic and research needs.

4.2.3 Scalability

Although not central to the correctness or core coordination logic of the system, scalability is an important secondary attribute due to the variability in deployment scenarios. In some cases, the system may be required to coordinate only a handful of nodes; in others, particularly during peak usage periods or across multiple classes, it may need to manage many more.

Scalability in this context refers specifically to the ability of the distribution layer to ingest, track, and route a large number of tasks efficiently, without becoming a bottleneck. It is not concerned with execution time, which is determined by the internal logic of the nodes, but with system responsiveness and throughput under increased coordination demand.

Testing scalability can be achieved through simulations that assume a high number of available nodes with fixed response times. This allows for evaluation of the coordination logic under load, even in the absence of actual processing infrastructure.

4.3 Design Constraints

The system must be designed within the bounds of certain non-negotiable conditions that reflect the nature of the execution environment and task model. These constraints arise from the external systems and operational realities that the architecture must accommodate and cannot change.

4.3.1 Heterogeneous and Opaque Execution Environments

The system coordinates the processing of tasks across nodes that it does not deploy, monitor, or manage directly. These nodes may run on heterogeneous infrastructure, be manually launched, and vary in availability or responsiveness. Moreover, the internal logic used by each node to process tasks is not defined by the coordination system. As such, the system must treat processing nodes as loosely coupled black boxes and rely on abstract coordination mechanisms that are agnostic to specific execution behavior or infrastructure details.

4.3.2 File-Based Task Representation

Tasks are represented as file-based artifacts, often including code, configuration, or machine-specific instructions. This constraint imposes the need for explicit file transfer protocols and metadata tracking, and it precludes the use of orchestration approaches that assume structured stateless service interactions. The architecture must accommodate asynchronous file handling, storage management, and routing of file-based tasks between processing stages.

4.4 Assumptions

In designing the system, a set of foundational assumptions were made about the operational context, external components, and expected usage patterns. These assumptions do not constrain the architecture, but provide necessary context for understanding its scope, behavior, and interaction boundaries. They also clarify what the system expects to be true at runtime in order to function correctly.

4.4.1 Task payloads may be syntactically or semantically incorrect

The system does not validate the content or correctness of the task files it transports. Task validation is assumed to be part of the execution logic implemented in the processing nodes, if required. The coordination layer is responsible only for transferring and routing task artifacts between stages.

4.4.2 Execution logic is externally defined and task-specific

Although the communication and base behavior of each node (e.g., file handling, message exchange) is part of the system design, the task-specific execution logic within each node is defined externally. The coordination layer does not interpret or modify task behavior and assumes no knowledge of how tasks are internally processed.

4.4.3 Node deployment is externally managed

Processing nodes will most likely not be deployed or orchestrated by the system and will be subject to university infrastructure. Their availability may vary, and the system must tolerate both the presence and absence of specific nodes at runtime.

4.4.4 All tasks are represented and processed as file-based artifacts

The system assumes a file-centric workflow: task input, intermediate results, and final outputs are managed as files. Although this design does not preclude integration with future models that support other forms of task representation, the architecture is built around explicit file handling and transport between components.

4.4.5 Execution environments are heterogeneous and potentially unreliable

The system must operate in various processing environments, including virtual and physical infrastructure. It is assumed that these environments may differ in performance, availability, and reliability, and the coordination logic must be resilient to such variability.

4.5 Architectural Challenges

The design of the system involves addressing a set of nontrivial architectural challenges that emerge from its intended purpose, operational context, and the quality attributes it prioritizes. These challenges are not imposed externally, but arise from the system's goal of coordinating multistage heterogeneous task workflows in environments characterized by decentralized processing, limited control over execution logic, and variable infrastructure reliability. This section outlines the most significant architectural concerns identified during the design process and highlights their implications for the structure and behavior of the system.

4.5.1 Task Variability and Heterogeneity

Given the architectural intent to coordinate tasks across decentralized and extensible environments, the ability to accommodate variability in task definitions and heterogeneity in execution infrastructure introduces a foundational design challenge. The system must be capable of managing tasks that differ significantly in structure, purpose, and expected processing behavior while interacting with execution environments that vary in platform, configuration, and reliability.

Task definitions are not centrally managed and may be introduced by different actors or evolve over time. Each task may be governed by its own set of processing requirements, number of stages, and outcome expectations. Moreover, the execution logic associated with each task type is implemented externally, within processing nodes that the system does not control. As such, the system must coordinate tasks without assuming knowledge of the internal execution semantics, relying solely on metadata, communication protocols, and declared state.

In parallel, the system must route tasks to execution environments that range from software-based infrastructure, such as emulated or containerized testbeds, to physical laboratory devices with hardware constraints and variable responsiveness. These environments may expose different interfaces, produce outputs in inconsistent formats, or impose execution delays that are not observable from the coordination layer. The architecture must abstract over these inconsistencies and provide a uniform coordination experience regardless of the underlying execution platform.

Addressing this challenge requires the system to maintain a high degree of decoupling between coordination logic and processing logic. Task routing, lifecycle tracking, and failure handling must be implemented without binding the system to particular task structures or infrastructure assumptions. This level of abstraction is essential to support extensibility, future integration scenarios, and long-term maintainability.

4.5.2 Multistage Processing

Given the architectural goal of enabling flexible task workflows composed of distinct processing stages, the ability to manage stage progression dynamically and consistently introduces a critical coordination challenge. Tasks may require multiple sequential stages (e.g., validation, transformation, simulation, execution) each handled by a different processing node. The number, type, and order of stages may vary depending on the task, and the system must support such workflows without enforcing static execution paths.

The system must maintain accurate knowledge of the current state of a task and determine its next destination based on the outcome of the most recent stage. Since the coordination layer is decoupled from the internal execution logic, it cannot infer the appropriate progression through introspection; instead, it must operate based on explicit state updates and standardized stage completion reports. This introduces the need for a robust state-tracking mechanism capable of capturing transitions, identifying failures, and enabling conditional routing based on declared outcomes.

Concurrency adds further complexity. The system must be able to coordinate the simultaneous processing of multiple tasks at different stages without introducing race conditions, bottlenecks, or state inconsistencies. Each task must be independently managed through its own lifecycle, while the system preserves correctness in routing and result handling even as node availability fluctuates and processing durations vary.

Architecturally, this challenge necessitates the design of a modular and extensible orchestration model that supports arbitrary stage compositions. The system must accommodate workflows that evolve over time, new task types with previously unseen stage requirements, and varying routing logic, all without altering the underlying coordination mechanisms. Ensuring that multistage processing is both flexible and reliable is essential to maintaining system generality and long-term adaptability.

4.5.3 Fault Tolerance

Given the architectural separation between task coordination and task execution, ensuring reliable system behavior in the presence of execution failures constitutes a key design challenge. Processing nodes are independently responsible for executing tasks and reporting their outcomes, and the system does not intervene in or monitor the internal execution logic. As such, it must be designed to handle incomplete or failed task executions without centralized visibility into their internal operation.

Failures may occur due to a wide range of causes, including timeouts, crashes within task-specific logic, or node-level interruptions. Because the system does not control execution environments directly, it must rely on each node to detect failures locally,using criteria defined within the node implementation, and to report those failures explicitly through the communication protocol. The coordination layer then must determine how to respond, such as retrying the task, rerouting it to another node, or marking it as failed.

This model requires the system to maintain a reliable and consistent failure handling mechanism that operates independently of any assumptions about the deployment topology. Tasks may be processed concurrently while node availability may vary over time. The system must therefore avoid cascading effects from isolated failures and ensure that the task state remains accurate throughout retry and recovery attempts.

From an architectural standpoint, this requires the integration of a failure-aware task management strategy, which includes retry policies, error classification, and termination criteria, all implemented in a way that is agnostic to the underlying infrastructure. Such capabilities are essential to ensure that the system remains reliable under a wide range of operational conditions and execution behaviors.

4.5.4 Traceability

Given the architectural focus on coordinating multistage task workflows across independently executing processing components, ensuring traceability of task progression constitutes a critical system requirement. Tasks may transition through multiple stages, each executed by different nodes under varying conditions and with independently defined logic. Therefore, the system must maintain a coherent and complete record of the task lifecycle, independent of the internal execution details of any single processing node.

Because the coordination layer does not observe task execution directly, it must rely on standardized signals from the nodes, such as status updates, completion messages, and file outputs, to infer task progression. These signals must be accurately captured and associated with the corresponding task, forming a traceable history from submission through completion or failure.

Traceability is essential for system observability, operational transparency, and accountability. It enables users or administrators to inspect task histories, identify failure points, verify results, and correlate inputs with results. In educational and experimental contexts, it further supports grading, feedback, and reproducibility.

Architecturally, this requires the implementation of a structured state-tracking and logging mechanism that persists task-level metadata and stage outcomes across all phases of execution. This mechanism must be designed to operate independently of the specific execution environments, ensuring that traceability is maintained regardless of infrastructure variability or task logic complexity. In addition, the traceability layer must support concurrent task tracking without conflict or inconsistency, enabling reliable operation under diverse and dynamic workloads.

Collectively, the challenges described in this section underscore the architectural complexity inherent in designing a coordination system that must operate across various types of tasks, execution environments, and runtime conditions. Addressing task variability, multistage progression, fault recovery, and traceability requires a design that is modular and resilient, capable of maintaining correctness, flexibility, and transparency without relying on control over execution logic or infrastructure. These challenges have directly informed the architectural principles and system components presented in the following chapter.

Chapter 5

System Design and Architecture

5.1 Overview of the Proposed System

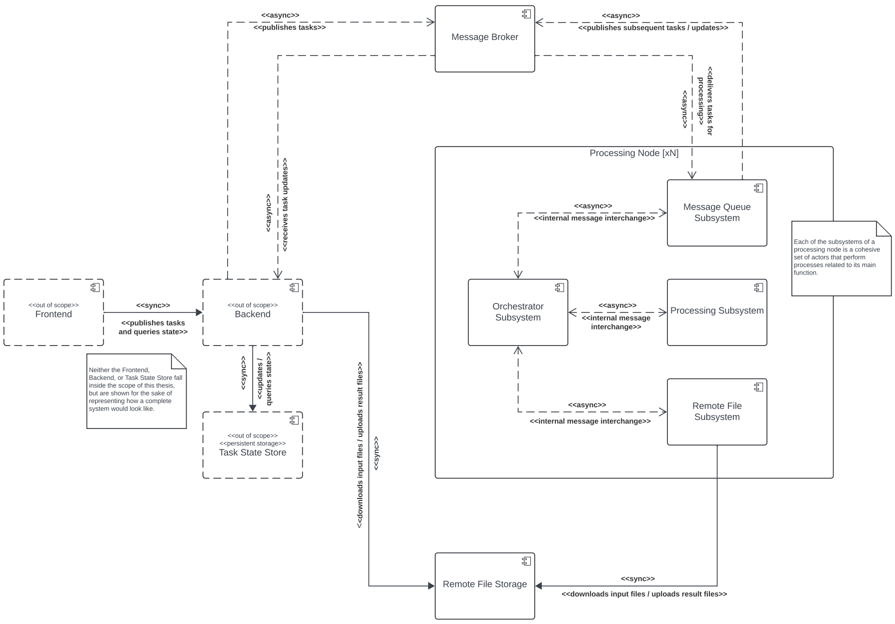
The system proposed in this work is designed to coordinate the lifecycle of tasks submitted for processing across diverse and potentially heterogeneous execution environments. These tasks may vary in type, required validation procedures, and processing stages, and may ultimately target physical or virtual infrastructures. The core responsibility of the system lies not in performing the actual task logic, but in managing task progression, routing, and persistence across a sequence of processing steps.

At its core, the system consists of a coordination layer that communicates with independently deployed processing nodes. These nodes handle specific processing stages and are responsible for interacting with the required execution environments, whether simulated, virtual, or physical. Task submissions are assumed to originate from a user-facing interface or integration layer, which stores associated task files in a remote file repository and notifies the system via message queue. The system then orchestrates the processing of each task by forwarding it through the appropriate nodes, recording the results, and issuing follow-up actions based on the stage results.

The coordination logic is built around a messaging-based architecture that allows asynchronous task routing, decoupling components, and extensibility in how nodes are added or removed. The processing nodes implement a common base interface that governs communication with the coordination system but remain agnostic to the global logic of other stages. This design allows for plug-in development of task-specific execution logic while preserving uniformity in system behavior.

Although certain components such as user-facing interfaces or integration with physical devices are outside the scope of this thesis, the system is intentionally designed to be extensible in those directions. The focus of this work is to define and validate an architectural model that supports task ingestion, state tracking, multistage processing coordination, and error recovery, providing a robust foundation for future system extensions.

5.2 Architectural Components


<<async>>
<<publishes tasks>>
Message Broker
<<async>>
<<publishes subsequent tasks / updates>>
<<async>>
<<delivers tasks for processing>>
<<async>>
<<receives task updates>>
Processing Node [xN]
Message Queue Subsystem
<<async>>
<<internal message interchange>>
<<out of scope>>
Frontend
<<sync>>
<<publishes tasks and queries state>>
<<out of scope>>
Backend
Orchestrator Subsystem
<<async>>
<<internal message interchange>>
Processing Subsystem
<<sync>>
<<updates / queries state>>
<<downloads input files / uploads result files>>
<<sync>>
<<out of scope>>
<<persistent storage>>
Task State Store
<<async>>
<<internal message interchange>>
Remote File Subsystem
Remote File Storage
<<sync>>
<<downloads input files / uploads result files>>

Neither the Frontend, Backend, or Task State Store fall inside the scope of this thesis, but are shown for the sake of representing how a complete system would look like.

Each of the subsystems of a processing node is a cohesive set of actors that perform processes related to its main function.

Figure 5.1: System architecture overview

5.2.1 Frontend

Responsibility. The Frontend represents the user-facing interface through which tasks are submitted to the system and results are queried. In a complete deployment scenario, it would allow students or users to interact with the orchestration layer by uploading files, selecting task parameters, and monitoring execution outcomes.

Position in the System. The Frontend resides at the outermost boundary of the system architecture. It acts as the entry point for task submission and the origin of user-driven interactions, forwarding task data and metadata to the Backend.

Interactions. The Frontend communicates exclusively with the Backend component, invoking operations such as task submission and state queries. It does not interact directly with the message broker, Processing Nodes, or any lower-level components of the orchestration architecture.

Design Rationale. Although the Frontend is not implemented or evaluated as part of this thesis, it is included in the architectural diagram to provide a complete representation of the envisioned system. Its presence highlights the separation of concerns between the user interface and the orchestration logic. By explicitly marking it out of scope, the design remains focused on the architectural core of the system while acknowledging the role the Frontend would play in a production environment.

5.2.2 Backend

Responsibility. The Backend is responsible for receiving task submissions from the Frontend, preprocessing them if necessary, and forwarding them to the message broker for asynchronous processing. It also handles task status queries and result retrieval requests from the Frontend. In addition to message handling, the Backend manages task-related files as it uploads the artifacts submitted by users to the Remote File Storage for later processing, and it retrieves the corresponding output files once tasks are completed so that users can access their results.

Position in the System. The Backend acts as an intermediary layer between the user interface and the orchestration infrastructure. It translates user-initiated operations into internal system events and coordinates data movement, while abstracting infrastructure complexity from the end user.

Interactions. The Backend interacts with four components:

- It receives task submissions and query requests from the Frontend.
- It publishes task metadata to the message broker using an asynchronous and message-based pattern.
- It queries the Task State Store in a synchronous manner to fetch metadata or status when requested.
- It performs synchronous file transfers with the Remote File Storage, both uploading input artifacts and downloading result files upon request.

Although the Backend may internally handle these interactions using concurrent or non-blocking mechanisms, the nature of its communication with the state store and file storage is inherently synchronous: it must obtain a definitive response before continuing its own operation.

Design Rationale. The Backend is not implemented in the current version of the system and is therefore marked as out of scope. However, its role is integral to the overall system architecture. It enables a clean separation between user interaction and orchestration logic and consolidates responsibility for file handling, task submission, and result retrieval in a centralized interface.

5.2.3 Task State Store

Responsibility. The Task State Store is responsible for maintaining the metadata associated with each task submitted to the system. This includes information such as task identifiers, submission timestamps, current execution status, error reports, and result availability. Its primary role is to provide a consistent and queryable source of truth about the state of all tasks managed by the orchestration system.

Position in the System. The Task State Store is on the Backend side of the architecture. It is accessed exclusively by the Backend component and is not exposed to Processing Nodes or orchestration components. It serves as the Backend's persistent memory for tracking task lifecycles and responding to user queries about task progress and results.

Interactions. The Backend communicates with the Task State Store through synchronous queries and updates. When a new task is submitted, an entry is created in the store. As task updates arrive via the message broker, the Backend updates the store accordingly. When a user requests the status or result of a task, the Backend queries the store to retrieve and present the relevant information.

Design Rationale. The Task State Store is modeled in the architecture to represent a critical component for maintaining task-level observability and traceability. Although it is not implemented within the scope of this thesis, its inclusion reflects the need for a persistent, queryable state Backend in systems that support asynchronous processing. By restricting access to the store to the Backend only, the design enforces a clear separation of concerns: Processing Nodes remain stateless with respect to task lifecycle management, and the Backend acts as the sole coordinator for user-visible state. This structure also simplifies future extensions such as auditing, analytics, and historical query support.

5.2.4 Remote File Storage

Responsibility. The Remote File Storage component is responsible for the storage of input artifacts submitted by users and output files generated during task processing. It serves as a shared repository accessible by both the Backend and the Processing Nodes, enabling a decoupled mechanism for exchanging large files without embedding them in task messages.

Position in the System. The Remote File Storage is located outside the core orchestration logic. It sits at the edge of the system and is accessed by both the Backend and the Processing Nodes. The Backend uploads input artifacts at the time of task submission and downloads output artifacts when the user requests the results. Processing Nodes retrieve these inputs before execution and upload the corresponding results upon completion.

Interactions.

- The Backend performs synchronous uploads of user-submitted files at submission time and downloads result files when the user requests them.

- The Remote File Subsystem within each Processing Node performs synchronous downloads to retrieve input artifacts and uploads result files after processing is complete.

Although all file operations are synchronous at the protocol level (e.g., FTP), their invocation is often wrapped in concurrent logic within Backend or node actors to avoid blocking system behavior.

Design Rationale. Including a separate Remote File Storage component decouples the handling of potentially large artifacts from the core messaging and orchestration infrastructure. This prevents payload bloat in task messages and allows independent scaling and optimization of file handling operations. The architecture also reflects a key assumption: artifact management is orthogonal to task logic and should be treated as an infrastructure service, not a core processing concern.

5.2.5 Message Broker

Responsibility. The message broker is responsible for facilitating asynchronous communication between decoupled components of the system. It acts as the central coordination mechanism through which tasks are submitted, routed, updated, and, in some cases, chained into follow-up tasks. Its role is to buffer messages, ensure reliable delivery, and enable the system to operate in a distributed and concurrent fashion without requiring direct coupling between producers and consumers.

Position in the System. The message broker occupies a central position in the architecture. It connects the Backend (which publishes new tasks and listens for updates) to the Processing Nodes (which consume, process, and publish results or follow-up messages). It also serves as the primary communication mechanism among internal node components in cases where inter-node messaging patterns are needed.

Interactions. The message broker handles three primary flows of communication:

- The Backend publishes new task submission messages to the broker, which distributes them to available Processing Nodes.

- Processing Nodes consume these task messages and may publish updates or follow-up tasks back to the broker.
- The Backend listens for updates or results emitted by the nodes in order to update task metadata and notify users.

All communication through the message broker follows a publish-subscribe pattern and is asynchronous by design. Message delivery is decoupled from processing, enabling system resilience and scalability under variable load conditions.

Design Rationale. The message broker is a fundamental element of the message-driven architecture of the system. Its use enables full decoupling between the Frontend/Backend and the processing layer, making it possible to dynamically scale Processing Nodes, introduce new task types, and tolerate transient failures without compromising message integrity. By abstracting the transport and delivery of messages, the broker also simplifies error handling and retry mechanisms. Its central position in the design reflects a deliberate architectural choice to favor asynchronous, loosely coupled coordination over tightly bound service invocation. This aligns with the need for scalability, fault isolation, and extensibility in heterogeneous task execution environments.

5.2.6 Processing Node

Responsibility. A Processing Node is responsible for executing tasks received from the message broker. It encapsulates the full internal pipeline required for task handling, from message reception and artifact retrieval to execution and result publication. Each node operates autonomously and includes all necessary subsystems to manage task flow end-to-end. The node is designed to handle tasks in a modular and fault-isolated manner, ensuring robustness, scalability, and extensibility.

Position in the System. Processing Nodes are distributed components within the system and act as task consumers. They connect directly to the message broker to receive task metadata and to publish execution outcomes. They also communicate with the Remote File Storage system to retrieve input files and upload results. Internally, each node consists of multiple subsystems that collaborate to perform all necessary operations without requiring centralized coordination.

Interactions. Each Processing Node interacts externally with:

- The Message Broker, to consume tasks and publish updates.
- The Remote File Storage, to download input files and upload results.

Internally, the node contains four main subsystems:

- The Orchestrator Subsystem, which initializes and supervises the internal architecture.
- The Message Queue Subsystem, which manages broker communication.
- The Remote File Subsystem, which handles file transfers to and from external storage.
- The Processing Subsystem, which executes task logic based on provided metadata and local artifacts.

Each subsystem is initialized during node startup and operates independently, with well-defined responsibilities and communication boundaries. Once a task enters the node and passes through the preparation stages, it is executed without further interaction between subsystems during its processing phase.

Design Rationale. The Processing Node is designed as a self-contained modular unit that follows the principles of isolation, supervision, and single responsibility. Its internal architecture emphasizes clear separation between coordination, communication, execution, and file handling. This structure improves resilience, simplifies error handling, and supports future extensibility, for example, by allowing the introduction of new task types or execution environments without architectural disruption. By ensuring that each subsystem handles its own domain concerns independently, the node architecture provides a scalable foundation for distributed task orchestration in heterogeneous environments.

Orchestrator Subsystem

Responsibility. The Orchestrator Subsystem is responsible for initializing and coordinating the internal components of the Processing Node. It defines the structure of the node by instantiating and managing the core subsystems responsible for message handling, task execution, file management, and system regulation. Although it does not process tasks directly, it ensures that each subsystem operates as intended and enforces clear lifecycle and supervision boundaries across the node.

Position in the System. This subsystem operates entirely within the Processing Node and acts as the logical root of its internal architecture. It is the first component initialized at startup and remains active throughout the lifetime of the node. It is not exposed to external components and does not perform direct message exchange with the Backend or broker. Instead, it coordinates the internal structure that enables the node to function autonomously.

Interactions. The Orchestrator Subsystem instantiates and supervises the following internal subsystems:

- The Message Queue Subsystem, which handles all communication with the message broker.
- The Processing Subsystem, which executes tasks based on prepared metadata and artifacts.
- The Remote File Subsystem, which manages file downloads and uploads from external storage.

The Orchestrator Subsystem does not interact with other subsystems at runtime beyond initial configuration and regulation feedback. Each subsystem operates independently after initialization.

Design Rationale. This subsystem serves as the architectural backbone of the node. By delegating functionality to specialized subsystems and centrally managing their lifecycle, the design promotes fault isolation, scalability, and modularity. Including a regulation mechanism as part of this subsystem allows dynamic control over task intake without coupling monitoring logic to task execution or communication layers. This structure aligns with architectural principles of separation of concerns and provides a clean foundation for internal supervision and extensibility.

Message Queue Subsystem

Responsibility. The Message Queue Subsystem is responsible for managing all interactions between the Processing Node and the message broker. It handles the reception of incoming task messages, the delivery of task-related updates, and any transformations required to adapt internal task representations to message formats suitable for transmission. This subsystem acts as the gateway for all task-related communication between the node and the broader system.

Position in the System. This subsystem is located at the external boundary of the Processing Node and is the only component within the node that communicates directly with the message broker. It receives messages forwarded to the node for processing and publishes messages to other system components via the broker, such as task completion signals or follow-up task notifications. Internally, it is initialized and managed by the Orchestrator Subsystem.

Interactions.

- Receives task messages from the message broker and forwards them to the orchestration logic within the node.
- Publishes task status updates or follow-up messages to the message broker for downstream consumption (e.g., by the Backend or other Processing Nodes).
- Transforms message formats if needed, ensuring compatibility between external message schemas and internal task representations.

This subsystem operates independently once it is initialized and does not participate in task execution or file management. It may buffer or adapt messages, but does not introduce execution delays or state coupling with other subsystems.

Design Rationale. The Message Queue Subsystem encapsulates all broker-facing communication within a dedicated, isolated layer. This design minimizes the exposure of the internal node structure to the external messaging infrastructure and allows for asynchronous, decoupled communication. By treating message I/O as a separate concern from processing or orchestration, the architecture improves clarity, testability, and resilience.

Remote File Subsystem

Responsibility. The Remote File Subsystem is responsible for managing all file operations required by the Processing Node. This includes downloading input artifacts prior to task execution and uploading result artifacts upon completion. Its role is to abstract file transfer logic away from the execution and orchestration components, ensuring that tasks have access to all necessary data while preserving a clean separation between data handling and processing logic.

Position in the System. This subsystem is an internal component of the Processing Node and operates under the supervision of the Orchestrator Subsystem. It is the only component within the node that interacts with the external Remote File Storage system. It performs file transfers before and after task execution, ensuring that the required inputs are present and that outputs are made available for retrieval by the Backend.

Interactions.

- Receives requests from internal components (typically during task preparation and finalization) to download or upload files.
- Performs synchronous interactions with the external Remote File Storage system to carry out file transfers.
- Notifies relevant internal subsystems when file operations complete, allowing task processing to proceed or results to be finalized.

This subsystem does not participate in task execution or messaging logic. It operates independently and reacts only to file transfer requests issued by other subsystems during the task lifecycle.

Design Rationale. Separating file transfer logic into its own subsystem improves the modularity and testability of the overall system. It ensures that file operations, which may involve latency, I/O errors, or protocol-specific behavior, are encapsulated and do not interfere with orchestration or processing logic. The subsystem also supports clearer lifecycle boundaries: tasks must be fully prepared before entering the processing stage, and results are not considered finalized until they are successfully uploaded. This design enables the use of different storage Backends or file transfer mechanisms in the future with minimal architectural impact.

Processing Subsystem

Responsibility. The Processing Subsystem is responsible for executing tasks using the input metadata and artifacts that have already been prepared by the orchestration logic and file subsystem. It encapsulates the logic required to process a task based on its type and content, and is designed to operate in isolation from the rest of the node during execution. Once a task enters this subsystem, it proceeds through execution without mid-process interaction with other internal components.

Position in the System. This subsystem is the final stage in the task pipeline within the Processing Node. It is supervised by the Orchestrator Subsystem and operates only after the message queue and Remote File Subsystems have completed their respective responsibilities. It does not interact with external services through the actor system, though it may internally invoke scripts, programs, or infrastructure APIs as part of the task logic.

Interactions.

- Receives fully prepared task metadata from the orchestration layer, including paths to locally available input files.
- Performs processing based on the task's defined logic, configuration, and content.
- Writes result artifacts to local storage once execution completes.
- Emits terminal signals indicating task outcome (e.g., success, failure, timeout) to orchestration for follow-up actions, such as result upload or message publication.

The Processing Subsystem does not request additional resources or perform coordination during execution. Its operation is designed to be self-contained per task instance.

Design Rationale. This subsystem enforces a strict boundary between task preparation and processing, which simplifies error handling, concurrency, and lifecycle management. By requiring all necessary inputs to be present before processing begins, the subsystem guarantees that tasks are deterministic and isolated from orchestration concerns. It also ensures that processing logic remains focused and testable while supporting flexibility: tasks may internally interact with external systems, tools, or physical infrastructure, without introducing message-passing dependencies into the actor system. This structure supports the long-term goal of integrating diverse types of tasks and environments while maintaining a clean and robust architectural separation of concerns.

5.3 Task Flows

This section describes how tasks move through the system from submission to completion. It builds on the architectural components defined in the previous section by explaining the runtime behavior of the system when processing individual tasks. The focus is on the lifecycle of a single task, how tasks are chained across processing stages, and how the architecture supports variations in execution paths depending on task characteristics or results. The overall objective is to clarify the flow of control and data through the system and how processing nodes interact with other components in a loosely coupled and message-driven manner.

5.3.1 Task Lifecycle

A task in the system follows a well-defined lifecycle composed of distinct stages, starting from submission by the user and ending with the publication of results and possible generation of follow-up tasks. This subsection describes the nominal sequential path taken by a single task through the system.

The lifecycle begins when a user (e.g., a student) submits a task through the Frontend interface. The Frontend sends the submission request to the Backend, which is responsible for registering the task, storing associated metadata, and uploading any task-related files to the remote file storage. The backend then creates a corresponding entry in the task state store and publishes a message to the message broker to enqueue the task for processing.

Once enqueued, the message broker delivers the task metadata to an available processing node subscribed to the corresponding queue. Upon receiving the message, the processing node downloads the referenced input artifacts from the remote file storage and proceeds to execute the task based

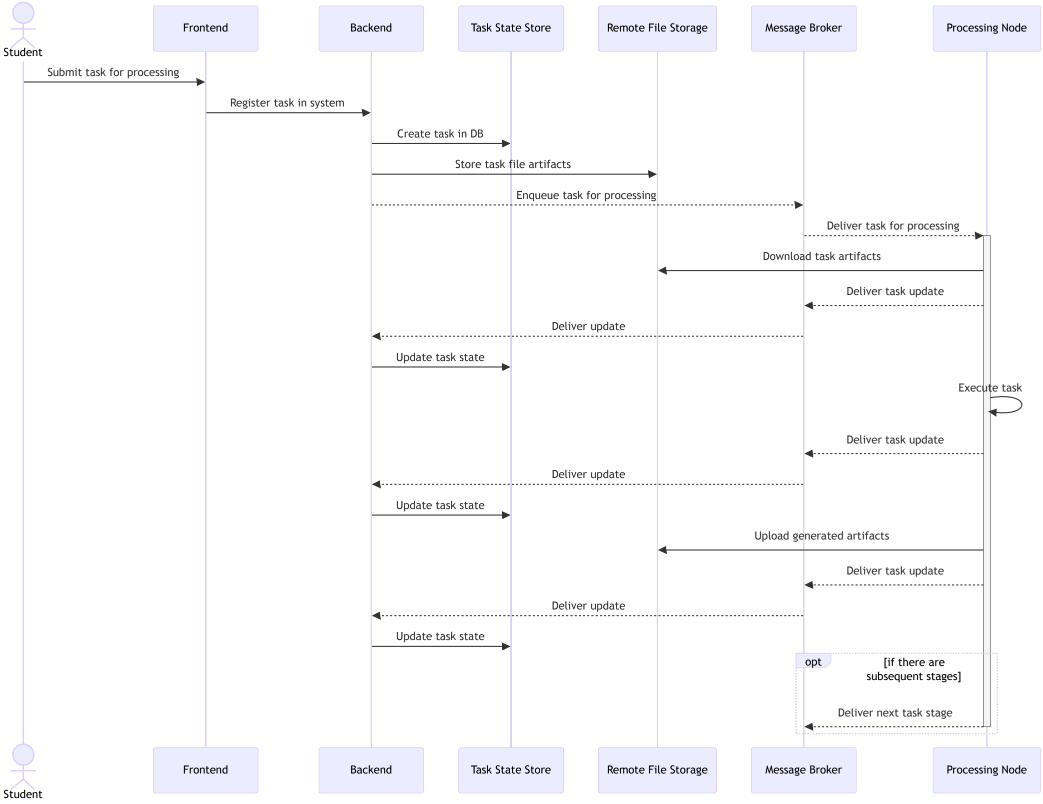


Figure 5.2: Basic flow of a task inside the system

on the provided metadata. Execution is performed in isolation, without mid-process messaging or coordination with other components. Upon completion, the node uploads the generated result artifacts back to the remote storage and sends a terminal update message to the message broker indicating the task's outcome (e.g., success or failure).

The backend, which subscribes to updates, receives the task completion notification and updates the task's status in the Task State Store. From this point, the task can be queried by the user via the Frontend and results can be retrieved. As the system supports multi-stage workflows, the same completion event can trigger the submission of a new, subsequent task, as described in the next subsection.

The complete lifecycle is illustrated in Figure 5.2, which depicts the ordered sequence of events across all participating components.

5.3.2 Task Chaining and Routing Decisions

The system supports multistage workflows through predefined routing logic embedded in task metadata. Once a task is submitted and enters the system, its complete execution path, that is, the sequence of processing stages it should follow, is fully determined at the time of submission. This decision is made by the Backend based on static workflow definitions or task-specific configuration, and it is encoded directly into the message metadata associated with the task.

Each task message contains a mapping of outcome-based routes, allowing both success and failure paths to be explicitly defined. When a processing node completes a task, it reads the routing metadata and, depending on the execution outcome, publishes a new message to the corresponding queue for the next stage. Both success and failure outcomes may themselves have subsequent routes, enabling branching workflows that form a tree of possible execution paths. If no route is defined for a given outcome, the task execution is terminated and a terminal status update is sent to the Backend.

This model prevents the Backend from influencing a task's flow after submission, ensuring immutability and consistency throughout the system. It also avoids requiring processing nodes to have knowledge of the system's global routing logic, which would tightly couple them to the orchestration layer. By enforcing a strict separation of concerns, submission time routing

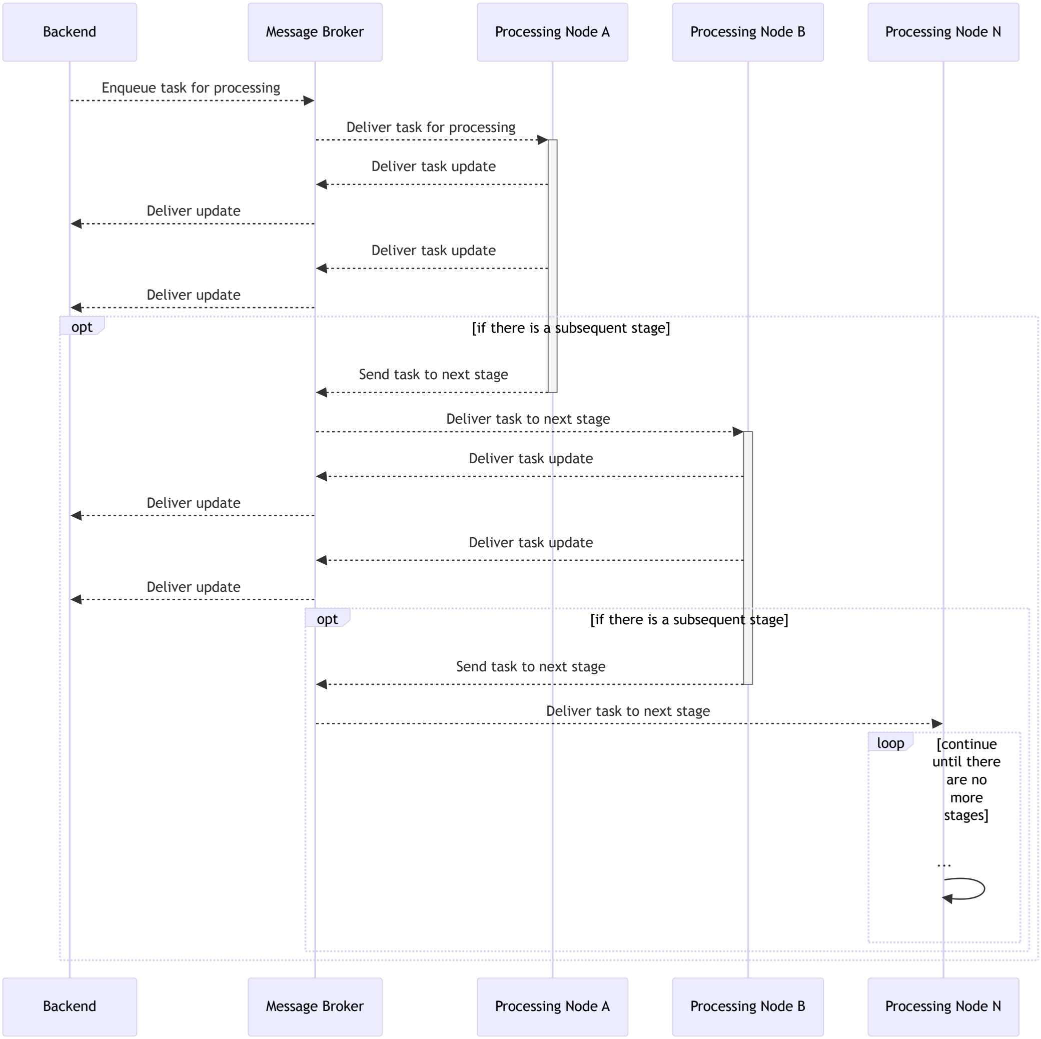


Figure 5.3: Multiple task chaining

definition by the Backend and runtime execution by the nodes, the architecture maintains a clean and scalable design.

This approach has several advantages:

- **Predictability:** All routing is deterministic and declarative
- **Modularity:** Nodes operate independently, without requiring awareness of other stages
- **Observability:** Each task's intended flow, for both success and failure outcomes, is explicitly available in its metadata
- **Flexibility:** Supports conditional branching on both outcomes without introducing runtime coupling

This routing mechanism is illustrated in Figure 5.3, where the next step is determined by the outcome of the current stage, and both success and failure paths can lead to further processing. This design supports a broad range of workflows while preserving subsystem isolation and ensuring predictable task behavior.

This approach allows the system to support multiple workflows by encoding different routing sequences, including both success and failure branches, at submission time. While the routes are determined before execution begins, this still enables dynamic progression through the workflow based on runtime outcomes. This design preserves subsystem isolation, ensures predictable task behavior, and allows branching without requiring the nodes to hold global routing logic. Future work may expand on this by introducing runtime adaptation or conditional routing rules generated during execution, but the current architecture already supports conditional branching for both outcomes through submission-time definitions.

5.4 Error Handling and Recovery Design

The proposed architecture is designed with a focus on fault isolation, predictable failure propagation, and structured recovery strategies. Given the distributed and asynchronous nature of the system, failure is not treated as an exceptional event, but as an expected and manageable condition. Each component, whether internal to a processing node or external in the messaging infrastructure, is designed to recover independently or fail in a way that does not compromise the overall system.

This fault-aware mindset is reflected in the system's modular design. Processing nodes are structured around independent subsystems that communicate exclusively through asynchronous message passing, without shared state or direct dependencies. This design ensures that failures, whether due to execution errors, file access issues, or transient messaging problems, are contained within their respective boundaries. Each subsystem is capable of terminating or recovering independently, without requiring coordinated intervention. The message-driven architecture further reinforces this separation, allowing the system to absorb and tolerate faults without cascading effects or systemic disruption.

5.4.1 Actor Supervision Strategies

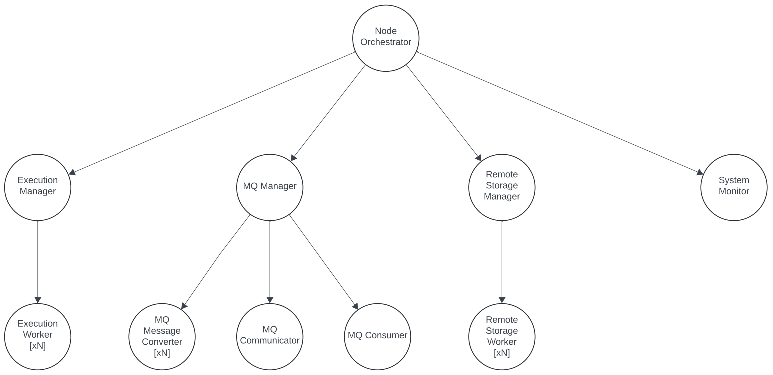
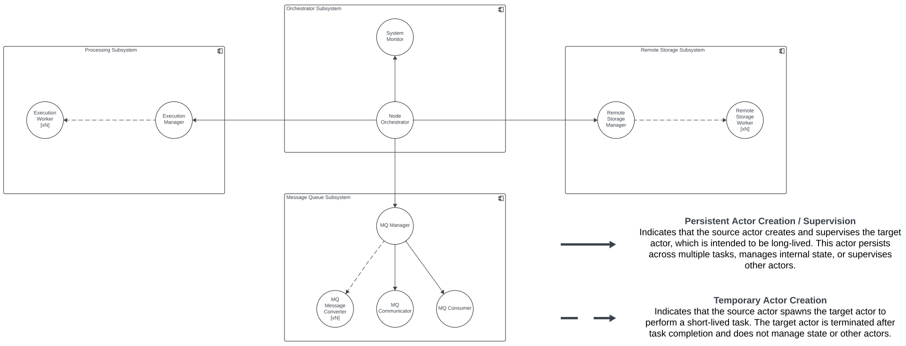


Figure 5.4: Actor hierarchy

The architecture of each processing node is based on the actor model, which provides native support for asynchronous message passing, hierarchical supervision, and fault isolation, core requirements for the system's design. At the core of each node is a supervision hierarchy that governs the lifecycle and isolation of internal components. This hierarchy, depicted in Figures 5.4 and 5.5, defines a clear parent-child structure in which each long-lived subsystem, such as message handling, file management, and execution coordination, is supervised by the Node Orchestrator. These subsystems, in turn, manage the



Persistent Actor Creation / Supervision
Indicates that the source actor creates and supervises the target actor, which is intended to be long-lived. This actor persists across multiple tasks, manages internal state, or supervises other actors.

Temporary Actor Creation
Indicates that the source actor spawns the target actor to perform a short-lived task. The target actor is terminated after task completion and does not manage state or other actors.

Figure 5.5: Actors that make up each component

creation and control of short-lived components responsible for task-specific operations.

In this architecture, supervision is not limited to a single fault-handling strategy. Depending on the nature of the failure and the role of the component, supervisors can choose to terminate, restart, or resume their children. The supervision tree is designed to allow localized recovery without affecting any sibling subsystems or the global behavior of the node. For example, a failed file transfer will only affect the corresponding file worker and will not disrupt message handling or execution logic.

Persistent components, which maintain node-level behavior across multiple tasks, are initialized at startup and designed to withstand or recover from failures in their child components. In contrast, task-specific components are created on demand to process individual units of work and are expected to fail fast and terminate cleanly in the event of an error.

This supervision structure enables the system to maintain resilience under failure conditions, support clear separation of concerns, and ensure that non-fatal errors do not cascade through the node. It also simplifies the implementation of node-internal logic by offloading error propagation and recovery behavior to the supervision model itself, in line with established actor-based system design principles.

A high-level view of the actor supervision structure is presented in Figure 5.4, which illustrates the hierarchical relationships within a processing node.

It shows how the Node Orchestrator supervises each of the core subsystems and how those subsystems maintain internal fault boundaries by supervising their own task-specific components. This structure reflects the architectural emphasis on modular fault containment and local recovery.

A more detailed breakdown is presented in Figure 5.5, which shows the internal actor composition of each subsystem within the processing node. The diagram clarifies which actors belong to which components and distinguishes between long-lived actors that persist across tasks and short-lived actors that are created per task instance. It reflects the strategy for isolating failure by containing task-specific logic in disposable actors, while delegating lifecycle management to persistent supervisory components. This composition pattern reinforces system resilience and supports fault recovery through clear supervision boundaries and predictable actor lifecycles.

The internal actor hierarchy of the processing node is made up of a combination of persistent supervisors and short-lived task-specific workers. Each actor is assigned a focused responsibility within the system and participates in a structured supervision tree to ensure fault isolation and modular behavior. The following is a brief overview of the primary actors and their roles within the node.

- **Node Orchestrator** — Root supervisor of the processing node. Initializes and supervises all major subsystems. Does not process tasks itself, but manages the internal structure and lifecycle of components.

- **Execution Manager** — Coordinates task execution. Receives fully prepared task metadata and delegates execution to a short-lived worker. Maintains high-level control of the processing flow within the node.

- **Execution Worker** — Executes a single task based on the provided metadata and input files. Operates in isolation. On completion or failure, sends a terminal message and terminates.

- **MQ Manager** — Handles communication with the message broker. Responsible for consuming incoming task messages and publishing task status updates. Supervises internal message-related logic.

- **MQ Message Converter** — Handles conversion between internal task representations and external message formats. Used for decoupling internal logic from broker-specific schemas.

- **MQ Communicator** — Responsible for sending messages to the broker, including status updates and follow-up tasks. May encapsulate broker API logic.

- **MQ Consumer** — Continuously listens for new task messages from a specific queue on the message broker. Upon receiving a message, it forwards it to the orchestrator.

- **Remote Storage Manager** — Coordinates file downloads and uploads to the remote storage system. Receives requests from other subsystems and supervises task-specific file workers.

- **Remote Storage Worker** — Performs a specific file transfer operation (download or upload). Terminates after completing the transfer. Faults are contained at this level.

- **System Monitor** — Observes node-level conditions such as task volume or system load. Provides feedback to the orchestrator to regulate task intake. It does not interfere with task execution logic.

5.4.2 Node-Level Fault Recovery

The system is designed to tolerate failures at the level of individual processing nodes without compromising the correctness or stability of the overall architecture. Each node is structured as a modular, self-contained unit with no persistent state across tasks. This design ensures that failure of a single node, whether due to task-level crashes, resource exhaustion, or infrastructure issues, has a limited blast radius and does not require system-wide intervention.

Processing nodes are initialized independently and communicate asynchronously with the rest of the system through the message broker and remote file storage. If a node fails during task execution, any task that was in progress may be lost; however, because the system does not rely on the node for persistent task tracking and because tasks are processed in isolation, such a failure is safely contained. The message broker does not assume task completion unless an explicit terminal message is received, allowing failed or incomplete tasks to be detected externally and retried if necessary.

Fault recovery at the node level consists primarily of component isolation and stateless design. Each task is executed in a short-lived and self-contained environment, and long-lived subsystems are designed to recover independently via actor supervision. If a subsystem fails internally, the supervisor can restart or replace it without impacting other components of the node. In the event of a complete node crash or shutdown, the system can redeploy a new instance of the node without coordination or state reconciliation, as all critical routing, state, and file artifacts are maintained externally.

This architectural approach reduces the need for coordinated recovery logic, avoids the complexity of distributed rollback, and allows for straightforward horizontal scaling. By treating each node as a stateless executor with internal fault boundaries and external interfaces, the system achieves both operational simplicity and robust fault containment.

5.4.3 Message Redelivery and Retries

The system relies on a message broker to decouple task submission from execution and to provide natural buffering between components. Given the asynchronous nature of task distribution, the design includes mechanisms to ensure that tasks are not silently lost and can be retried under certain failure conditions.

Task messages are published to the broker by the Backend and consumed by processing nodes. Once a message is delivered to a node, that node becomes responsible for processing the task and eventually reporting a terminal result. However, no task is considered complete unless an explicit success or failure message is published back to the broker. This behavior ensures that tasks interrupted by node crashes or internal errors do not disappear silently from the system.

Message redelivery is handled at the broker level. If a message is delivered but not acknowledged or followed by a terminal update within a predefined time window, it remains available in the queue or can be re-queued depending on broker configuration. The architecture assumes that processing nodes do not maintain exclusive locks or internal queues, so any node capable of handling a given task type can process re-delivered messages.

Retries are treated conservatively in this design. The system does not implement node-side retry logic; instead, the message broker's delivery semantics are relied upon to support limited redelivery in the event of consumer failure. This avoids duplicate processing by design, since tasks are assumed to be idempotent or externally tracked. More complex retry mechanisms, such as exponential backoff, dead-letter queues, or retry limits, are out of scope for this version of the system but can be introduced later without modifying the node architecture.

This approach strikes a balance between reliability and simplicity. By offloading retry mechanics to the messaging infrastructure and enforcing clear terminal signals from processing nodes, the system avoids unnecessary coupling and ensures that unprocessed or failed tasks remain visible and recoverable without centralized coordination.

5.4.4 Traceability and Observability

The architecture is designed to ensure traceability of task execution without increasing coupling or complexity across components. Rather than relying on direct communication between processing nodes and the Backend, the system centralizes all status updates and failure reports through the message broker. This decision supports architectural decoupling, allows for Backend unavailability, and ensures that updates are not lost if the Backend is temporarily offline.

Each processing node emits status updates based on the internal state of task execution. These updates are published asynchronously to a broker-managed queue and may include signals such as successful completion, failure, or timeout. By relying on the broker as an intermediary, the system ensures that all updates are durable and eventually delivered, regardless of Backend availability at the time of publication.

This messaging-based reporting approach simplifies node design. Nodes do not require HTTP clients, state retention, or retry logic to reach the Backend. Instead, they remain lightweight and stateless, with their only responsibility being to publish updates when available. The message broker acts as a buffer and transport mechanism, absorbing transient Backend failures and ensuring that traceability is preserved.

Observability in this design is focused on task-level outcomes, rather than fine-grained monitoring of internal execution steps. Although the architecture does not provide real-time telemetry or metrics from within the processing subsystems, it does guarantee a reliable and queryable view of task progress through status messages handled by the Backend. This model supports operational visibility without compromising fault isolation or increasing the surface area for failure within processing nodes.

5.5 Extensibility and Task Integration Model

A key architectural goal of the system is to support the integration of new task types without requiring structural changes to the overall design. Although the system does not implement runtime plug-and-play mechanisms, it is intentionally modular: task-specific logic is encapsulated within clearly bounded components, and task routing is handled externally through metadata and queue configuration. This section describes how the system enables the addition of new task types, outlines the boundaries of this extensibility model, and discusses the minimal changes required to support integration at the node level.

5.5.1 Task Logic Encapsulation

The system architecture is explicitly designed to separate task execution logic from the orchestration, messaging, and file handling layers. This separation is enforced through the internal structure of the processing node, where task execution is delegated to a dedicated subsystem that operates independently of the message queue and remote file subsystems. This allows each task type to be defined and handled in a self-contained manner.

Execution logic for a given task type is encapsulated within the processing subsystem, which interprets task metadata and executes the corresponding behavior using input files already prepared by the orchestration flow. Because the task logic is isolated from the rest of the node, extending the system to support a new task type involves only modifying or adding logic within this specific subsystem. The rest of the node, and the system as a whole, remains unaffected.

This encapsulation also supports flexibility in the way task logic is implemented. Nodes can be specialized to support only a subset of task types or generalized to handle multiple task types through branching logic within the processing subsystem. In both cases, task routing remains external to the node and is determined by metadata and message queue configuration, allowing processing logic to evolve independently of the orchestration mechanism.

By encapsulating task execution in isolated components and avoiding shared state or cross-subsystem dependencies, the architecture ensures that supporting new task types requires minimal coordination and avoids cascading changes. This structure simplifies the development, testing, and deployment of new task logic while preserving the integrity of the overall system.

5.5.2 Task Type Deployment Strategy

The deployment of new task types in the system follows a predictable and minimally invasive strategy, enabled by the modular and message-driven design of the architecture. Because task execution logic is isolated within the processing node and task routing is determined externally, integrating a new task type does not require modification of the Backend, message broker, or other system components outside the processing layer.

To support a new task type, three main actions are required:

- **Implementation of task-specific logic within the processing subsystem of the node.** This involves adding a new execution branch or handler capable of interpreting the corresponding task metadata and executing the associated logic.
- **Configuration of routing metadata in the Backend.** At submission time, the Backend must define the appropriate queue or sequence of queues that the task will traverse. This enables routing to the correct processing nodes according to the type and stage of the task.
- **Deployment or update of processing nodes.** Nodes that are expected to handle the new task type must be redeployed with the updated logic or newly provisioned as specialized instances. This step is straightforward, as each node operates independently and does not maintain a global state.

This strategy supports both specialized nodes, which are configured to handle a single task type, and general purpose nodes, which are capable of handling multiple task types based on internal dispatch logic. The choice between specialization and generalization can be made based on operational needs such as resource constraints, deployment environment, or fault isolation requirements.

The system's reliance on message-based routing and stateless node behavior ensures that changes related to new task types remain localized. There is no need for coordinated redeployment of Backend components or broker configuration beyond adding or binding to a new queue. As a result, the system can scale and evolve with relatively low overhead, supporting the incremental integration of new workflows without requiring structural changes to the overall architecture.

Chapter 6

Implementation

This chapter presents the implementation of the proposed architecture described in Chapter 5, realized as a functional prototype and deployed in a controlled environment. The objective is to demonstrate how the design principles are translated into a working system and to assess the practical feasibility of the architecture under representative operating conditions.

The implementation targets a simplified yet realistic scenario inspired by the context introduced in Chapter 2, in which task orchestration is applied to multistage workflows derived from student-submitted assignments. Although not all components of the full envisioned system were implemented, the prototype captures the architectural core necessary to validate key properties such as modularity, fault tolerance, and integration capacity.

6.1 Technology Selection and Mapping

The technologies selected to implement the proposed architecture are chosen based on their alignment with the design principles of the system: message-driven coordination, modular component boundaries, and stateless processing. Rather than focusing on convenience or performance benchmarks, the choices reflect architectural compatibility, long-term maintainability, and extensibility potential.

6.1.1 Programming Language: Scala

Scala was selected as the primary language due to its native integration with the Akka toolkit, which supports the actor-based concurrency model of the system. Scala's concise syntax and functional programming capabilities make it well-suited for building reactive, message-oriented systems. Furthermore, Scala runs on the Java Virtual Machine (JVM), enabling seamless interoperability with Java libraries. This provides access to a mature ecosystem of tools and avoids the need to reimplement low-level functionality that may not yet exist in the Scala ecosystem. In the context of future extensibility, this interop allows third-party task logic or utilities to be integrated with minimal effort, whether they are written in Scala or Java.

6.1.2 Actor Framework: Akka/Pekko

The internal behavior of the processing nodes is implemented using the actor model, with either Akka or Apache Pekko as suitable frameworks. Akka was initially considered due to its mature actor model, strong fault-tolerance primitives, and robust supervision strategies. However, following its transition to a Business Source License (BSL), Apache Pekko, an open-source community-driven fork of Akka maintained under the Apache 2.0 license, emerged as a compatible alternative. Both frameworks provide native abstractions for actor hierarchies, message passing, and system-level resilience, all of which directly map to the architecture described in this chapter. Although other actor-model frameworks were considered, such as Elixir's native actor system, the decision was made in favor of a JVM-based solution due to its extensive documentation, production use cases, and ecosystem tooling. Elixir, while architecturally aligned, presents adoption risks due to a smaller ecosystem and reduced interoperability with mainstream libraries.

6.1.3 Message Broker: RabbitMQ

RabbitMQ was selected as the message broker due to its ease of setup, flexibility of configuration, and routing intelligence. It operates as a broker-managed queueing system, where the broker is responsible for maintaining the queue state, message ordering, delivery guarantees, and routing logic. This model complements the stateless node design by offloading delivery tracking and persistence to the broker. In contrast, alternatives like Kafka rely on a consumer-managed model, where each processing node must maintain state regarding message offsets and partitions. Such a design would introduce unnecessary complexity into the node logic and violate the architecture's principle of stateless and disposable task execution. RabbitMQ's support for multiple queues, topic-based routing, and acknowledgment mechanisms make it a natural fit for the system's multistage and decoupled task flow.

6.1.4 File Storage: FTP-based Remote Storage

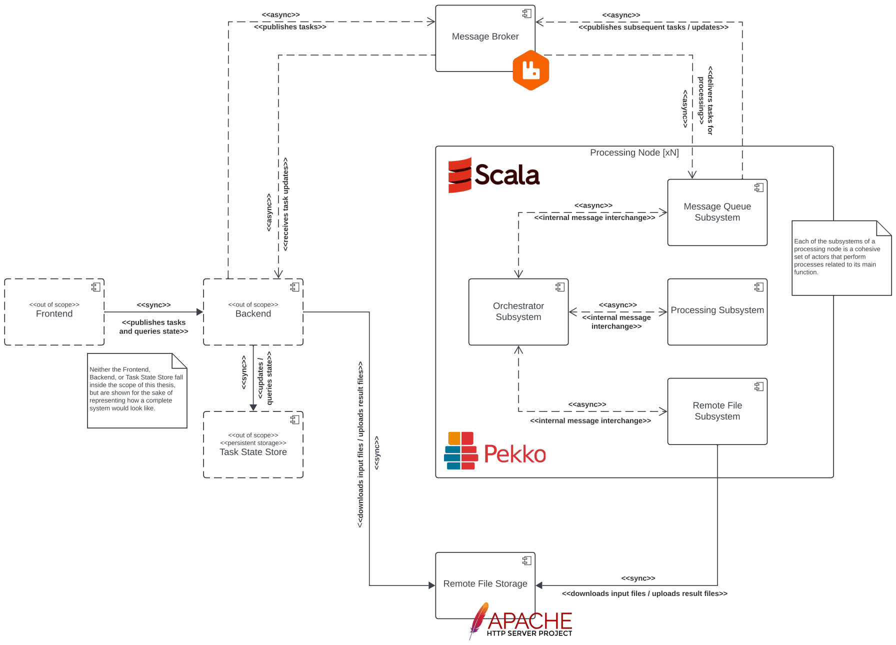
The remote file storage layer is abstracted from the core system and interacts with processing nodes only for artifact retrieval and result upload. The initial implementation is expected to use a simple FTP server due to its accessibility and familiarity. However, the architecture is storage-agnostic, and this component can be replaced with any file transfer protocol or storage backend, such as S3 or NFS, without requiring changes to the processing logic or coordination model. File storage serves as a supporting mechanism and does not affect the architectural properties of routing, execution, or fault tolerance.

A depiction of the technologies chosen and where they were implemented can be seen in Figure 6.1.

6.2 Representative Workload Selection

To evaluate the orchestration capabilities of the prototype, three containerized executors were deployed, each representing a distinct type of educational task:

- **Cypress-based Grammar Checker** – Performs automated validation of text-based submissions by detecting grammatical errors.
- **Cypress-based Task Runner** – Executes the tasks submitted to it by the orchestration system, using Cypress as the automation framework.
- **G-code Execution Engine** – Interprets and validates G-code instructions, mainly used in this implementation to run 3D printer simulations.



<<async>>
<<publishes tasks>>
Message Broker
<<async>>
<<publishes subsequent tasks / updates>>
<<delivers tasks for processing>>
<<async>>
<<async>>
<<receives task updates>>
Processing Node [xN]
Scala
<<async>>
<<internal message interchange>>
Message Queue Subsystem
Each of the subsystems of a processing node is a cohesive set of actors that perform processes related to its main function.
<<out of scope>>
Frontend
<<sync>>
<<publishes tasks and queries state>>
<<out of scope>>
Backend
Orchestrator Subsystem
<<async>>
<<internal message interchange>>
Processing Subsystem
Neither the Frontend, Backend, or Task State Store fall inside the scope of this thesis, but are shown for the sake of representing how a complete system would look like.
<<sync>>
<<updates / queries state>>
<<downloads input files / uploads result files>>
<<sync>>
<<async>>
<<internal message interchange>>
Remote File Subsystem
<<out of scope>>
<<persistent storage>>
Task State Store
Pekko
Remote File Storage
<<sync>>
<<downloads input files / uploads result files>>
APACHE
HTTP SERVER PROJECT

Figure 6.1: Technologies used for the prototype

These workloads were selected for three main reasons:

1. **Relevance to the Target Context** – They represent classes of assignments common in the academic environments described in Chapter 2, including programming, automated testing, and hardware-oriented workflows. Their inclusion directly reflects the diversity of tasks and processing requirements that teaching staff face in real courses.
2. **Feasibility for a Prototype** – All three can be containerized, have minimal external dependencies, and can run safely within the controlled environment of the implementation.
3. **Representation of Multistage Workflows** – Each workload can be integrated into multiphase task pipelines involving validation, simulation, and execution steps, allowing the architecture's routing and lifecycle management features to be exercised in scenarios analogous to those encountered in real educational settings.

The execution environments for these tasks were developed by undergraduate students as part of their own undergraduate thesis projects. The author provided technical leadership and close collaboration throughout these efforts, ensuring that design decisions and implementation choices aligned with the architectural principles presented in this work. As a result, all executors integrated seamlessly into the orchestration system, requiring only minimal adaptation to achieve full operational compatibility. Detailed descriptions of the design and implementation of these execution environments can be found in their respective undergraduate thesis project (Castro Iregui et al., 2025).

Although these executors do not interact with physical laboratory hardware, they provide a realistic approximation of the processing patterns, execution constraints, and feedback requirements present in real educational use cases. In particular, their orchestration within the prototype demonstrates the architecture's potential to automate the execution of diverse assignments, reduce the manual workload of teaching assistants and professors, eliminate the need for repeated environment-specific debugging, and accelerate feedback cycles for students. This direct alignment with the challenges and goals outlined in Chapter 2 reinforces the validity of the prototype as a practical embodiment of the proposed architecture.

A representation of a complete workflow based on the selected workloads is depicted in Figure 6.2.

6.3 Implementation Constraints

The development of the prototype was shaped by several constraints that influenced both the scope of the system and the design decisions made during implementation:

- **Time** – The implementation was carried out within the limited timeframe of a master's thesis, requiring a focus on building a functional and demonstrable architecture rather than a fully featured production system.
- **Maintainability** – As the system is expected to be maintained primarily by undergraduate students after its initial development, emphasis was placed on creating clear, modular, and well-documented components. This reduces the learning curve for future maintainers and lowers the risk of introducing defects during future modifications.
- **Practicality** – Certain architectural possibilities, such as integration with complex or high-risk environments (e.g., physical laboratory equipment), were intentionally excluded from the prototype. Implementing and validating such integrations would have required significantly more time, specialized resources, and safety measures, which were beyond the scope of this work.

Although these constraints imposed clear boundaries on the project, they also shaped its priorities in a way that strengthened its relevance to the real-world context described in Chapter 2. The emphasis on maintainability ensures that the system can be realistically adopted and extended in an academic environment where staff turnover is frequent and development is often student-led. The deliberate focus on containerized self-contained execution environments allowed the prototype to capture the diversity of real assignment workflows without incurring the risks and costs of integrating physical equipment at this stage. In this way, the constraints served not only as limitations, but also as guiding factors that aligned the prototype with the operational realities of its intended educational setting.

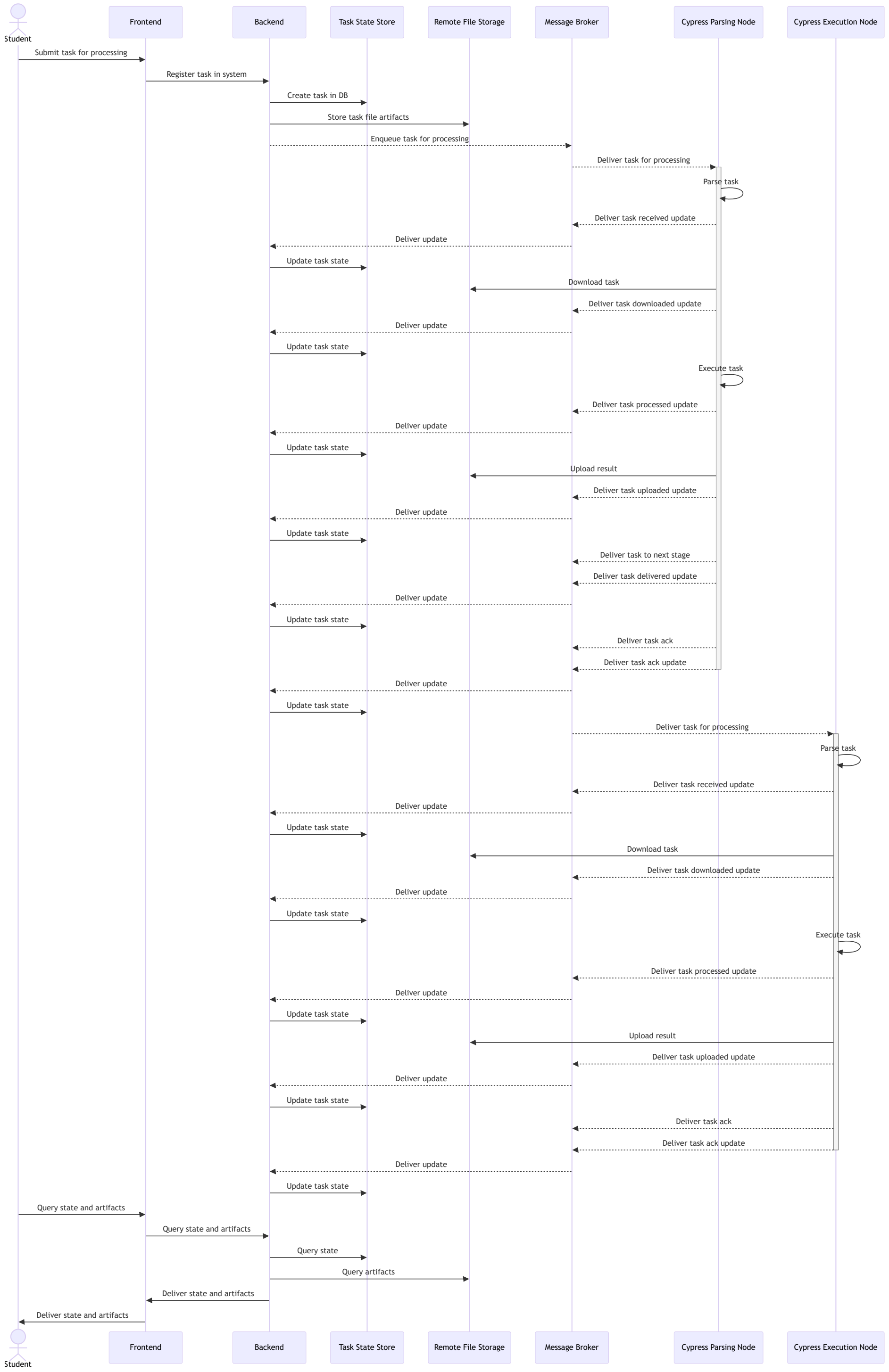


Figure 6.2: Complete Cypress workflow

6.4 Technology-Related Implementation Adaptations

During the implementation of the prototype system, certain design elements were adapted to align with the operational characteristics and constraints of the technologies employed, while maintaining full adherence to the overall architectural model.

In the case of message broker communication, the architecture's requirement for isolated and reliable message handling was preserved, but the mechanism was adjusted to comply with the AMQP protocol and RabbitMQ's threading model. The initial plan to have short-lived actors dynamically create channels for message publication and acknowledgment was revised when it became clear that AMQP channels are not thread-safe and that acknowledgments must be issued on the same channel that received the message. Because channels are bound to a specific thread, sharing them across concurrent actors would have introduced instability. To address this, the message broker communicator was implemented as a long-lived dedicated actor maintaining a persistent channel, with all publications and acknowledgments funneled through it. This approach preserves the intended separation of concerns while ensuring correct protocol usage.

Similarly, the MQ Consumer actor initially conceived as an explicit consumption loop was omitted, as RabbitMQ's client library already provides a dedicated thread for managing consumer callbacks. Implementing a separate wrapper actor would have introduced unnecessary complexity without improving reliability. Instead, message ingestion was integrated directly with the client's callback mechanism, which passes received messages into the actor system for further processing.

For the node's internal system monitor, the architecture called for an actor capable of influencing task admission based on real-time resource usage. While this remains architecturally feasible, the complexity of implementing an effective and reliable feedback mechanism exceeded the scope of the prototype. Consequently, the monitor was implemented in a passive role, collecting and reporting CPU and memory metrics without altering execution concurrency. In this version, workload limits are set manually via broker prefetch counts and blocking thread configurations. The monitoring actor remains in place, preserving the extension point for future adaptive load-control features.

These implementation choices reflect practical adaptations to the behavior and constraints of the chosen technologies, rather than deviations from the architecture. The resulting system retains its modular structure, clear separation of responsibilities, and extensibility for future enhancements.

6.5 Deployment Context

The prototype was deployed in a controlled environment designed to emulate a small-scale distributed system. Two machines connected over a local home network formed the hardware base:

- **Machine 1:** Apple MacBook Pro (M1) with 16 GB RAM.
- **Machine 2:** Tower workstation with AMD Ryzen 7 8700, 32 GB RAM, and GPU RX 7600.

The message broker (RabbitMQ) and remote file storage service (FTP-based) were hosted together on a single machine, while the processing nodes were distributed across both machines. This arrangement allowed for realistic testing of inter-node communication, message passing, and workload distribution under network conditions representative of a campus or lab-scale deployment.

This deployment architecture reflects the intended operational environment of the system in an academic setting. By distributing processing nodes across multiple machines and centralizing coordination and storage services, the prototype replicates the conditions of a real-world laboratory or departmental infrastructure without the cost and complexity of large-scale production systems. This configuration provided a suitable platform to validate the architecture's ability to orchestrate heterogeneous workloads, manage distributed execution, and maintain predictable performance under realistic constraints.

Chapter 7

Evaluation and Results

This chapter presents an evaluation of the proposed architecture through a series of targeted scenarios designed to validate its core properties: fault tolerance, integrability, and scalability. The focus is not on benchmarking raw performance or throughput, but rather on examining how the system behaves under realistic operational conditions: how reliably it executes tasks, maintains isolation under failure, and supports composable multistage workflows. Each scenario was tested on a functional prototype using synthetic tasks and controlled input to simulate expected system behavior. The analysis highlights both the strengths and limitations of the current implementation in relation to the architectural goals defined in previous chapters.

7.1 Evaluation Goals and Methodology

7.1.1 What is being Evaluated and Why

The evaluation focuses on assessing the system's behavior in relation to the architectural objectives defined in Chapter 5. Specifically, the evaluation targets three main concerns: fault tolerance, understood as the system's ability to continue operating in the presence of internal failures or exceptional conditions; integrability, referring to the capacity of individual components to interact in a modular and decoupled fashion; and scalability, considered here in relative and exploratory terms rather than as a benchmarked performance metric.

These properties were selected based on the system's intended role as a processing backbone for task-based workflows in educational or heterogeneous environments. Since the system is designed to operate under asynchronous, message-driven conditions, correctness is defined in terms of internal execution consistency, fault isolation, and predictable coordination among components, rather than by application-specific outcomes or external outputs. Therefore, the evaluation emphasizes execution flow, actor coordination, and system responsiveness under controlled conditions.

7.1.2 Evaluation Scenarios

The system was evaluated through a series of targeted fault injection and integration scenarios designed to validate core architectural behaviors under normal and exceptional conditions. These scenarios test the robustness of the actor supervision strategy, the reliability of task routing and execution, and the integration with external components such as the message broker, remote file storage, and containerized executors. Each scenario was implemented on a live prototype and observed for correct failure propagation, recovery behavior, and notification dispatch through the message queue. All tests were performed using synthetic task inputs, controlled faults, and pre-configured routing paths.

Scenario 1 **- Remote Storage Access Failure Recovery: Download and Upload**

This scenario evaluates the system's behavior under various failure conditions related to accessing the remote file storage subsystem. Two types of operations were tested independently: downloading task artifacts at the beginning of task execution and uploading result files at the end. In both cases, failures were induced by preventing a successful interaction with the remote FTP server through different methods including the following:

- Incorrect credentials for FTP server
- Unreachable host for FTP server
- Induced connection drop before file download

- Induced connection drop during file download
- Induced connection drop before file upload
- Induced connection drop during file upload

In each case, the failure occurred within a dedicated ephemeral actor spawned by the Remote Storage Manager. When the actor attempted the connection and failed, it crashed as expected. The supervising manager detected the failure and issued a structured failure notification to the message broker, including the relevant task metadata and the specific error encountered. The task was not forwarded to any subsequent processing stage and was rejected from the queue. No retry or internal compensation mechanism was triggered, adhering to the system's design principle of stateless execution and clear fault boundaries.

The Remote Storage Manager remained fully operational, allowing subsequent tasks to be processed without interruption. These tests demonstrate the architecture's capacity to isolate and contain I/O-related faults while maintaining system responsiveness. Successful handling of both upload and download failures reinforces the resilience of the actor supervision hierarchy and confirms the design's suitability for integrating with unreliable or external infrastructure.

Scenario 2 **- Execution Failure Recovery: Crash and Timeout**

These scenarios test the system's behavior when execution is disrupted due to either an unexpected crash or a prolonged runtime that exceeds the defined execution timeout. Execution crashes were induced by explicitly raising unhandled exceptions during task processing, either by the implemented executor or the task sent for processing, representing cases where the executor encounters unexpected conditions or errors that cause it to terminate abnormally. Timeout scenarios, in contrast, allowed the task to continue until the system-enforced time limit was reached, at which point the task was forcefully terminated.

In both cases, execution occurred within an ephemeral actor spawned by the Execution Manager. When the execution actor crashed or was terminated due to a timeout, the supervising manager detected the failure and responded by emitting a failure notification to the message broker. This notification included the task's metadata along with the specific reason, either a crash trace or a timeout message. The task was then rejected from the queue and not forwarded to any subsequent processing stage. As in other fault cases, no internal retries were performed.

These tests confirm that the system correctly isolates execution-level crashes and timeouts without propagating the fault. The actor supervision strategy ensures that only the faulty component is terminated and the node remains fully operational. This validates the architecture's ability to absorb unexpected runtime crashes and enforce execution limits while preserving task boundaries and reporting consistency.

Scenario 3 **- Message Broker Communication Failure Recovery: Connection and Messages**

This scenario evaluates the system's resilience in the presence of message broker connectivity issues, both during initialization and mid-operation. Two distinct cases were tested.

In the first case, the system fails to establish or maintain a connection to the message broker induced through different methods:

- Misconfiguration of broker connection parameters
- Intentional network disconnection
- Intentional unreachability of broker host

In both situations, the Message Broker Manager crashes. The Node Orchestrator, which supervises the manager, attempts to restart it using a backoff strategy. If reconnection attempts repeatedly fail, the orchestrator ultimately shuts down the node to prevent it from operating in a partially functional state. This ensures that no task processing proceeds without guaranteed messaging capabilities.

In the second case, the broker remains reachable and the connection and channel are technically active, but outgoing messages intermittently fail to transmit, such as due to protocol-level errors or broker-side constraints. For example, this was induced by setting a maximum number of messages that could be present in a queue and so new incoming messages are automatically rejected. In these cases, failures are logged but do not immediately terminate the manager. However, to guard against silent or undetected degradation, the system monitors the number of consecutive failed message attempts. If a threshold is exceeded, the Message Broker Manager crashes voluntarily, prompting the orchestrator to restart it with a new connection and channel.

These tests confirm that the system actively supervises its messaging infrastructure and avoids silently continuing in degraded states. It ensures that nodes are either fully connected or fully deactivated, maintaining architectural integrity and preventing loss of delivery guarantees. This behavior upholds the system's fault tolerance and reinforces the reliability of its external communication boundary.

Scenario 4 **- Dynamic Task Routing Based on Execution Outcome**

This scenario evaluates the system's ability to route tasks dynamically based on execution results, using routing metadata embedded within each task. Upon completion of a task, the system examines whether the execution was successful or failed in a controlled manner. Based on this outcome, the system selects the appropriate next queue, which is pre-defined in the task's metadata. Routing decisions are never generated by the node itself; instead, the system reads and applies the instructions specified when the task was initially submitted.

The tests were conducted across multistage workflows, where tasks were routed through up to five consecutive processing stages. Each stage ended with a conditional routing decision, directing the task to a different queue depending on whether the outcome was classified as pass or fail. This mechanism was verified both for successful executions and for cases where the executor determined that the task should not proceed further, even though no crash occurred. In all cases, the system correctly routed the task to the designated queue and continued processing without error.

This behavior confirms that the system fully supports dynamic, metadata-driven routing across multiple stages. It also demonstrates that execution logic remains decoupled from routing logic, as required by the architecture. By delegating routing decisions to task-level metadata and not allowing nodes to alter routing information at runtime, the system maintains a clean separation of concerns and reduces the complexity of processing nodes. These results validate the system's ability to orchestrate flexible, outcome-dependent workflows without requiring centralized coordination or embedded routing logic.

Scenario 5 - Malformed Task Message Recovery: Deserialization

This scenario tests the system's ability to reject invalid tasks that cannot be interpreted due to incorrect or malformed structure. A message was deliberately submitted to the system in a format that violated the expected schema and could not be deserialized into a valid task object. Because task decoding occurs immediately upon message reception, the failure was detected before any further processing began.

Upon encountering the malformed message, the system rejected the task without attempting to notify the backend. Since the message structure itself was invalid, the system could not extract the necessary metadata required to construct a failure notification. Instead, the task was rejected at the queue level and discarded or requeued according to the message broker configuration. No actor within the system was created to handle the task, and no crash occurred, as deserialization failures are treated as non-recoverable validation errors.

This test confirms the system's defensive behavior against malformed inputs and demonstrates that invalid messages are filtered out early in the processing pipeline. It also highlights the architectural decision not to attempt recovery or compensation for tasks that lack structural integrity. The system maintains robustness by ensuring that only well-formed and decodable messages are admitted into the actor system, preserving internal stability and reducing the risk of undefined behavior from malformed external input.

Scenario 6 - Integration with External Executors

This scenario confirms the system's ability to integrate with containerized execution environments. Three types of external executors were used: a Cypress-based grammar checker, a Cypress-based task runner, and a G-code execution engine. Each executor was packaged as a Docker image that complied with the system's expectations for input and output behavior. As mentioned in the previous chapter, these images were not developed as part of this thesis, but were integrated into the processing pipeline to validate compatibility and demonstrate the architecture's support for containerized workloads.

Integration required adapting each executor's invocation, file-handling logic, and result interpretation of each executor to the system's task model. Each image was launched as an isolated execution step within a disposable actor, and the result was interpreted to determine whether the task should be considered successful or not. The execution result was then used to drive dynamic routing decisions based on the task's metadata.

The integration process did not require architectural changes and was completed over approximately 30 man-hours. This includes adapting the task processing logic, connecting to the appropriate image interfaces, and validating the correct routing and output handling for all three cases. These results support the claim that the system's processing model can accommodate external logic in a modular and scalable way, particularly when the integration contract is well defined.

Scenario 7 **- Per-Node Scalability and Throughput**

To explore the basic throughput characteristics of the system, a controlled experiment was conducted using a single node and a high volume of synthetic tasks. The goal was not to evaluate end-to-end system scalability, but rather to observe the internal throughput limitations of a single processing node when all external execution steps were replaced with simulated logic. By isolating the system from task-specific computational workloads, the test focused solely on core architectural behavior such as message ingestion, actor spawning, supervision, and task routing.

Tests were performed under varying message prefetch settings, using between 1 and 32 concurrent task slots. Across all cases, the system maintained high throughput and was able to scale up to 1000 tasks in under one minute, with error rates dependent on prefetch configuration and dispatcher saturation. Although these results confirm that the system can ingest and manage tasks efficiently under load, they do not reflect real-world performance, since actual execution workloads were omitted. Therefore, a detailed explanation of this experiment's parameters and configurations is omitted as it would not be very significant for real workloads.

In practice, the scalability of this architecture depends on multiple deployment factors. Each node is stateless, allowing independent vertical scaling and horizontal scaling by adding nodes. However, meaningful scalability is tightly coupled to the nature of the workloads being processed. Different task types may demand different CPU, memory, disk, or I/O profiles, and as such, throughput and system responsiveness will vary accordingly.

Furthermore, several runtime parameters can be fine-tuned based on workload characteristics. These include the message broker's prefetch count, which affects how many tasks a node attempts to process in parallel; the number of blocking threads configured for execution (which run outside of the actor system); and timeout thresholds for task execution, which can be set per task type to reflect expected runtimes or service-level constraints. The optimal configuration for these parameters depends on the complexity, duration, and resource footprint of each task class.

Although the test provides useful insight into internal processing overhead and concurrency handling, it does not represent real-world scalability. Comprehensive evaluation would require profiling actual workloads, measuring executor performance under realistic conditions, and validating system behavior under mixed and concurrent task streams. Nevertheless, because each node operates independently and does not maintain global state, vertical and horizontal scalability remains fully achievable. Expanding system capacity is a matter of allocating additional resources and configuring them appropriately, making the architecture well-suited for elastic deployment in diverse environments.

7.2 Reflections on Design and Implementation Choices

Throughout the implementation of the prototype system, several lessons emerged regarding the alignment between architectural intentions and the realities of system-level behavior in actor-based runtimes. These reflections concern both the specific adjustments made during development and broader considerations about the choice of implementation technology.

One key realization involved the limitations of threading models in the JVM-based actor system used in this prototype. Akka actors share a thread pool by default, which introduces a risk of blocking operations that consume CPU time meant for other actors. If not properly managed, this can lead to starvation and degraded responsiveness across the actor hierarchy. To mitigate this, the implementation required the use of dedicated blocking dispatchers to isolate long-running tasks such as file I/O or external command execution. Although effective, this introduces configuration overhead and complicates tuning for heterogeneous workloads.

In contrast, actor frameworks based on the Erlang VM, such as Elixir with the OTP (Open Telecom Platform), offer a preemptive concurrency model. Each process is lightweight and isolated, and the scheduler explicitly manages when a process must yield. This model eliminates the need for manual dispatcher separation and allows long-running operations to be safely scheduled without jeopardizing system responsiveness. From a concurrency perspective, such a model would eliminate many of the tuning challenges encountered in this implementation.

However, this advantage comes with trade-offs. Elixir, while architecturally well-suited to the actor model, is part of a narrower ecosystem. Integration with mainstream technologies, including JVM-based tooling, libraries, or protocols such as AMQP, may be more difficult. Fewer libraries and production-ready components can hinder rapid prototyping and increase the cost of integration. Moreover, adopting Elixir would require a rethinking of the structure of some subsystems. For example, responsibilities currently managed by single actors, such as the Remote Storage Manager, which handles file upload, download, and error recovery, would likely be decomposed into multiple more specialized actors, each with a narrower responsibility aligned with OTP design principles.

Although Elixir was not selected for this prototype, its concurrency model offers significant architectural appeal and may warrant further exploration. Future work could investigate whether porting parts of the system to the OTP environment would reduce implementation complexity, improve runtime predictability, or enable better scalability under real-world workloads. For the current scope, the use of Akka/Pekko and the JVM remains justified due to ecosystem maturity, library availability, and interoperability.

7.3 Discussion of Results

The previous sections presented the system's behavior across a variety of controlled scenarios designed to validate its architectural principles, execution model, and operational robustness. This section interprets those findings to assess the system's broader qualities, including fault tolerance, scalability, and integration potential. Rather than focusing on individual test outcomes, the discussion emphasizes emergent behavior, systemic trade-offs, and the architectural implications observed throughout the evaluation process.

7.3.1 System Behavior and Limitations

The system exhibited stable and predictable behavior across all tested scenarios, reflecting the outcome of an iterative design and implementation process. Many of the failure-handling patterns and architectural choices emerged not as static decisions but as refinements made in response to earlier failures and edge cases. Faults encountered during early development cycles led to targeted refactorings, ultimately resulting in a robust actor hierarchy, clean separation of concerns, and a clear delegation model within each processing node.

Fault isolation was one of the system's most successful attributes. Each task was processed independently, and task-related failures were fully contained within their own ephemeral actors. Supervising components, such as managers responsible for file handling, message communication, and task execution, were able to manage multiple concurrent tasks without interference. The use of supervision strategies eliminated the need for pervasive error-handling logic within business-level actors and proved effective in maintaining node stability. At no point did a task failure compromise the processing of unrelated tasks.

While task execution and state progression followed expected behavior, the system makes no guarantees of atomicity across a task's lifecycle. For example, it is possible for a result file to be successfully uploaded even if the final task notification fails to be delivered. Acknowledgments and message rejections are reliably handled by RabbitMQ, which is configured to use durable queues and persistent messages. However, notification messages sent to the broker as part of failure reporting or progression updates are subject to loss if the broker is unavailable at the time of transmission. This limitation reflects the system's asynchronous design and its delegation of durability and ordering to the messaging infrastructure.

Scalability is theoretically strong, particularly at the horizontal level. Since processing nodes are stateless and independent, the system can be scaled by deploying additional nodes without modification. RabbitMQ supports high message throughput and can be distributed if necessary. However, scalability is ultimately bounded by available infrastructure and the characteristics of the workloads being executed. The current system does not implement fine-grained task retry mechanisms or task-level idempotency, which introduces the small risk of a task being processed multiple times under rare failure conditions. These design omissions were intentional to keep the system manageable and observable, and they could be addressed in future work.

Finally, the system diverged from its original design in several non-trivial ways. The Message Broker Communicator was refactored into a permanent actor after it became clear that RabbitMQ channels are not thread-safe and must be reused for both receiving and acknowledging messages. Additionally, the dedicated consumer actor was removed because the RabbitMQ client manages its own thread for consumption internally, making a wrapper actor redundant. Backpressure management also proved more nuanced than expected and is currently delegated to operator-level configuration through prefetch counts and dispatcher tuning.

Together, these behavioral traits and limitations define a system that is architecturally sound and operationally resilient within its intended scope, while also clearly delineating areas for future robustness and scaling improvements.

7.3.2 Observation on Scalability and Performance

While the system was not designed or implemented to achieve high-performance throughput in its current form, certain observations about scalability and performance emerged through controlled experimentation and architectural analysis. These findings are not intended to serve as exhaustive benchmarks, but rather to highlight the system's behavior under synthetic load and its readiness for horizontal scaling.

To evaluate internal throughput, a synthetic task execution scenario was used in which external processing steps were bypassed. This allowed for a clearer measurement of the system's baseline performance, isolating factors such as actor spawning, supervision, message acknowledgment, and file operation orchestration. The system was able to process thousands of tasks per node in under one minute, and prefetch count configurations were shown to affect parallelism and task ingestion rates. However, increasing prefetch beyond a certain point offered no substantial gains, likely due to the saturation of the configured thread pool and diminishing returns under the simulated workload.

It is important to note that these tests do not represent real-world system performance under heterogeneous, resource-intensive workloads. In the actual system, execution behavior depends heavily on the task type, external resource utilization, and duration of blocking operations. As such, scalability must be evaluated in context. Real scalability depends not only on node throughput but also on how different task types interact with compute, memory, disk I/O, and external infrastructure.

Despite this, the architecture supports horizontal scalability by design. Because processing nodes are stateless and operate independently, new nodes can be deployed in parallel to handle increased throughput, without architectural changes. RabbitMQ, the message broker used in the prototype, is capable of handling high message volumes and can be distributed if needed. These traits make the system well-suited for scaling in response to demand, provided that node resources and routing logic are tuned appropriately.

Several runtime parameters can be fine-tuned based on workload profiles. These include the message broker's prefetch count, the number of concurrent blocking threads allocated to task execution, and per-task timeout thresholds. None of these tuning strategies are automated in the current system, but they remain accessible to an operator seeking to optimize resource usage for specific deployments.

In summary, while the prototype does not provide performance guarantees or implement dynamic scaling mechanisms, it demonstrates architectural readiness for scalable deployment. Future evaluations should focus on benchmarking the system under real workloads and developing workload-aware tuning strategies.

7.3.3 Observations on Fault Tolerance

Fault tolerance is one of the system's central design goals, and the evaluation scenarios confirm that the system is capable of isolating, recovering from, and reporting faults without compromising the integrity or responsiveness of the processing node. These behaviors emerged as a result of both the actor-based supervision model and the architectural decision to encapsulate risk-prone operations in ephemeral components.

Each task is handled in isolation by a short-lived actor that performs a specific operation such as downloading files, executing a command, or uploading results. If a failure occurs during one of these operations, the responsible actor crashes, and its supervising component captures the failure and responds accordingly. This design ensures that task-specific faults do not propagate across the system or affect other tasks being processed concurrently. Managers responsible for supervision remain long-lived and stateless, delegating all execution responsibility to disposable children and avoiding the accumulation of unrecoverable internal state.

The supervision hierarchy proved highly effective in enabling recoverability without entangling business logic with defensive programming. Instead of embedding error-handling logic directly in each operation, the system relies on supervision strategies to restart, ignore, or terminate failing actors depending on the failure context. This significantly reduced complexity and improved maintainability. Failures during task execution, file operations, or broker communication were all handled without triggering wider instability.

That said, fault handling is not fully comprehensive. The system assumes that certain failures such as the loss of notification messages to the message broker—are tolerable, given the asynchronous and best-effort nature of external communication. Similarly, there is no guarantee of atomicity across all steps of task execution. A task may partially complete and leave side effects (such as uploaded results) even if later stages fail. These limitations are inherent to the system's stateless, decoupled design, and while they do not undermine the overall robustness of the node, they highlight the trade-off between flexibility and strict consistency.

Overall, the system exhibits clear boundaries for fault containment, well-scoped failure propagation, and effective recovery through supervision. These properties, while not eliminating all forms of failure, ensure that the system remains predictable, maintainable, and stable in the face of common faults.

7.3.4 Suitability for Integration with External Infrastructure

The architecture was designed to accommodate heterogeneous execution environments, and evaluation results confirm that the system is suitable for integration with external infrastructure. Processing nodes treat task execution as a black-box operation: the system handles orchestration, file delivery, and task lifecycle tracking, while the execution step itself can delegate to any external tool or environment that conforms to the expected input-output contract.

To validate this integration capability, the system was tested with three Dockerized execution environments: a Cypress-based grammar checker, a Cypress-based test executor, and a G-code execution tool. Each task type was routed through the system, executed in isolation within its respective container, and correctly reported a pass or fail outcome. These integrations did not require modification to the system's architectural core, only adjustments to the logic responsible for invoking external executors. The total integration effort across all three task types was approximately 30 man-hours, demonstrating that the system can be extended in a modular and tractable way.

The use of Docker containers aligned well with the system's actor-based execution model. Each task was executed by a disposable actor, which launched a single container, monitored its result, and shut down. This pattern supports strong execution isolation, reproducibility, and alignment with the stateless, fault-tolerant nature of the architecture.

Although physical infrastructure integration was not evaluated directly, the mechanisms for launching external processes and reacting to their output are applicable to hardware control, device orchestration, or external system invocation. The system does not constrain the nature of the execution environment, provided that appropriate input artifacts are available and a result can be reported back. However, it is important to note that integration with physical systems often requires careful programming, validation, and safety checks to avoid unintended effects, such as leaving devices in inconsistent states or damaging sensitive equipment. These workloads may involve control protocols, low-level device interfaces, or time-sensitive operations, all of which introduce additional engineering challenges.

As such, while the system can support physical and external workloads, the effort required to integrate them will depend on the complexity and risk profile of the infrastructure involved. Workloads that involve physical actuation, high concurrency, or specialized protocols may require significant engineering effort and validation. Nevertheless, the evaluation confirms that the architectural foundation supports this type of integration, and that the extension path, while workload-dependent, is clearly defined and feasible.

These results support the claim that the proposed architecture is suitable for deployment in hybrid environments that combine virtualized execution with physical task execution stages. The integration model has been shown to be robust, flexible, and compatible with real-world tooling.

Chapter 8

Conclusions and Future Work

8.1 Summary of Contribution

This thesis presents the design, implementation, and evaluation of a distributed task processing system tailored to orchestrate multistage execution workflows in environments where virtual and physical infrastructure may be involved. The system was designed with an emphasis on modularity, fault tolerance, and integration readiness and provides a robust foundation for scalable task execution in educational and other workflow-driven contexts.

The primary contribution lies in the architectural design of a message-driven and actor-based processing model that ensures fault isolation, execution flow traceability, and system extensibility. The system leverages RabbitMQ as a message broker and builds upon the actor model using the Pekko toolkit, enabling the decomposition of responsibilities into independently supervised and recoverable components. A detailed fault model was proposed and implemented through a supervision hierarchy, allowing a clear separation between the control, execution, and failure recovery logic within each processing node.

A secondary but significant contribution is the validation of the architecture through the integration of multiple task types, implemented as Docker containers, and the demonstration of end-to-end task routing, execution, and result notification under both nominal and failure conditions. This validation confirms the system's ability to support heterogeneous workloads and lays the groundwork for future integration with physical infrastructure and real-world deployment environments.

The thesis also explores critical implementation trade-offs, including messaging guarantees, supervision granularity, and dispatcher configuration, while proposing a design that remains agnostic to specific execution environments. Together, these contributions establish a functional, extensible, and resilient execution layer that can serve as the computational backbone for more complex orchestration workflows involving user interfaces, data storage, and domain-specific integration.

8.2 Lessons Learned

The development of this system provided several important lessons regarding the design and implementation of fault-tolerant and message-driven architectures, particularly in the context of asynchronous and loosely coupled task workflows.

One of the most critical insights was the need to better understand the operational constraints of third-party systems, particularly the message broker. Early designs assumed that RabbitMQ channels could be freely created and shared across threads, which proved to be incorrect. The requirement that acknowledgments must occur on the same channel that received the message required a refactoring of the message communication logic and reinforced the importance of aligning the system architecture with the semantics and limitations of its external dependencies.

Another key lesson was the power of actor supervision hierarchies in simplifying error management. Rather than defensively guarding each operation with extensive error-handling code, failures were naturally isolated and propagated through well-scoped supervision trees. This not only reduced complexity, but also enforced discipline around responsibility boundaries and recovery semantics. The actor model proved particularly well suited for decomposing complex node behavior into manageable, isolated components.

At the same time, the actor model introduced challenges of its own. Coordinating blocking operations, such as external command execution, required careful tuning of dispatchers to avoid starvation of system threads and preserve responsiveness. This experience prompted a re-evaluation of the actor model not as a universal behavior, but as an abstraction whose guarantees and constraints depend heavily on its implementation environment. Prior exposure to the BEAM virtual machine and Elixir's native actor system had shaped an understanding of actors as lightweight, isolated, and preemptively scheduled processes. However, implementing actors on the JVM revealed a different reality: the JVM uses cooperative scheduling, where threads voluntarily yield control, making it susceptible to thread starvation if blocking operations are not explicitly delegated. In contrast, BEAM enforces preemptive scheduling, where the runtime forcibly interrupts processes after a set number of reductions, ensuring fair scheduling and isolation without additional configuration. This contrast revealed that, while the actor model provides useful conceptual structure, its operational behavior must be carefully reconsidered in the context of the underlying runtime. Within the JVM, this made it essential to introduce dedicated dispatchers for blocking operations to preserve the responsiveness of supervisory and coordination actors.

Finally, the iterative nature of the development process reinforced the value of prototyping and early testing. Many design choices, such as the elimination of a dedicated MQ consumer actor or the simplification of the System Monitor, emerged from practical experimentation and failure, not from initial planning. As a result, the final system reflects a more grounded and operationally informed architecture than would have been possible by top-down design alone.

These lessons collectively emphasize the need for tight feedback loops between architectural intent and implementation reality, especially in systems where concurrency, fault recovery, and external integration intersect.

8.3 Limitations

Although the system successfully demonstrates a resilient and modular foundation for distributed task processing, several limitations remain, both by design and as a result of implementation constraints.

First, the system does not guarantee atomicity or complete end-to-end consistency for task execution. Tasks are processed as isolated units by disposable actors, and intermediate steps such as file uploads or result generation may succeed even if subsequent stages fail. Because no transactional boundary spans all stages of task handling, partial execution is possible and side effects may persist in external systems despite task failure. This trade-off is inherent to the stateless, loosely coupled design and was accepted to preserve scalability and fault containment.

Second, status updates and task progression notifications are delivered via a best-effort model. Although message delivery to the broker is reliable under normal conditions, due to persistent queues and acknowledgment mechanisms, there is no guarantee that all updates reach the backend, particularly in the event of transient broker failures. Furthermore, once an update is acknowledged by the broker, the system assumes that it has been successfully received, even though the backend may be temporarily unavailable. This introduces a risk of incomplete visibility into task history unless additional delivery guarantees or state reconciliation mechanisms are introduced.

Third, while the actor model provides strong isolation and supervision boundaries, the system does not implement true task-level idempotency. A task may be reprocessed more than once under rare conditions, such as consumer failure after partial execution but before acknowledgment. This limitation reflects the absence of deduplication logic or persistent task state tracking at the processing node level. For tasks with side effects, this may require additional safeguards in downstream systems to ensure correctness.

Fourth, backpressure control and concurrency tuning are not fully dynamic. Each node must be manually configured with a maximum number of concurrent blocking threads and a message prefetch count. Although a monitoring actor is present in the architecture, it does not currently influence task admission or scaling behavior. As such, the nodes cannot adapt in real time to workload surges or changing resource availability, and relies on operator judgment for capacity tuning and manual deployment of new nodes.

Finally, this thesis focused exclusively on the core execution infrastructure. Important components, such as the backend service, database integration, user interface, and persistent task history, were explicitly excluded from the scope. As a result, while the execution layer is functional and extensible, full system deployment in an educational or production context would require the addition of several supporting modules to handle coordination, submission management, and user interaction.

These limitations do not undermine the validity of the system's core architecture, but rather reflect deliberate scope boundaries and implementation choices made to prioritize fault isolation, modularity, and feasibility.

8.4 Future Work

The system presented in this thesis establishes a solid architectural and operational foundation for distributed task orchestration, but it represents only a portion of the broader infrastructure needed for full deployment in real-world scenarios. Several avenues for future development have been identified, some involving natural extensions of the current design, and others requiring deeper architectural enhancements or integration with external systems.

Future work can be categorized along five main axes: extending the system to interact with physical infrastructure, improving executor runtime behavior, enhancing backpressure management, enabling dynamic and containerized deployment strategies, and completing the orchestration stack through the development of user-facing and coordination components. Each of these directions builds on the current architecture while addressing limitations or operational challenges encountered during prototyping.

8.4.1 Integration with Physical Infrastructure

An important direction for future work involves the extension of the system to support workflows that interact with physical infrastructure, such as laboratory equipment, robotic platforms, or embedded control systems. Although the current prototype was validated using containerized workloads, the architecture is designed to remain agnostic to the environment in which tasks are executed. This makes it suitable, in principle, for integration with physical systems without requiring changes to the orchestration layer.

In such cases, infrastructure-specific behavior and communication must be encapsulated entirely within the executor logic. The system assumes that each executor adheres to a defined interface. As long as these conditions are met, task execution remains compatible with the orchestration flow.

The responsibility for safe and consistent interaction with external systems should lie entirely on the executor's implementation. Tasks involving hardware devices may require custom communication protocols, error-handling logic, or safety mechanisms to prevent equipment damage or inconsistent physical states. These requirements must be addressed within the executor and are outside the concern of the orchestration platform.

This separation of concerns helps preserve the extensibility of the architecture. Integration with physical infrastructure, regardless of its complexity, is achievable through the development of specialized executor modules. Future work in this direction may involve designing executor templates, incorporating safety wrappers, or developing validation frameworks for hardware-bound execution, all without altering the orchestration logic or compromising the modularity of the system.

8.4.2 Executor Runtime Tuning and Dispatcher Configuration

The current system uses a dedicated blocking dispatcher to execute tasks that involve long-running or potentially blocking operations. While this separation protects coordination actors from thread starvation, the dispatcher itself is configured with only basic parameters intended for prototype validation. These include static values for thread pool size and task queue limits, which may not generalize well to more demanding or heterogeneous workloads.

Future work should focus on fine-tuning this dispatcher configuration to better align with the execution characteristics of specific tasks and the JVM's threading model. This requires a deeper analysis of how the virtual machine schedules threads, how blocking operations compete for system resources, and how different dispatcher settings affect throughput and latency under load.

In addition, tuning must consider the machine-level context in which nodes are deployed. The number of concurrent tasks a node can handle is ultimately constrained by available CPU cores, memory, and I/O bandwidth. A one-size-fits-all configuration may lead to underutilization on capable machines or overload on constrained ones. For this reason, it is worth exploring approaches in which nodes can adjust their own execution parameters based on available system resources, real-time usage metrics, or operator-defined policies.

Although the current implementation provides the necessary infrastructure for this separation of execution contexts, further work is needed to ensure that dispatcher behavior can be adapted, optimized, and monitored in ways that improve system responsiveness and resource efficiency.

8.4.3 Backpressure Management and Dynamic Load Control

Although the system successfully handles concurrent task execution and isolates failures at the actor level, it currently does not implement an internal mechanism to regulate the rate at which tasks are ingested and processed by each node. Backpressure management is critical to prevent overload, especially in distributed systems where task submission can exceed the processing capacity of individual nodes.

In the current implementation, backpressure is managed indirectly through RabbitMQ's prefetch count, which defines how many unacknowledged messages can be delivered to a node at once. Although this provides basic control over task intake, the prefetch count is fixed at startup and must be manually tuned by the operator. Furthermore, it does not adapt to dynamic fluctuations in system load, such as CPU saturation, memory pressure, or I/O congestion.

Future work should explore more dynamic load control strategies. One direction involves integrating the System Monitor actor, which currently tracks CPU and memory usage, into the decision-making process to adjust prefetch behavior or temporarily pausing message consumption. Although RabbitMQ does not allow changes of the prefetch count after a consumer has started, alternative strategies may involve periodically closing and reopening the consumer, or adopting a pull-based model where tasks are explicitly requested rather than pushed.

More generally, the system could benefit from the development of adaptive policies that consider both static machine characteristics (e.g., core count, dispatcher capacity) and runtime conditions (e.g., task execution time, failure rate, system pressure) to automatically regulate the number of concurrent tasks being processed. This would reduce the need for manual tuning and allow each node to operate closer to its optimal capacity.

Improving load control mechanisms is essential to ensure stability under high-throughput conditions and maintain predictable performance as task types and workloads evolve.

8.4.4 Automated Deployment and Container Runtime Strategy

Currently, the processing nodes must be manually deployed and configured, which limits the scalability and responsiveness of the system. This static deployment model does not align with the system's architectural goal of supporting horizontally scalable execution on distributed resources. To address this, future work should focus on enabling automated node provisioning using container orchestration platforms such as Kubernetes.

Containerization already provides a clean encapsulation of node behavior, but the current implementation relies on Docker and assumes a manually managed Docker daemon. This introduces limitations, especially when deploying at scale or in environments where Docker-in-Docker setups are problematic. A more robust approach would involve migrating to a container runtime that supports rootless execution or is more compatible with orchestrated deployments, such as **containerd** or **Podman**, and configuring the system to be agnostic to the underlying runtime.

Automated deployment would allow the system to elastically allocate processing nodes based on workload demand, resource availability, or external scheduling policies. In such a model, the nodes could be launched as ephemeral containers with environment-specific configuration injected at runtime. This would make the system more responsive to spikes in task volume and reduce the operational burden of managing node lifecycles.

This direction also opens the door to defining deployment profiles for different classes of nodes, such as those optimized for CPU-heavy execution, low-latency I/O, or integration with external infrastructure. By containerizing nodes in a consistent way and deploying them through orchestrators, the platform could evolve into a fully self-managed distributed processing layer.

The architecture already supports this mode of deployment, but realizing it in practice will require integration with container orchestration tools, runtime configuration management, and more flexible support for resource discovery and node registration.

8.4.5 Completion of Orchestration Stack and User-Facing Interfaces

The focus of this thesis has been on the design and implementation of the distributed execution layer. Although this component provides the foundation for orchestrating complex multistage task workflows, a complete system capable of serving end users, such as students or instructors in an educational context, requires several supporting layers that remain outside the scope of the current work.

Most importantly, the system lacks a centralized backend service to coordinate submissions, store task metadata, and expose APIs to querty task status and results. This component would act as the central point of interaction between the user-facing interface and the distributed execution layer, bridging task submission with monitoring, progress tracking, and result retrieval. Although the message broker facilitates asynchronous communication between components, it does not by itself maintain historical state or enforce consistency across task sessions. A persistent coordination layer would fill this role.

In parallel, a user interface, such as a web-based dashboard or course management integration, is needed to allow users to submit tasks, view execution progress, and access results. Although the architecture does not constrain the design of this interface, it must integrate with the backend and conform to the asynchronous, event-driven nature of the system. This includes handling delayed task execution, intermittent updates, and result availability that may not be immediate.

These components are critical to deliver a usable end-to-end platform. Their design and implementation are left for future work, but the architecture developed in this thesis provides the execution guarantees and extension points required to support them.

8.5 Conclusion

The work presented in this thesis lays the foundations for a distributed, modular, and fault-tolerant system capable of orchestrating multistage task execution across heterogeneous environments. By focusing on architectural separation, actor-based fault isolation, and message-driven coordination, the system achieves a high degree of extensibility and resilience, essential properties for real-world applications that demand flexibility and robustness. Although several implementation trade-offs were made to preserve focus and feasibility, the core execution layer is fully operational and extensible. Future work can build upon this foundation to extend the system's capabilities, whether by integrating physical infrastructure, optimizing execution runtimes, improving load regulation, automating deployment, or completing the orchestration stack with backend and user-facing components. These next steps do not undermine the current contribution; they reinforce the architectural vision of a system that is open-ended by design and adaptable to the evolving requirements of its deployment contexts.

Bibliography

Alario-Hoyos, C., Muñoz-Cristóbal, J., Prieto, L., Bote-Lorenzo, M., Asensio-Pérez, J., Gómez-Sánchez, E., Vega-gorgojo, G., & Dimitriadis, Y. (2012). Glue! - glue!-ps: An approach to deploy non-trivial collaborative learning situations that require the integration of external tools in vles. *1st Moodle Research Conference*.

Apache Software Foundation. (2025). *Apache airflow* [Accessed: 2025-04-15]. https://airflow.apache.org/

Arévalo, G. A. S. (2013). *UnaCloud Suite – Solución Integral para Computación de Alto Rendimiento sobre Cloud Computing Oportunista* [Trabajo de grado de maestría]. Universidad de los Andes. Retrieved April 15, 2025, from http://hdl.handle.net/1992/12164

Argo Project. (2025). *Argo workflows documentation* [Accessed: 2025-04-15]. https://argoproj.github.io/workflows/

Castro Iregui, M., Parra Martínez, D., & Escobar Rivera, J. (2025). *Uniandes lab orchestration system (ulos): Ui, api and execution nodes* (tech. rep.). Universidad de los Andes. https://hdl.handle.net/1992/76498

Celery Project. (2025). *Celery 5.3 documentation* [Accessed: 2025-04-15]. https://docs.celeryq.dev/en/stable/index.html

Conductor OSS. (2025). *Conductor - microservices and workflow orchestration platform* [Accessed: 2025-04-15]. https://conductor-oss.org/

González, E. (2024). *Sistema de orquestación para laboratorios remotos uniandes* (Proyecto de pregrado) (Available from The Software Design Lab repository). Universidad de los Andes. https://hdl.handle.net/1992/74499

Lachand, V., Jalal, G., Michel, C., & Tabard, A. (2018). Toccata: An activity centric orchestration system for education. *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, 1–6. https://doi.org/10.1145/3170427.3188484

Lara-Bercial, P. J., Gaya-López, M. C., Martínez-Orozco, J.-M., & Lavado-Anguera, S. (2024). Pbl impact on learning outcomes in computer engineering: A 12-year analysis. *Education Sciences*, *14*(6), 653. https://doi.org/10.3390/educsci14060653

Lavado-Anguera, S., Velasco-Quintana, P.-J., & Terrón-López, M.-J. (2024). Project-based learning (pbl) as an experiential pedagogical methodology in engineering education: A review of the literature. *Education Sciences*, *14*(6), 617. https://doi.org/10.3390/educsci14060617

Mahtani, K., Guerrero, J. M., & Decroix, J. (2024). Implementing innovation in project-based learning in electro-mechanical engineering education. *International Journal of Mechanical Engineering Education*. https://doi.org/10.1177/03064190241284600

Montañez, L. F. P. (2019). *Opportunistic IaaS platform based on containers* (Trabajo de grado). Universidad de los Andes. Retrieved April 15, 2025, from http://hdl.handle.net/1992/45208

Prieto, L., Asensio-Pérez, J., Dimitriadis, Y., Gómez-Sánchez, E., & Muñoz-Cristóbal, J. (2011). GLUE!-PS: A Multi-language Architecture and Data Model to Deploy TEL Designs to Multiple Learning Environments. *Proceedings of the 6th European Conference on Technology Enhanced Learning (EC-TEL 2011)*, 285–298. https://doi.org/10.1007/978-3-642-23985-4_23

Sekler, E. K. (2016). *Contenedores en UnaCloud* (Trabajo de grado). Universidad de los Andes. Retrieved April 15, 2025, from http://hdl.handle.net/1992/14156

Temporal Technologies, Inc. (2025). *Temporal - durable execution for microservices* [Accessed: 2025-04-15]. https://temporal.io/

Urazmetov, A. A. O. (2011). *UnaCloud MSA – Plataforma Basada en UnaCloud para la Generación y Análisis de Alineamientos Múltiples de Secuencias* (Trabajo de grado). Universidad de los Andes. Retrieved April 15, 2025, from http://hdl.handle.net/1992/14748

Worcester Polytechnic Institute. (2024). *Wpi alumni survey confirms value of project-based learning in higher education* [Accessed April 2025]. https://www.wpi.edu/news/wpi-alumni-survey-confirms-value-project-based-learning-higher-education