UNIVERSITEIT VAN AMSTERDAM

Master Thesis

Standardizing Workflow Execution: CWL Integration in Brane for Distributed Scientific Computing

---

Author: Hamza Hadda (2612591)

1st supervisor: Adam Belloum
2nd supervisor: Natallia Kokash

A thesis submitted in fulfilment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

June 29, 2025

VRIJE UNIVERSITEIT VAN AMSTERDAM

Abstract

Msc Computer Science

Standardizing Workflow Execution: CWL Integration in Brane for Distributed Scientific Computing

by Hamza HADDA

Modern scientific and data-intensive applications increasingly rely on workflow management systems to ensure the reproducibility, portability, and scalability of computational pipelines. The Common Workflow Language (CWL) has emerged as a community-driven standard for describing such workflows in a platform-independent and declarative manner. In contrast, the Brane framework offers an imperative, multi-site orchestration system designed for dynamic execution and programmability across distributed environments. This thesis investigates the feasibility and design of integrating CWL into the Brane ecosystem. The goal is to enhance Brane's compatibility with standardized workflow descriptions while preserving its core strengths in flexible orchestration and distributed deployment. The system implemented a translation mechanism to parse CWL CommandLineTool definitions and generate Brane-compatible packages, including support for Docker-based execution and metadata encapsulation. This work contributes a foundation for future interoperability between CWL and distributed workflow engines, opening up new possibilities for reproducible, scalable, and context-aware scientific computing.

Keywords: Workflow orchestration, Common Workflow Language (CWL), Brane framework, distributed computing, containerization, scientific workflows, interoperability

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, **Dr. Adam Belloum**, for his invaluable guidance, constructive feedback, and continuous support throughout this project.

I am especially thankful for the freedom he provided me to explore and design the technical solution independently, while still offering clear direction when needed. His mentorship helped me grow as a researcher and engineer.

I would also like to thank the Brane development team and contributors to the Common Workflow Language (CWL) community for their inspiring work, which laid the foundation for this project.

Finally, I am grateful to my family, friends, and colleagues for their support and encouragement during the course of this Master's program.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

The rapidly increasing complexity of computational applications across various domains has necessitated advanced methods for managing and executing workflows. These workflows, composed of interconnected tasks and processes, are particularly critical in scientific research, data-intensive industries, and high-performance computing. Workflow orchestration—managing and automating the execution of workflows across distributed and heterogeneous environments—has emerged as a cornerstone for achieving scalability, reproducibility, and efficiency. This paper investigates the integration of the Common Workflow Language (CWL) standard into the Brane framework, a programmable orchestration platform, to enhance its compatibility and usability in diverse application domains.

Workflow Orchestration

Workflow orchestration involves coordinating multiple interdependent tasks, ensuring they are executed in the correct sequence, with appropriate resource allocation and data flow. In the context of scientific research and data-driven applications, the demand for scalable and portable workflow solutions has grown significantly. The challenges in orchestrating workflows are multifaceted:

Heterogeneous Infrastructure: Workflows often need to operate across diverse computational resources, including local clusters, cloud platforms, and edge devices. Managing compatibility and performance in such heterogeneous environments is non-trivial.

Reproducibility and Portability: Ensuring that workflows can be replicated

across different systems with identical outcomes is a critical requirement, particularly in scientific domains where reproducibility underpins research validity.

Ease of Use: Many workflow orchestration tools are targeted at domain experts who may lack advanced programming skills, necessitating intuitive interfaces and abstractions. Modern orchestration frameworks address these challenges by leveraging containerization, declarative specifications, and dynamic resource management. However, gaps remain in ensuring interoperability between different workflow standards and tools, limiting their adoption across diverse use cases.

The Brane Framework

The Brane framework is a container-based platform designed to facilitate the orchestration of multi-site applications. It addresses the complexities of distributed workflow execution by providing a programmable environment with a high degree of flexibility and abstraction.
Brane's architecture is highly modular, integrating tools and workflows into a unified runtime system. The use of containerization ensures that workflows and their dependencies are encapsulated, portable, and deployable across different environments. Brane also includes a package registry for managing pre-defined functionalities and provides interfaces for integrating external standards and systems[1].

Despite its strengths, Brane's adoption has been somewhat limited by the need for seamless integration with established workflow standards, such as the Common Workflow Language. Bridging this gap could significantly enhance Brane's versatility, enabling users to import existing workflows, leverage community-supported tools, and expand its applicability across domains.

The Common Workflow Language

The Common Workflow Language (CWL) is an open standard for describing workflows in a platform-independent and reproducible manner. Designed to address the needs of scientific and data-intensive workflows, CWL emphasizes the portability and reusability of workflow definitions. CWL separates

the description of tools and workflows into two key components:

Command-Line Tool Descriptions: Define the inputs, outputs, and command-line operations for individual tasks.

Workflow Descriptions: Specify the sequence and dependencies of tasks, enabling complex multi-step processes to be represented declaratively.

CWL is implemented in a human-readable format, typically using YAML or JSON, and is supported by a broad ecosystem of tools, including CWL runners such as cwltool and Toil. The CWL standard is widely adopted in fields such as genomics, bioinformatics, and data analysis, making it a key enabler for sharing and replicating workflows across diverse environments[2].

1.2 Motivation

The increasing reliance on computational workflows across scientific and industrial domains necessitates platforms that are both versatile and interoperable. While existing workflow orchestration frameworks provide tools to manage complex workflows, they often lack the flexibility to seamlessly integrate with emerging standards and technologies. The Brane framework offers a promising solution by leveraging containerization and programmable orchestration to support multi-site applications. However, its current limitations in supporting widely adopted workflow standards, such as the Common Workflow Language (CWL), restrict its applicability in diverse environments. CWL, as an open standard, is extensively used in fields such as genomics, data science, and high-performance computing due to its emphasis on reproducibility and portability[3].

1.3 Problem statement

The absence of CWL compatibility in Brane restricts its utility for users relying on these standards, creating a barrier to adoption in domains where interoperability is essential. Without this integration, Brane is unable to fully leverage the extensive ecosystem of CWL-compliant workflows and tools, hindering its potential as a versatile and widely applicable orchestration framework. This project aims to address this gap by enabling Brane to parse, execute, and manage CWL workflows, thereby bridging the divide between

Brane's orchestration strengths and the interoperability demands of modern workflow ecosystems.

1.4 Research questions

The following research questions were established in order to address the state-of-the-art of Workflow Languages:

- **RQ1: How can the integration of Common Workflow Language (CWL) into the Brane framework enhance its compatibility and adaptability across diverse computational infrastructures?**
- **RQ2: What are the key challenges and opportunities in aligning Brane's current architecture with emerging standards in workflow and API technologies?**
- **RQ3: What impact does the integration of CWL have on the scalability of Brane for exascale computing workflows?**
- **RQ4: What are the practical implications of integrating CWL specifications into Brane for real-world scientific use cases?**

Research Method

This research follows a design science methodology, which is particularly suitable for addressing technical challenges through iterative development and evaluation of artifacts. The study began with a literature review to understand existing standards and tools in workflow orchestration and identify gaps in interoperability between declarative and imperative systems. Subsequently, the system implements a prototype integration layer between CWL and the Brane framework. We tested this layer using real-world CWL examples to evaluate feasibility, compatibility, and performance. Emphasis was placed on practical experimentation, code evaluation, and usability validation in line with software engineering research best practices.

1.5 Thesis structure

To guide the reader through the complexities of workflow orchestration and the integration of CWL into the Brane framework, this thesis is structured as a gradual exploration, starting from the foundations and progressing toward

concrete implementation and reflection.

We begin in Chapter 1, where the context, motivation, and challenges that prompted this research are introduced. This sets the stage for understanding why interoperability between workflow languages and orchestration platforms is both timely and necessary.

In Chapter 2, we dive into the technical background, unraveling the workings of the Brane framework and the principles behind CWL. This chapter equips the reader with the knowledge needed to grasp the rest of the thesis. Next, Chapter 3 tells the story of the integration itself. It details how we implemented CWL support within Brane—from parsing CWL files to building Docker images and packaging them for Brane execution. This is where the theoretical becomes practical. With the core contribution established, Chapter 4 turns outward, comparing our approach with related systems and prior work in the field. It situates the project within the broader landscape of workflow engines and container orchestration tools. Chapter 5 then outlines the research methods that guided this work. It discusses how the implementation was approached, which criteria were used for evaluation, and what tools and datasets supported the process. Looking ahead, Chapter 6 reflects on the many possibilities for extending this integration—supporting full CWL workflows, handling imports, and improving user experience. It acts as a roadmap for future contributions.

To compile the strategic and technical choices made during the project, Chapter 7 offers a broader discussion of the challenges encountered and the design trade-offs involved in bringing two different paradigms together. Finally, Chapter 8 brings the journey full circle by summarizing the main findings, highlighting the contributions, and reflecting on the implications for scientific computing and beyond.

Chapter 2

Background Methodology

2.1 Workflow orchestration overview

Workflow orchestration is the automated coordination, management, and execution of interdependent tasks within complex processes. This concept is critical in scientific, data-intensive, and high-performance computing applications because it ensures efficient management of resources, scalability, and seamless execution of intricate workflows. For instance, scientific workflows often involve computationally intensive tasks, such as simulations or data analyses, which require precise coordination to process large datasets effectively. Workflow orchestration abstracts the complexity of task dependencies, allowing researchers to focus on problem-solving rather than technical execution details[4]. It facilitates dynamic resource allocation, especially in distributed environments like cloud or high-performance computing systems, optimizing computational and storage resources while reducing overhead. As data volumes and computational demands grow, orchestrated workflows enable scalable solutions, ensuring that large-scale experiments, such as climate modeling or gravitational wave detection, can be executed efficiently[5]. Furthermore, the integration of diverse computational platforms, such as local clusters, HPC systems, and cloud environments, underscores its significance in unifying and streamlining processes across heterogeneous infrastructures. These capabilities make workflow orchestration a cornerstone of modern scientific and computational advancements[6].

2.1.1 Challenges in workflow orchestration

While workflow standards provide robust frameworks for managing computational processes, several challenges persist in their orchestration[7][8][9]:

Heterogeneous Environments: Scientific workflows often operate across heterogeneous platforms, from cloud services to high-performance computing (HPC) clusters. Ensuring compatibility between diverse computational resources remains a significant challenge.

Scalability of Workflows: Scaling workflows to handle increasing data volumes and computational tasks requires efficient resource management. As systems grow, the complexity of coordinating distributed components increases.

Data Dependency Management: Many workflows involve intricate dependencies between data inputs and outputs. Managing these dependencies efficiently is critical to avoid bottlenecks, particularly in domains like bioinformatics and climate modeling.

Error Recovery and Fault Tolerance: Workflows running across distributed systems are prone to failures due to network issues, resource unavailability, or software bugs. Ensuring workflows can recover gracefully and maintain progress is a critical orchestration challenge.

User Accessibility and Complexity: Many workflow systems, despite their powerful capabilities, present steep learning curves for end-users. Bridging the gap between high-level abstractions and low-level execution details remains an area of active research.

Integration with Modern Technologies: Integrating workflows with emerging technologies such as containerization (e.g., Docker, Singularity) and orchestration tools (e.g., Kubernetes) requires constant adaptation to evolving standards.

2.1.2 Key use cases

Workflow orchestration has emerged as a critical enabler of complex computational tasks across diverse domains. Its ability to streamline and manage intricate processes has revolutionized fields ranging from scientific research to industrial applications. Figure 2.1 illustrates the general structure of a scientific workflow system deployed in cloud environments, highlighting the various components that collaborate to support scalable, distributed execution.

In the realm of bioinformatics, researchers are often tasked with analyzing massive genomic datasets to uncover insights about diseases or genetic traits. Imagine a scientist embarking on a project to identify gene mutations linked to a rare disorder. By leveraging workflow orchestration frameworks such as CWL or Pegasus[10], the scientist can define a pipeline that automates each step of the analysis—data preprocessing, sequence alignment, and mutation detection. These tools ensure that dependencies are correctly managed and results remain reproducible, even as datasets and computational environments evolve[11].

Astronomy provides another striking example of the power of workflows. Consider the international team behind the Laser Interferometer Gravitational-Wave Observatory (LIGO), which detects ripples in spacetime caused by massive cosmic events. The workflow begins with raw data collection from sensitive detectors and extends to filtering noise, processing gravitational wave signals, and validating results. Such a pipeline demands immense scalability and precision.

In the world of drug discovery, orchestration frameworks are accelerating the search for life-saving treatments. Picture a pharmaceutical company racing against time to find a compound that could halt a viral outbreak. Workflow systems like Clara guide researchers through the intricate steps of virtual screening, molecular docking, and chemical analysis. These tools not only save time but also ensure that researchers can test thousands of compounds without worrying about computational failures derailing their efforts[12].

Meanwhile, climate scientists face an equally daunting challenge: predicting the future of our planet. In projects like global climate modeling, workflows orchestrate data from satellites, sensors, and simulations, allowing scientists

to piece together a coherent picture of environmental change. These workflows must account for dependencies between datasets while providing robust error recovery mechanisms to ensure that crucial predictions, such as those about hurricanes or rising sea levels, are delivered on time[11].

Even in the fast-paced world of financial risk analysis, workflows play a crucial role. Picture a financial analyst monitoring global markets for potential risks. Behind the scenes, workflows coordinate streams of real-time data, automate calculations, and generate predictive models. Orchestration frameworks ensure that even the slightest anomaly—such as a sudden market dip—is detected and analyzed without delay.

Finally, platforms like Brane epitomize how workflow orchestration empowers multi-site applications. For example, a group of scientists working on a federated healthcare project may need to run complex analytics on patient data while adhering to strict privacy regulations. Brane simplifies this process by encapsulating the technical details and allowing the scientists to focus on their domain-specific tasks. This ensures both compliance and efficiency, enabling meaningful collaboration across institutions.

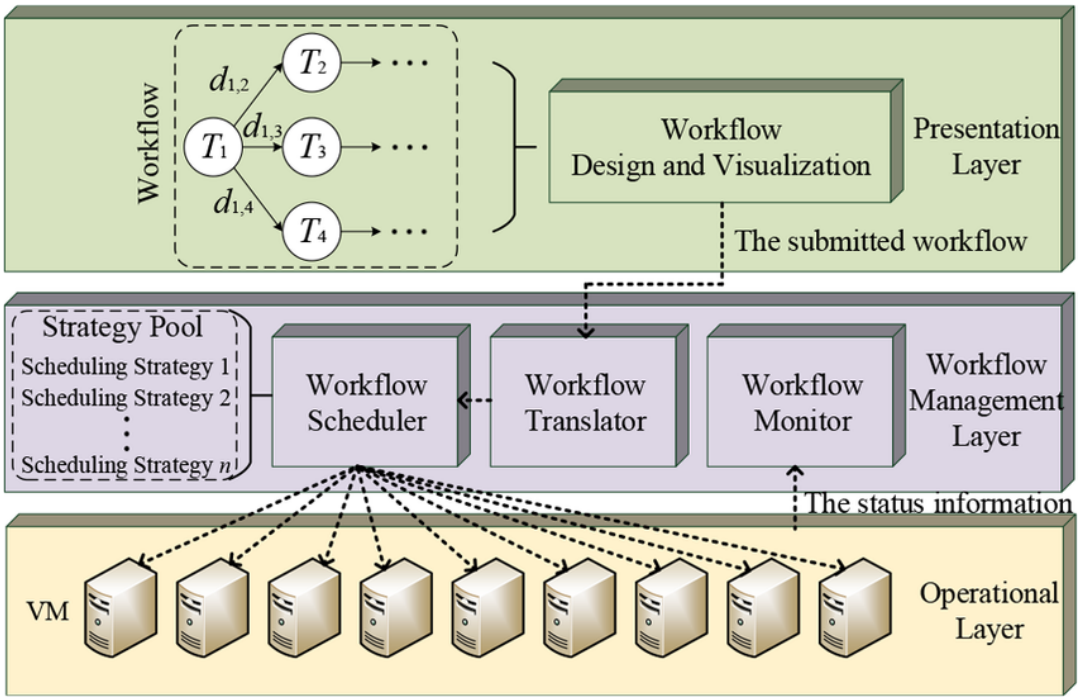

FIGURE 2.1: Structure of a typical scientific workflow system in clouds[13]

2.1.3 Popular Workflow Orchestration Tools

Apache Airflow

Apache Airflow is an open-source platform for programmatically authoring, scheduling, and monitoring workflows. Its Directed Acyclic Graph (DAG)-based[14] approach allows users to define workflows as code, providing flexibility and version control. Airflow's extensible architecture supports custom plugins and integrations with cloud services like AWS, Google Cloud, and Azure, making it a popular choice for data engineering and machine learning pipelines[11].

Kubernetes native tools

Argo Workflows, designed for Kubernetes environments, specializes in container native workflows. It provides high scalability and efficiency, leveraging Kubernetes' inherent features like dynamic resource provisioning. Argo's ability to manage workflows through YAML manifests aligns well with DevOps practices, making it a preferred tool for cloud-native applications and CI/CD pipelines[11].

Nextflow

Nextflow is widely used in bioinformatics and life sciences due to its strong focus on reproducibility and scalability. It supports running workflows across various platforms, including local systems, clusters, and cloud environments. Nextflow's compatibility with Docker and Singularity containers ensures consistent execution, regardless of the environment, and its DSL (domain-specific language) simplifies the definition of complex workflows[15].

Snakemake

Snakemake, another tool popular in the scientific community, uses a Python-based declarative language to define workflows. It excels in bioinformatics and computational biology applications, where modular, reproducible workflows are critical. Snakemake's ability to optimize resource allocation dynamically ensures efficient utilization of computational infrastructures[15].

Pegasus

Pegasus Workflow Management System is tailored for large-scale scientific applications. It supports workflows across distributed systems, such as grids and clouds, and integrates with monitoring tools to track performance and

reliability. Pegasus excels in managing fault tolerance and providing provenance tracking, which are crucial for reproducibility in scientific research[10].

Luigi
Developed by Spotify, Luigi focuses on long-running batch processes and complex dependency management. Its Python-based configuration allows developers to define intricate workflows easily. While commonly used in data engineering tasks, Luigi's ability to handle failures and retry steps makes it versatile in other domains[11].

2.1.4 International Research status

In recent years, the adoption of standardized workflow languages and API orchestration frameworks has gained momentum in computational platforms across various domains. Research institutions and industries in countries like the United States, Canada, and Europe have prioritized the use of Common Workflow Language (CWL) and OpenAPI Workflows Specification for their flexibility and adaptability.

Key projects in the domestic research landscape include the application of CWL in bioinformatics pipelines. For instance, organizations such as the Broad Institute have successfully integrated CWL into their platforms to enable portable and reproducible workflows in genomics research. Similarly, OpenAPI specifications are widely employed in enterprise systems to manage API interactions, notably by companies developing API management tools like Postman and SwaggerHub[16].

However, platforms like Brane, specifically tailored to personalized interventions, remain underrepresented in this integration effort. Research efforts focus primarily on individual implementations rather than creating holistic systems that address workflow portability and API orchestration simultaneously. This research seeks to bridge this gap by adapting CWL within the Brane ecosystem.

Globally, the adoption of CWL has been even more pronounced, driven by

the demand for interoperability in multi-cloud environments and collaborative research efforts. In Europe, initiatives like the ELIXIR platform have incorporated CWL to standardize workflows across distributed bioinformatics resources. Similarly, in Asia, large-scale data platforms in Japan and South Korea have embraced CWL to enhance scalability and efficiency in healthcare analytics and AI-driven research[17].

The OpenAPI Workflows Specification, though a relatively recent addition, has been a focus of international organizations working on API ecosystems. The OpenAPI Initiative, supported by tech giants such as Google and IBM, has made significant strides in creating extensions like the OpenAPI Workflows, which allow the seamless chaining of API operations. For instance, these frameworks are critical for orchestrating APIs in microservices architectures widely used in international cloud platforms[18].

Despite these advancements, challenges persist in implementing these standards into domain-specific platforms like Brane. This gap presents an opportunity to leverage international expertise while adapting these technologies to meet the unique needs of personalized intervention platforms.

2.2 Architecture of the Brane Framework

The Brane framework is a container-based orchestration platform designed to simplify the execution and management of workflows across multi-site, heterogeneous computing environments. Its architecture emphasizes modularity, programmability, and flexibility to accommodate diverse computational tasks and resources. This section provides an in-depth description of Brane's architecture and its core components, showcasing its design to support multi-site orchestration and user-friendly programmability.

2.2.1 Modular Design

Brane employs a modular architecture to decouple various responsibilities, enabling scalability and ease of maintenance. Its modular design ensures that each component of the framework can evolve independently while interacting seamlessly with others. Key components include:

Package Management System: Facilitates the creation, storage, and retrieval of reusable functionalities encapsulated as containerized packages. These packages are stored in a package registry and serve as building blocks for workflows.

Domain-Specific Languages (DSLs):
Brane provides two DSLs for workflow definition:

- BraneScript: A C-like language for users with programming expertise, offering granular control over workflow orchestration.
- Bakery: A more human-readable DSL tailored for non-technical users, simplifying workflow composition.

Containerization

The framework relies heavily on containerization technologies such as Docker and Open Container Initiative (OCI)-compliant tools to achieve portability, scalability, and isolation[19]. Each package or task in a workflow is encapsulated in a container, ensuring that:

- Dependencies are bundled with the application, eliminating compatibility issues.
- Workflows can execute consistently across different environments, from on-premise clusters to cloud platforms.

Multi-Site Orchestration

One of Brane's defining features is its ability to orchestrate workflows across multiple geographically dispersed sites. This is achieved through a clear *Separation of Concerns (SoC)* across different user roles, as depicted in Figure 2.2. Brane divides responsibilities into distinct layers:

Brane divides responsibilities into distinct layers:

- **System Engineers** manage infrastructure configurations and network setups.
- **Software Engineers** develop containerized packages and register them in the system.

- **Domain Experts** compose workflows using Brane's DSLs without needing extensive technical knowledge.

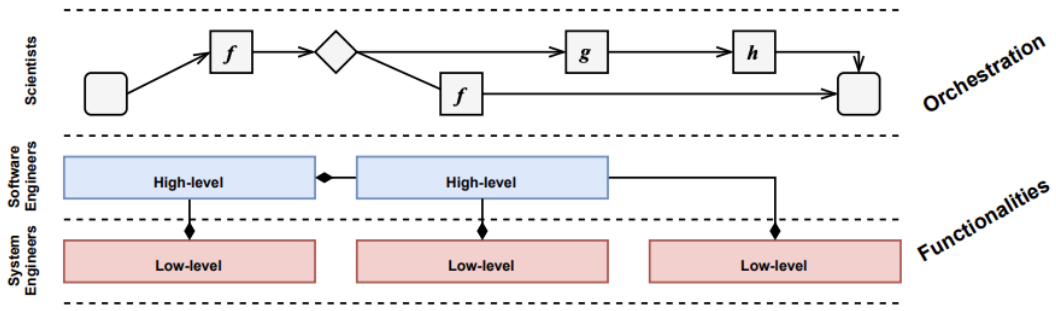

FIGURE 2.2: Separation of concerns (SoC)[1]

Tasks are dynamically allocated to the most suitable resources across multiple sites, considering factors like data locality, compute capacity, and network performance.

Runtime system

The runtime system in Brane handles the execution of workflows and manages dependencies, resource allocation, and task scheduling. It ensures:

Fault Tolerance: By isolating tasks in containers, the system minimizes the impact of failures on other workflow components.

Efficient Resource Utilization: The runtime dynamically adjusts resource allocation to optimize performance across heterogeneous environments.

Package registry

The package registry is a centralized component that stores containerized packages with metadata describing their inputs, outputs, and computational requirements.It also allows users to share and reuse packages, fostering collaboration and reducing duplication of effort.

Example Code:
This example(Listing 2.1) demonstrates how a workflow can be defined in the Brane framework using `BraneScript`, a domain-specific language (DSL) for composing and orchestrating workflows.

```
import grep
import wc

# Define the workflow function
fn process_log(log: file) -> int {
    # Step 1: Filter lines containing "error"
    let filtered_output = grep::filter(log, "error");

    # Step 2: Count the number of lines in the filtered output
    let line_count = wc::count_lines(filtered_output);

    return line_count;
}

# Define the input file
let log_file = "sample.log";

# Run the workflow and print the result
let error_lines = process_log(log_file);
print("Number of error lines:", error_lines);
```

LISTING 2.1: Brane Workflow Definition (brane-workflow.bs)

2.3 Explanation of the Brane Workflow

- **Imports:** The script imports two predefined Brane packages: `grep` for filtering and `wc` for counting lines.
- **Workflow Function:**
  - Defines a function `process_log` that takes a log file and returns the number of error lines.
  - Calls the `grep::filter()` function to extract lines containing "error."
  - Calls the `wc::count_lines()` function to count the filtered lines.
- **Execution:** The script loads a log file, runs the workflow, and prints the result.

2.4 CWL

The Common Workflow Language (CWL) is a standardized, platform independent specification designed to describe computational workflows and

tools. Established in 2014, CWL aims to enhance the portability and reproducibility of data analyses across diverse computational environmentscite. By providing a common framework, CWL enables researchers to define and share workflows that can be executed on various platforms without modification. This standardization facilitates collaboration and ensures consistent results across different systems. CWL's declarative nature allows users to specify the inputs, outputs, and execution requirements of each workflow component, abstracting the underlying computational details. This abstraction promotes interoperability and simplifies the adaptation of workflows to new environments.

One of the primary goals of CWL is to provide standardization in workflow descriptions, ensuring that workflows can be executed consistently across different computing environments, including high-performance computing (HPC) clusters, cloud platforms, and local workstations. The CWL standard achieves this by:

- **Using a declarative syntax:** CWL defines workflows in a structured format, typically using YAML or JSON, specifying input and output data, tool execution, and dependencies.
- **Separating workflow logic from execution:** Unlike many workflow management systems (WMSs) that tightly integrate workflow definitions with execution engines, CWL is independent of specific runtime implementations.
- **Facilitating interoperability:** CWL supports execution on multiple platforms through compatible workflow runners such as cwltool, Toil, and Arvados[20].

Central to CWL are two primary components: **Tool Definitions and Workflow Descriptions.**

Tool Definitions: Inputs, Outputs, and Commands
CWL's Tool Definitions specify how individual command-line tools are described, focusing on their inputs, outputs, and execution commands. Key elements include:

- **Inputs:** Define the parameters required by the tool, such as files, directories, or primitive data types. Each input is characterized by an identifier (id) and a data type, ensuring that the tool receives the necessary information to operate correctly.
- **Outputs:** Specify the results produced by the tool upon execution. Outputs are also identified by an id and a data type, detailing what the tool will generate, such as processed files or data structures.
- **Base Command:** Indicates the primary executable or command that the tool runs. This field outlines the specific command-line instruction to be executed, forming the core operation of the tool.

By clearly defining these components, CWL ensures that tools are described in a consistent and platform-independent manner, facilitating their integration into larger workflows.

Workflow Descriptions: Dependency and Task Flow
Workflow Descriptions in CWL outline how individual tools (or steps) are connected, detailing the sequence and dependencies between tasks. Essential aspects include:

- **Steps:** Represent individual tasks or tool executions within the workflow. Each step is defined by specifying the tool to be executed and its associated inputs and outputs.
- **Dependencies:** Managed through the source field, which connects the output of one step to the input of another. This linkage establishes the execution order, ensuring that a step commences only after its prerequisite steps have successfully completed and provided the necessary data.

By structuring workflows with explicit steps and dependencies, CWL enables the creation of complex, reproducible, and portable computational processes.

Example Code:

```
1 cwlVersion: v1.2
2 class: Workflow
3
```

```
# Define workflow inputs
inputs:
  log_file:
    type: File

# Define steps (each calling a separate CWL tool)
steps:
  filter_errors:
    run: grep-tool.cwl
    in:
      input_file: log_file
      pattern: { default: "error" }  # Filters lines containing "error"
    out: [filtered_output]

  count_lines:
    run: count-lines.cwl
    in:
      input_file: filter_errors/filtered_output
    out: [line_count]

# Define workflow outputs
outputs:
  error_line_count:
    type: int
    outputSource: count_lines/line_count
```

LISTING 2.2: CWL Workflow Definition (count-workflow.cwl)

```
cwlVersion: v1.2
class: CommandLineTool
baseCommand: ["grep"]

inputs:
  pattern:
    type: string
    inputBinding:
      position: 1
  input_file:
    type: File
    inputBinding:
      position: 2

outputs:
  filtered_output:
    type: File
    outputBinding:
```

```
      glob: "filtered.txt"

stdout: filtered.txt
```

LISTING 2.3: CWL Tool Definition (grep-tool.cwl)

```
cwlVersion: v1.2
class: CommandLineTool

# Define the command to be executed
baseCommand: ["wc", "-l"]

# Define inputs
inputs:
  input_file:
    type: File
    inputBinding:
      position: 1  # The input file appears as the first
   argument in the command

# Define outputs
outputs:
  line_count:
    type: int
    outputBinding:
      glob: "*"  # Capture standard output

# Runtime hints (optional)
hints:
  DockerRequirement:
    dockerPull: "ubuntu:latest"
```

LISTING 2.4: CWL Tool Definition (count-lines.cwl)

`count-workflow.cwl(Listing 2.2):`
The `grep-tool.cwl` extracts lines containing `"error"` from a log file.
The `count-lines.cwl` then counts the number of lines in the filtered output.
The final workflow (`count-workflow.cwl`) ties everything together, ensuring that the steps execute in the correct order.

`grep-tool.cwl(Listing 2.3):`
Tool that filters lines matching a given pattern.

`count-lines.cwl(Listing 2.4):`
Tool that takes a file as input and counts the number of lines using the Linux `wc -l` command.

- **Inputs:**
  - `input_file` - A file whose lines need to be counted.
- **Outputs:**
  - `line_count` - The number of lines in the input file.
- **Execution:**
  - The tool runs inside a Docker container using the `ubuntu:latest` image.

2.5 Differences Between CWL and Brane Code

While both CWL and Brane aim to define workflows in a structured and reproducible manner, they differ in several key ways:

Programming Paradigm

- **CWL:** Uses a *declarative* approach, where users describe workflows in YAML format without specifying execution logic explicitly.
- **Brane:** Uses an *imperative* approach through `BraneScript`, allowing users to define execution logic programmatically.

Component Modularity

- **CWL:** Workflows are composed of standalone tool definitions ('.cwl' files), each specifying inputs, outputs, and commands.
- **Brane:** Workflows leverage **pre-packaged functions** (e.g., `grep::filter`, `wc::count_lines`) from the Brane registry, abstracting low-level execution details.

Execution Model

- **CWL:** Requires a workflow runner (e.g., 'cwltool') to interpret and execute workflows in a containerized environment.
- **Brane:** Directly executes workflows within the Brane runtime, managing tasks dynamically across multi-site infrastructure.

Flexibility and Complexity

- **CWL:** Requires detailed YAML specifications but ensures strict reproducibility and compatibility with external workflow engines.
- **Brane:** Provides a more flexible, script-like syntax that is closer to general programming languages, making it easier for developers to integrate logic dynamically.

Use Case Suitability

- **CWL:** Well-suited for scientific workflows requiring strict reproducibility and integration with bioinformatics tools.
- **Brane:** More appropriate for multi-site orchestration, distributed computing, and scenarios where workflow logic needs to be dynamically controlled.

Error Handling

- **CWL:** Relatively basic; workflows typically fail at the point of error unless wrapped in external error-handling constructs or extensions.
- **Brane:** Allows try-catch mechanisms and explicit error management inside workflows, improving robustness for long-running or distributed workflows.

Ecosystem and Community Support

- **CWL:** Backed by a broad community across bioinformatics, genomics, and medical research. Integrated into platforms like Galaxy, Seven Bridges, and Dockstore.
- **Brane:** A newer, research-driven project with a growing but smaller ecosystem primarily targeting distributed systems research, edge-cloud computing, and scientific computing applications.

Both CWL and Brane serve critical roles in workflow orchestration but cater to different audiences and use cases. CWL is ideal for scientific reproducibility, while Brane offers a more flexible and programmable approach suitable for distributed computing environments.

Feature	**CWL**	**Brane**
Approach	Declarative (YAML)	Imperative (BraneScript)
Execution	Workflow runner (cwltool)	Direct execution in Brane runtime
Modularity	Separate tool definitions	Pre-packaged functions
Flexibility	Strict structure	Dynamic and programmable
Error Handling	Basic (fail on error)	Try-catch error management inside workflows
Ecosystem and Community Support	Broad, mature (bioinformatics, genomics)	Emerging, research-focused (distributed computing)
Use Case	Bioinformatics, research reproducibility	Multi-site orchestration, dynamic workflows

TABLE 2.1: Comparison of CWL and Brane Frameworks

Chapter 3

CWL integration

Brane is a platform designed to facilitate the seamless execution of distributed workflows across heterogeneous infrastructures. Originally, Brane focused primarily on supporting its native domain-specific languages such as BraneScript and Bakery. However, as scientific workflows became increasingly standardized around the Common Workflow Language (CWL), it became essential to extend Brane's capabilities to also handle CWL workflows.

This chapter discusses the implementation process, challenges faced, and current results of integrating CWL support into Brane. Additionally, it outlines areas of potential future development.

3.1 Implementation

Initial Goal

The primary objective was to allow users to build and run CWL workflows on Brane as first-class citizens, similar to how BraneScript and Bakery workflows are supported. This involves:

- CWL Parsing: Implement a parser capable of reading CWL v1.1 CommandLineTool documents.
- Docker Image Creation: Build a Docker image from the generated files to encapsulate the CWL tool, ensuring reproducibility.
- Package Generation: Translate a CWL document into a Brane package, including automatically generating: Package.toml (for legacy support and information structuring), package.yml (for full runtime integration), Dockerfile and execution scripts necessary to run the workflow.
- Supporting brane package build and brane cwl commands

CWL parsing

The cwl Rust crate is used to parse CWL v1.1 files. A new module cwl.rs was introduced inside the brane-cli crate, with a function: `pub async fn handle(path: PathBuf) -> Result<()>`
This function is responsible for:

- Reading the CWL file into memory
- Parsing it into a `CwlDocument`
- Extracting workflow metadata (name, version, description)
- Building an output structure in target/generated/

Initially, the focus was on CommandLineTool-type CWL documents to establish a solid foundation.

Package Generation

For each parsed CWL workflow:

- A Dockerfile is created, installing cwltool inside a Debian-based container.
- An entry.sh script is generated, defining the container's entry point.
- The CWL file itself is copied into the image's working directory.
- A Brane Package.toml and package.yml are generated, registering the tool in a Brane-compatible way.

The Brane CLI (brane-cli) was extended with a new pathway: `brane package build -kind cwl: Allows users to package CWL files manually.`

3.2 CWL v1.1 Specification and Scope of Implementation

The *Common Workflow Language (CWL)* is a widely adopted, declarative language designed to describe analysis workflows and command-line tools in a portable and platform-agnostic manner. CWL version 1.1 introduces a rich feature set that enables reproducibility and scalability in data-intensive research environments.

Key Components of CWL v1.1

- **CommandLineTool**: Specifies a single command-line tool, detailing inputs, outputs, arguments, and environment variables.
- **Workflow**: Defines a pipeline of steps where each step may be a `CommandLineTool` or another `Workflow`, enabling modular composition.
- **Inputs and Outputs**: Declarative parameter definitions for tools or workflows, often tied to specific types such as `File`, `int`, or `string`.
- **Expressions**: Support for dynamic evaluation of values using JavaScript, particularly useful in deriving outputs or transforming inputs.
- **Requirements and Hints**: Mechanisms for specifying constraints and execution environments, such as Docker containers or hardware resources.

Subset Implemented in Brane

Given the prototypical nature of the integration, only a limited subset of the CWL v1.1 specification has been implemented. The table below summarizes the supported and unsupported components.

TABLE 3.1: CWL v1.1 Support Matrix in Brane

Feature	CWL v1.1	Brane Implementation
`CommandLineTool`	✓	✓(fully supported)
`Workflow`	✓	× (not yet supported)
`Inputs and Outputs`	✓	✓(basic types only)
`DockerRequirement`	✓	✓(via Dockerfile injection)
`Expressions`	✓	× (not yet supported)
`ResourceRequirement`	✓	× (not implemented)
JavaScript Expressions	✓	× (not implemented)

This initial implementation focuses on supporting `CommandLineTool` definitions with static input/output mappings and containerized execution. The aim was to achieve a functional baseline that demonstrates feasibility, with a modular design allowing future support for additional CWL features such as nested workflows and dynamic expressions.

Justification of Scope

The decision to focus on a subset of CWL was driven by several factors:

1. **Time Constraints:** Implementing full CWL v1.1 compliance would require significantly more time and architectural changes to Brane's runtime and CLI logic.
2. **Proof of Concept:** The goal of the thesis was to demonstrate feasibility—i.e., that CWL tools can be parsed, dockerized, and run via Brane's ecosystem.
3. **Incremental Complexity:** By limiting support to `CommandLineTool`, we avoided complications like DAG(directed acyclic graph) resolution, step dependencies, and expression evaluation.
4. **Alignment with Brane's Design:** Brane's package model naturally maps to CWL tools as containerized units, making `CommandLineTool` a perfect initial candidate for integration.

Planned Extensions

While the current scope proves CWL integration is possible, future work can extend support to full workflows, complex types, and runtime expressions. A roadmap for this work is discussed in the Future Work section.

3.3 Challenges and Engineering Decisions

During the development of CWL integration into Brane, several technical and architectural challenges emerged. This section outlines the most significant obstacles and explains the rationale behind key design decisions.

YAML Parsing

CWL documents are authored in YAML and often include advanced features such as anchors, references, and schema extensions. We utilized the `cwl-rs` crate to parse CWL v1.1 files into a Rust-friendly AST(Abstract Syntax Tree). However, this introduced two difficulties:

- **Incomplete Rust Bindings:** The `cwl-rs` library had partial support for the full CWL spec, requiring manual extensions or fallbacks for certain constructs.

- **Error Diagnosis:** YAML parsing errors were cryptic, especially with complex CWL files, which slowed down debugging and testing.

To mitigate these issues, we constrained the scope of accepted CWL inputs to well-formed `CommandLineTool` definitions and introduced descriptive error messages during parsing.

Mapping Declarative to Imperative Execution

CWL is inherently declarative, while Brane packages expect imperative logic encoded in entrypoint scripts. Bridging this gap required:

- Generating a custom `entry.sh` script that serves as the executable command for the Brane container. This script invokes `cwltool` within the Docker container.
- Treating the CWL file as a static asset and wrapping it within a consistent shell entrypoint.

This design trades flexibility for simplicity, enabling fast integration at the cost of not leveraging CWL's full dynamic capabilities (e.g., parameter expressions).

Dockerization

CWL's support for containerized execution (via `DockerRequirement`) aligned well with Brane's own Docker-based runtime. However, dynamic Docker image resolution was not implemented. Instead:

- A static `Dockerfile` was generated for each CWL tool.
- This file installs `cwltool` and bundles the relevant CWL file and entry script.

This allowed us to build deterministic and reproducible Docker images for each CWL package but required additional storage and build time for each tool, even if identical.

CLI Integration with Brane

Integrating CWL support into the Brane CLI required modification of the main entrypoint logic to:

- Add a new subcommand: `brane cwl <file>`, which triggers CWL parsing and package generation.
- Maintain compatibility with existing `brane package build` and `brane package load` workflows, which initially assumed only ECU/Bakery-based packages.

Since `PackageKind::Cwl` was not originally supported in Brane's CLI architecture, temporary patches were applied to skip type-based dispatch where necessary. Future improvements should refactor CLI logic to treat `CWL` as a first-class citizen.

Trade-offs

The overarching trade-off throughout the project was between generality and deliverability. Instead of attempting full CWL 1.1 coverage—which would have demanded significant time and architectural changes—we focused on a stable subset that proved feasibility and aligned with Brane's container-based paradigm. This decision allowed us to implement end-to-end CWL-to-Brane translation, test Docker execution, and demonstrate integration with Brane's runtime, all within the thesis timeline.

Technical chanllenges

Several challenges arose during the integration:

- Error Type Mismatch: CWL parsing used anyhow::Error, while Brane expected specific custom error types (BuildError, CliError). This required careful wrapping of errors.
- Future vs Result Mismatch: Since handle() was an async fn, calls to it needed explicit .await handling, and initial attempts forgot this, causing map_err issues.
- Docker Build Failures: Misconfigured Dockerfiles or missing CWL dependencies (e.g., cwltool) initially caused build problems.

Despite these hurdles, a minimally viable integration was achieved.

Current Status

Feature	Status
CWL CommandLineTool parsing	Successfully parses CWL v1.1 CommandLineTool files
Brane package generation	Generates valid 'Package.toml' and 'package.yml' for CWL workflows
Docker image building	Builds and loads Docker images automatically for CWL workflows
CLI integration	Available via both `brane package build -kind cwl` and `brane cwl` commands
Error handling	Integrated into Brane's error system using `CliError` and `BuildError`
Architecture compatibility	Follows existing Brane CLI modular structure and practices

TABLE 3.2: Current Status of CWL Integration into Brane

3.4 Validation and Testing

To ensure the correctness and reliability of the CWL-to-Brane translation process, several validation and testing strategies were employed. These addressed both the syntactic accuracy of the parser and the functional integrity of the generated Brane packages.

Parser Validation

The parser's primary responsibility is to accurately interpret CWL v1.1 documents, specifically 'CommandLineTool' definitions. To verify correctness:

- **Schema compliance:** Each parsed CWL file was validated against the CWL v1.1 schema using the `cwltool` utility. This ensured that the files being parsed were syntactically valid according to the CWL standard.
- **Round-trip inspection:** After parsing, the extracted fields (e.g., `name`, `version`, `inputs`, `baseCommand`) were printed in human-readable format during development to manually verify their values.
- **Error handling:** The parser was tested with malformed CWL files to ensure that appropriate error messages were shown, e.g., missing fields, invalid types, or unsupported classes.

Generated Package Verification

The correctness of the generated Brane package was evaluated through the following steps:

- **Package structure checks:** The output directory (`target/generated/`) was inspected for the presence of required files, such as `Dockerfile`, `entry.sh`, `package.yml`, and the original CWL file.
- **Docker image build test:** The `docker build` process was invoked and monitored for successful completion. Any errors (e.g., missing base image, permissions) were logged and addressed.
- **Brane integration test:** The resulting package was loaded into Brane using `brane package load`, after which it became visible via `brane package list`. The package could then be invoked within a workflow for end-to-end testing.

Tested Workflows and Output Comparison

The implementation was evaluated using a minimal CWL example: the canonical `hello_world.cwl`, which executes a basic echo command. This example was selected for its simplicity and clarity.

- The workflow was first executed using `cwltool` to observe the expected output: `"Hello World"`.
- The same CWL file was then translated into a Brane package using the CLI: `brane cwl hello_world.cwl`.
- After loading the package into Brane, a BraneScript workflow was constructed to invoke the `hello_world` package.
- The output produced by Brane was compared against the original `cwltool` output and matched exactly.

This comparative testing ensured semantic fidelity of the translated workflow. Although only a minimal case was tested in-depth, the modular design supports gradual extension to more complex workflows in the future.

Chapter 4

Related work & Methodology

The execution of computational workflows has been a vital topic in distributed computing, bioinformatics, and cloud systems. Over the years, multiple workflow management frameworks and standards have been developed to address challenges of portability, scalability, and reproducibility. This chapter presents an overview of relevant works that have influenced or relate to the objectives of this study, which focuses on integrating the Common Workflow Language (CWL) into the Brane orchestration framework.

4.1 Workflow Management Systems

Several early workflow management systems laid the foundation for modern orchestration approaches. For instance, Pegasus was introduced as a scientific workflow management system to automate large-scale data analysis tasks, especially in the context of distributed and grid computing environments [10]. Pegasus emphasized abstraction, enabling users to define workflows at a high level, while the system managed resource mapping and execution.

Similarly, the Taverna Workbench allowed scientists without deep technical expertise to create complex bioinformatics workflows **Taverna**. Taverna's focus on user accessibility, however, came at the cost of flexibility and fine-grained execution control.

In contrast, newer frameworks such as Nextflow [21] and Snakemake [22] offer more programmable workflow descriptions, leveraging technologies like Docker and cloud-native infrastructures to ensure reproducibility and scalability. These systems support parallel execution, dynamic scheduling,

and compatibility with heterogeneous resources, pushing workflow management closer to the needs of modern distributed systems.

Brane emerges from this lineage, designed to address orchestration across multiple sites, where concerns such as data locality, resource distribution, and cross-infrastructure deployment become critical [1].

4.2 Standardization of Workflow Descriptions

The growing complexity of scientific workflows, combined with the need for reproducibility across different environments, has driven efforts toward standardization. The Common Workflow Language (CWL) [2] was developed to offer a declarative, portable, and platform-independent specification for describing command-line tool usage and workflow structures.

Crusoe et al. [23] proposed CWL with the goal of enabling workflows to be defined once and executed consistently across diverse backends. CWL separates tool descriptions from workflow descriptions, emphasizing modularity and clear definition of inputs, outputs, and computational requirements. This approach promotes interoperability among institutions and platforms, as workflows can be shared without binding to specific infrastructure assumptions.

Frameworks such as Toil [24] and Rabix [25] support CWL execution, fostering a rich ecosystem. However, CWL's strict declarative design may limit dynamic adaptation during execution, which is increasingly necessary in distributed, heterogeneous environments—a gap this project aims to bridge by integrating CWL workflows into the dynamic, multi-site Brane system.

4.3 Containerization in Workflow Execution

Containerization technologies have played a transformative role in workflow execution, ensuring that tools can run reproducibly across systems without dependency conflicts. Docker [4] and Singularity [26] are prominent solutions that encapsulate application binaries and environments into portable containers.

Scientific workflow systems, including those compliant with CWL, heavily rely on containerization to guarantee execution fidelity. Brane also leverages container technology, allowing tasks to be executed consistently across distributed sites while minimizing configuration overhead [1].

However, Brane extends the concept by dynamically allocating tasks based on site capabilities, bandwidth constraints, and compute availability, an execution flexibility that pure container-based CWL runners traditionally lack.

4.4 Security, Privacy, and Policy Management in Distributed Workflows

As workflows increasingly operate across organizational and national boundaries, concerns over data security and privacy become paramount. Research efforts such as "Exploring the Enforcement of Private Dynamic Policies on Medical Workflow Execution" [27] have highlighted the need for policy-aware orchestration, where workflows must adapt dynamically to security requirements.

While CWL focuses primarily on execution portability and reproducibility, frameworks like Brane are positioned to incorporate dynamic policies regarding data movement, compute location, and compliance constraints. Understanding these complementary strengths forms a crucial motivation for extending Brane's capabilities through CWL integration.

4.5 Inclusion and Exclusion Criteria

In assembling this related work chapter, priority was given to peer-reviewed papers, journal articles, and conference proceedings indexed in databases such as IEEE Xplore, ACM Digital Library, ScienceDirect, SpringerLink, and Google Scholar. Grey literature such as blogs, Wikipedia articles, or non-peer-reviewed reports were excluded to ensure academic rigor and reliability of referenced material.

4.6 Summary

This review of existing workflow management systems, workflow standardization initiatives, containerization technologies, and policy-enforcement approaches provides context for this research. While CWL offers a robust standard for tool and workflow description, its static design contrasts with Brane's dynamic, distributed execution model. The integration of CWL into Brane aims to combine the strengths of both worlds: achieving platform-independent workflow definitions while supporting scalable, adaptable orchestration across heterogeneous computing sites.

4.7 Methodology

This chapter describes the research methods used to investigate the integration of the Common Workflow Language (CWL) into the Brane framework. It outlines the type of research conducted, data sources consulted, selection criteria for relevant literature, and the experimental steps followed during the development and analysis phases.

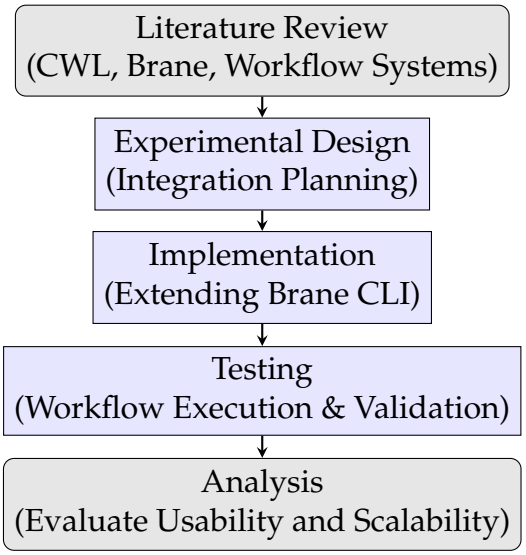

FIGURE 4.1: Overview of the research methodology used in this study.

4.8 Research Methods

The methodology section presents the approach followed to collect, analyze, and synthesize information to address the research questions.

4.8.1 Type of Study

Two main types of research methodologies exist: qualitative and quantitative.
Qualitative research seeks to understand complex phenomena through non-numerical data such as literature reviews, observations, and interviews.
Quantitative research focuses on measurable data, applying statistical or computational methods to derive conclusions.
Since this study focuses on understanding the technological gaps between current workflow management solutions and proposing an integration design between CWL and Brane, a **qualitative research approach** is the most appropriate. Through qualitative methods, a deep understanding of system architectures, workflow specifications, and interoperability challenges is achieved by reviewing existing literature, analyzing technical standards, and evaluating real-world implementations.

4.8.2 Analysis Approach

This research adopts a literature-driven analysis combined with system-level experimentation. Relevant academic publications, technical specifications, and software documentation were reviewed to:

- Map the features, strengths, and limitations of CWL and Brane.
- Identify challenges in achieving workflow portability and dynamic execution.
- Investigate best practices in orchestrating multi-site computational workflows.

Subsequently, experimental implementation activities involved extending the Brane CLI and runtime to support CWL workflows. Findings from these experiments were cross-referenced against the literature review results.

4.8.3 Data Sources

Data was collected from:

- Peer-reviewed conference and journal papers published in venues like ACM, IEEE, Springer, and Elsevier.
- Official documentation of CWL, Brane, Docker, and related technologies.
- Open-source repositories and technical white papers when official documentation was not sufficient.

Non-academic sources such as blogs, forums, and Wikipedia articles were excluded to ensure scientific rigor.

4.9 Experimentation: CWL Integration into Brane

The experimental phase involved the design, implementation, and testing of an integration layer between CWL and Brane. The methodology followed included:

1. **Parsing CWL files:** Extending Brane's CLI to accept .cwl workflows and interpret their structure.
2. **Translating workflows:** Mapping CWL workflows into Brane's internal package and workflow formats, ensuring compatibility with BraneScript's execution model.
3. **Executing workflows:** Deploying and running CWL-defined workflows across distributed Brane environments.
4. **Testing and validation:** Executing a range of workflows (simple, multi-step, conditional) to verify correctness, error handling, and scalability.

This methodology ensured that the developed integration was systematically evaluated for functional correctness, usability, and performance characteristics.

Chapter 5

Future work

5.1 Future Work

While the current implementation successfully integrates `CommandLineTool` support from the Common Workflow Language (CWL) into the Brane ecosystem, there are several opportunities for future work that would significantly enhance the system's capabilities.

First and foremost, supporting full CWL `Workflow` documents is a logical next step. These workflows define complex, multi-step pipelines that chain together multiple tools with clearly defined input/output relationships and control flow. Implementing support for them would require recursive parsing of CWL documents and the generation of composite Brane packages that reflect these pipelines.

Another area for improvement involves type inference and metadata enrichment. Currently, the mapping between CWL types and Brane's internal type system is minimal. Introducing a more sophisticated translation layer would improve validation, enable richer tooling, and increase compatibility with the Brane runtime.

Additionally, enabling the use of remote CWL documents through HTTP(S) imports would support distributed and cloud-native scenarios. This requires implementing network-safe fetching and dependency resolution mechanisms.

Finally, user-facing improvements are also desirable. These include introducing pre-build validation for CWL files, verbose or dry-run modes for the build process, and better feedback for malformed or unsupported documents.

TABLE 5.1: Overview of Proposed Future Work

Feature	**Description**
Full Workflow Support	Parse and compile CWL `Workflow` documents into composite Brane packages.
Type Mapping	Improve mapping from CWL types/formats to Brane's type system for stronger typing and validation.
Remote Imports	Allow importing CWL files and their dependencies via remote URLs (e.g., GitHub or HTTP).
CLI UX Improvements	Add validation steps, dry-run options, and verbose logging to the build CLI.
New CWL Features	Support additional CWL constructs such as `Hints`, `Requirements`, and `Scatter`.

Chapter 6

Discussion

This chapter reflects on the design and implementation of integrating the Common Workflow Language (CWL) into the Brane framework. It addresses the challenges encountered during the development process, the trade-offs made in architectural choices, the implications for future workflow systems, and how the integration contributes to the broader goals of scientific reproducibility and distributed computing.

6.1 Design Trade-offs and Technical Decisions

One of the key decisions in this project was to implement CWL support as a separate translation layer within the Brane CLI, rather than embedding it directly into the core BraneScript runtime. This decision follows the principle of separation of concerns, allowing CWL workflows to be interpreted and converted into Brane-compatible packages without modifying the core execution engine.

While this modular approach improves maintainability and extensibility, it also introduces an intermediate translation step that can become complex, especially for deeply nested workflows. As such, the current implementation supports a substantial subset of CWL but does not yet achieve full feature parity.

Furthermore, the reliance on YAML parsing and mapping to Brane's 'package.yml' structure required careful attention to type compatibility and data binding semantics. Decisions had to be made regarding how CWL command-line tools would map onto Brane package actions and how to handle multiple input/output artifacts in a way that preserves reproducibility.

6.2 Challenges in Bridging Declarative and Imperative Models

A central challenge in this project was reconciling the declarative nature of CWL with the imperative design of BraneScript. CWL assumes a static, data-driven workflow definition in which the execution order is inferred from data dependencies. Brane, on the other hand, supports explicit control flow, function definitions, and dynamic branching.
This mismatch required the development of a mapping strategy where CWL steps are treated as isolated Brane actions, with BraneScript-generated code organizing these steps in a semantically equivalent execution order. However, complex control constructs in CWL (e.g., conditionals or dynamic subworkflows) do not always have a direct analogue in Brane, which required compromises in how expressive the translated workflows could be.
Future improvements could involve extending BraneScript with declarative constructs or hybrid parsing capabilities that can natively understand CWL structures.

6.3 Reproducibility vs. Adaptability

CWL is designed for strict reproducibility: given the same inputs and environment, it should always produce the same outputs. Brane, however, introduces dynamic elements—such as site-aware execution and runtime optimization—which prioritize adaptability and performance over deterministic execution paths.

The integration effort highlights an important tension in workflow design: the need to balance reproducibility (critical for scientific research) with adaptability (crucial for performance in distributed and resource-constrained environments). By enabling CWL workflows to run within Brane, this project contributes to a new paradigm where workflows can be portable and reproducible by design, but also optimized dynamically based on context.

6.4 Limitations of the Current Integration

While functional, the current CWL-to-Brane integration has several limitations:

- Not all CWL features are supported (e.g., expressions, conditionals, and custom JavaScript blocks).
- There is limited support for runtime resource hints (e.g., CPU, memory), which are currently ignored by Brane's scheduler.
- Error reporting during workflow translation could be more informative and developer-friendly.

These limitations are known and provide a roadmap for future development.

6.5 Generalization and Future Potential

Although this project focused specifically on CWL, the underlying architectural principles can be reused for supporting other workflow standards. Brane's modular CLI and runtime make it a promising platform for hosting multi-standard workflow execution capabilities.

Moreover, the project demonstrates that bridging formal workflow specifications with flexible, programmable orchestration is not only feasible but advantageous. It empowers users to reuse standardized workflows while gaining the benefits of Brane's dynamic, multi-site execution model.

6.6 Lessons Learned

Several key lessons emerged during this project:

- Workflow standardization alone is not sufficient—execution semantics must also align to ensure meaningful interoperability.
- Imperative and declarative paradigms can be reconciled, but doing so requires deliberate architectural abstraction.
- Simplicity and clarity in translation logic often outperform feature completeness, especially in early-stage prototypes.
- Containerization remains the cornerstone of reproducibility, and any orchestration framework must treat it as a first-class concern.

Chapter 7

Conclusion

This thesis explored the integration of the Common Workflow Language (CWL) into the Brane framework to bridge the gap between standardized, declarative workflow descriptions and dynamic, multi-site orchestration. The work demonstrates that it is technically feasible to parse CWL CommandLineTool files, encapsulate them into Brane packages, and enable execution using Brane's existing CLI and Docker infrastructure.

The integration successfully supports packaging, Docker image generation, and CLI interaction, contributing a minimally viable interface between CWL and Brane. Although the scope is currently limited to simple tools, this foundational work paves the way for broader support, including complex CWL workflows and dynamic execution patterns.

From a practical standpoint, the project enhances Brane's usability in scientific domains that have adopted CWL, thereby increasing its interoperability and adoption potential. From a theoretical perspective, it exemplifies how declarative and imperative paradigms can coexist in hybrid workflow systems. Future work will involve recursive workflow compilation, improved type mapping, remote imports, and user-centric enhancements. The lessons learned here contribute to the broader discourse on reproducible and adaptable computing in distributed environments.

Appendix A

Appendix

This appendix provides the source code for the Rust implementation that parses a CWL file and generates a Brane-compatible Docker package.

```
use std::collections::HashMap;
use std::fs::{self, create_dir_all, File, write};
use std::io::BufReader;
use std::path::PathBuf;
use std::process::Command;
use std::fmt::Write as _;

use anyhow::{Context, Result};
use cwl::v11::CwlDocument;
use specifications::version::Version;
use specifications::package::{PackageInfo, PackageKind};
use specifications::common::{Function, Type};
use brane_cli::errors::BuildError;

/// Parses a CWL file and generates a Brane-compatible package directory & Docker image.
pub async fn handle(path: PathBuf) -> Result<()> {
    // Open and parse CWL
    let file = File::open(&path).context("    Failed to open CWL file")?;
    let reader = BufReader::new(file);
    let document = CwlDocument::from_reader(reader).context("    Failed to parse CWL document")?;

    match &document {
        CwlDocument::CommandLineTool(tool) => {
            println!("    Parsed CWL CommandLineTool");

            // Extract fields
            let name = tool.schema.name.clone().unwrap_or_else(|| "unknown".into());
```

```
28              let version_str = tool.schema.version.clone().
    unwrap_or_else(|| "0.1.0".into());
29              let description = tool.label.clone().unwrap_or_else
    (|| "No description provided".into());
30
31              // Fallback hardcoded version
32              let version = Version::new(1, 0, 0);
33
34              // Prepare output
35              let out_dir = PathBuf::from(format!("target/
    generated/{}", name));
36              create_dir_all(&out_dir).context("    Failed to
    create output directory")?;
37
38              // --- Package.toml ---
39              let mut toml = String::new();
40              writeln!(toml, "name = {:?}", name)?;
41              writeln!(toml, "version = {:?}", version_str)?;
42              writeln!(toml, "kind = \"cwl\"")?;
43              writeln!(toml, "description = {:?}", description)?;
44              write(out_dir.join("Package.toml"), toml).context("
        Failed to write Package.toml")?;
45
46              // --- entry.sh ---
47              let entry = "#!/bin/bash\ncwltool hello_world.cwl\n
    ";
48              write(out_dir.join("entry.sh"), entry).context("
    Failed to write entry.sh")?;
49
50              // --- Dockerfile ---
51              let dockerfile = r#"
52 FROM debian:bullseye-slim
53 RUN apt-get update && apt-get install -y cwltool
54 COPY hello_world.cwl /app/hello_world.cwl
55 COPY entry.sh /app/entry.sh
56 WORKDIR /app
57 RUN chmod +x entry.sh
58 CMD ["./entry.sh"]
59 "#;
60              write(out_dir.join("Dockerfile"), dockerfile).
    context("    Failed to write Dockerfile")?;
61
62              // --- Copy CWL ---
63              fs::copy(&path, out_dir.join("hello_world.cwl")).
    context("    Failed to copy CWL file")?;
64
```

```
65              // --- Docker build ---
66              println!("       Building Docker image...");
67              let image_name = format!("brane-cwl-{}:latest", name
    );
68              let status = Command::new("docker")
69                  .arg("build")
70                  .arg("--load")
71                  .arg("-t")
72                  .arg(&image_name)
73                  .arg(&out_dir)
74                  .status()
75                  .context("    Failed to invoke docker build")?;
76              if !status.success() {
77                  anyhow::bail!("    Docker build failed");
78              }
79
80              println!("    Docker image built: {image_name}");
81
82              // --- Create PackageInfo ---
83              let package_info = PackageInfo::new(
84                  name.clone(),
85                  version,
86                  PackageKind::Cwl,
87                  vec![],
88                  description.clone(),
89                  true,
90                  HashMap::new(),
91                  HashMap::new(),
92              );
93
94              // --- Write package.yml ---
95              package_info.to_path(out_dir.join("package.yml")).
    context("    Failed to write package.yml")?;
96
97              println!("       Brane CWL package available at:
    {}\\", out_dir.display());
98          }
99          _ => {
100             println!("       Unsupported CWL class: {:?}",
    document);
101         }
102     }
103
104     Ok(())
105 }
106
```

```
107 /// ‘brane package build‘ calls this entry point for CWL
        packages.
108 pub fn build(_workdir: PathBuf, file: PathBuf) -> Result<(),
        BuildError> {
109     println!("           Building Brane CWL package...");
110     futures::executor::block_on(handle(file))
111         .map_err(|e| BuildError::PackageInfoFromOpenAPIError {
        source: e })
112 }
```

LISTING A.1: CWL to Brane package builder in Rust(https://github.com/Hamza0320/brane_cwl)

Bibliography

[1] O. Valkering, R. Cushing, and A. Belloum, "Brane: A framework for programmable orchestration of multi-site applications," in *2021 IEEE 17th International Conference on eScience (eScience)*, IEEE, 2021, pp. 277–282.

[2] P. A. et al., "Common workflow language, v1.0," *Common Workflow Language Specification*, 2016. [Online]. Available: https://www.commonwl.org/.

[3] M. R. Crusoe, J. Chilton, S. Abeln, and et al., "Common workflow language v1.2," *F1000Research*, vol. 9, p. 295, 2022. DOI: https://doi.org/10.1177/1094342017704893.

[4] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *IEEE Cloud Computing*, 2014. DOI: 10.1109/MCC.2014.51.

[5] E. Deelman and et al., "The future of scientific workflows," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 159–175, 2018.

[6] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, 2014. DOI: 10.1109/MCC.2014.63.

[7] P. B. C. P. R. G. Kousalya, "Managing scientific workflows across heterogeneous environments," in *Advances in Workflow Management*, Springer, 2020, pp. 152–165. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-56982-6_8.

[8] T. Imkin, *How modern workflow orchestration solves scalability challenges*, 2021. [Online]. Available: https://temporal.io/blog/how-modern-workflow-orchestration-solves-scalability-challenges.

[9] R. F. d. Silva, L. Pottier, T. Coleman, E. Deelman, and H. Casanova, "Workflowhub: Community framework for enabling scientific workflow research and development," in *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, 2020, pp. 49–56. DOI: 10.1109/WORKS51914.2020.00012.

[10] E. D. et al., "The evolution of the pegasus workflow management software," *Computing in Science Engineering*, 2019. DOI: 10.1109/MCSE.2019.2919690.

[11] M. Barika, S. Garg, A. Y. Zomaya, L. Wang, A. van Moorsel, and R. Ranjan, "Orchestrating big data analysis workflows in the cloud: Research challenges, survey, and future directions," *ACM Computing Surveys*, vol. 52, no. 5, Article 95, 41 pages, Sep. 2019. DOI: 10.1145/3332301.

[12] NVIDIA Corporation, *Clara for biopharma: Drug discovery with generative ai*, https://www.nvidia.com/en-us/clara/biopharma/, Accessed: 2025-01-07, 2024.

[13] Y. Wang, Y. Guo, Z. Guo, W. Liu, and C. Yang, "Protecting scientific workflows in clouds with an intrusion tolerant system," *IET Information Security*, vol. 14, Mar. 2020. DOI: 10.1049/iet-ifs.2018.5279.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, ACM, 2007, pp. 59–72.

[15] S. Pohl, N. Elfaramawy, A. Miling, K. Cao, B. Kehr, and M. Weidlich, "How do users design scientific workflows? the case of snakemake and nextflow," in *Proceedings of the 36th International Conference on Scientific and Statistical Database Management (SSDBM)*, Rennes, France: Association for Computing Machinery, Jul. 2024, Article 95, 12 pages, ISBN: 979-8-4007-1020-9. DOI: 10.1145/3676288.3676290.

[16] D. Westerveld, *API Testing and Development with Postman: A practical guide to creating, testing, and managing APIs for automated software testing*. Packt Publishing Ltd, 2021.

[17] ELIXIR Europe, *Elixir: Uniting europe's biological data*, https://elixir-europe.org/, Accessed: 2025-01-07, 2025.

[18] J. Ponelat and L. Rosenstock, *Designing APIs with Swagger and OpenAPI*. Simon and Schuster, 2022.

[19] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[20] E. Deelman, Y. Gil, M. Livny, K. Kate, B. Berriman, and R. F. da Silva, "Towards a new paradigm for programming scientific workflows," *Future Generation Computer Systems*, vol. 138, pp. 46–62, 2023. DOI: 10.1016/j.future.2022.09.012.

[21] P. D. Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, C. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature Biotechnology*, vol. 35, no. 4, pp. 316–319, 2017. DOI: 10.1038/nbt.3820.

[22] J. Köster and S. Rahmann, "Snakemake—a scalable bioinformatics workflow engine," *Bioinformatics*, vol. 28, no. 19, pp. 2520–2522, 2012. DOI: 10.1093/bioinformatics/bts480.

[23] M. R. Crusoe *et al.*, "Methods included: Standardizing computational reuse and portability with the common workflow language," *Proceedings of the 2015 IEEE International Conference on e-Science*, pp. 370–377, 2015. DOI: 10.1109/eScience.2015.40.

[24] J. Vivian, A. A. Rao, F. A. Nothaft, *et al.*, "Toil enables reproducible, open source, big biomedical data analyses," *Nature Biotechnology*, vol. 35, no. 4, pp. 314–316, 2017. DOI: 10.1038/nbt.3772.

[25] G. C. Carrasco *et al.*, "Rabix: An open-source workflow executor supporting recomputability and interoperability of workflow descriptions," *Pacific Symposium on Biocomputing 2016*, pp. 154–165, 2016.

[26] D. Gannon and V. Sochat, "Singularity: A container system for hpc applications," *Cloud Computing for Science and Engineering*, 2017. [Online]. Available: https://www.researchgate.net/publication/317905090.

[27] C. Basescu, P. Druschel, M. Wählisch, and R. Kapitza, "Exploring the enforcement of private dynamic policies on medical workflow execution," in *Proceedings of the 2015 IEEE 35th International Conference on Distributed Computing Systems Workshops*, IEEE, 2015, pp. 27–32. DOI: 10.1109/ICDCSW.2015.14.