VisualML

An OOP-Oriented Workflow Orchestrator for Machine Learning Pipelines

Name: Farid Nowrouzi
Matriculation Number: 552343
University of Messina – Department of Computer Science
Course: Object-Oriented Programming
Instructor: Prof.Salvatore Distefano
Academic Year: 2024–2025

Submission Date: May 26th, 2025

Abstract

This report presents the design and implementation of **VisualML**, a JavaFX-based workflow orchestration system built using Object-Oriented Programming principles. The system enables visual creation, connection, and execution of machine learning pipelines, and demonstrates encapsulation, inheritance, abstraction, modularity, and other key OOP paradigms.

Contents

1 Introduction

This report presents the design and development of **VisualML**, an OOP-oriented workflow orchestrator for machine learning pipelines. Built entirely in Java using JavaFX, VisualML enables users to visually construct, connect, and execute modular steps in a data science workflow—such as preprocessing, training, inference, and evaluation—through an intuitive graphical interface.

The system was developed as a final project for the Object-Oriented Programming course at the University of Messina and serves as a hands-on demonstration of core OOP principles including encapsulation, abstraction, inheritance, polymorphism, composition, and modularity. These concepts are applied consistently across the system's layered architecture, which includes controller, model, and view packages, along with utility, factory, and command-based components.

The interface allows users to add various node types, connect them through directional arrows, and execute workflows dynamically while viewing logs in real time. Additional features include undo/redo functionality, workflow saving/loading in JSON format, execution animations, and contextual node editing through a sidebar panel.

VisualML is not just a student project—it is a foundational prototype for a scalable and extensible orchestration system. In the future, the system will be enhanced using software engineering principles such as service-oriented architecture and microservice integration. Each node in the workflow is envisioned to represent a stand-alone microservice, making it possible to deploy workflows on distributed systems or cloud platforms.

The goal is to evolve VisualML into a powerful orchestration engine suitable for real-world data workflows, while maintaining a strong object-oriented codebase. This report outlines the current implementation, the architectural decisions made, and the object-oriented patterns applied, laying the groundwork for future expansion.

2 System Overview

2.1 2.1 System Architecture

The VisualML application is built using the Model-View-Controller (MVC) architectural pattern. The `MainViewController` serves as the central coordinator between the graphical user interface (GUI), the logic layer, and the workflow model. The system enables users to visually design, manage, and execute complex machine learning workflows in a user-friendly canvas environment.

Each workflow is composed of modular nodes representing specific pipeline components (e.g., `TaskNode`, `ConditionNode`, `TrainingNode`, etc.), with connections drawn between them to define the data flow. These components are styled dynamically, can be dragged and repositioned freely, and support branching logic with YES/NO labels for conditional paths.

2.2 2.2 System Features

The system includes a rich set of interactive features, many of which are handled directly by the `MainViewController` class:

- **Node Creation:** Users can add new nodes using the `Create Node` dialog, specifying type, name, and details.
- **Node Connection:** Connections are created interactively, with optional custom labels (e.g., YES/NO).
- **Execution Engine:** Pressing `Execute Workflow` triggers a visual simulation of the workflow with real-time logs and node-specific behavior.
- **Sidebar Editing:** A sidebar displays and allows editing of selected node's ID, name, type (via ComboBox), details, and execution status.
- **Zooming and Panning:** The canvas supports smooth zoom in/out and free panning using mouse controls and buttons.
- **Undo/Redo Stack:** Every major action (create, delete, move, connect) is tracked and reversible.
- **Validation and Highlighting:** Workflow structure is validated with visual error highlights for disconnected nodes or structural violations (e.g., cycles or wrong number of branches).
- **Mini-map Navigation:** A real-time mini-map allows quick spatial orientation on large workflows.
- **Workflow Title Management:** The user can rename the current workflow using an in-app editable field.
- **Persistence:** Workflows can be saved to and loaded from disk in JSON format.

2.3 2.3 Component Roles

- **MainViewController:** GUI logic, visual layout, node handling, sidebar updates, and action events.
- **MainController:** Backend logic coordinator for node creation, connection, and validation.
- **WorkflowNode + Subclasses:** Core node representations (TaskNode, ConditionNode, etc.) containing type-specific logic and data.
- **Arrow:** Graphical connection with optional labels between nodes.
- **UndoableAction:** Tracks state changes to enable undo/redo.
- **WorkflowService:** Handles internal workflow representation and logic validation.

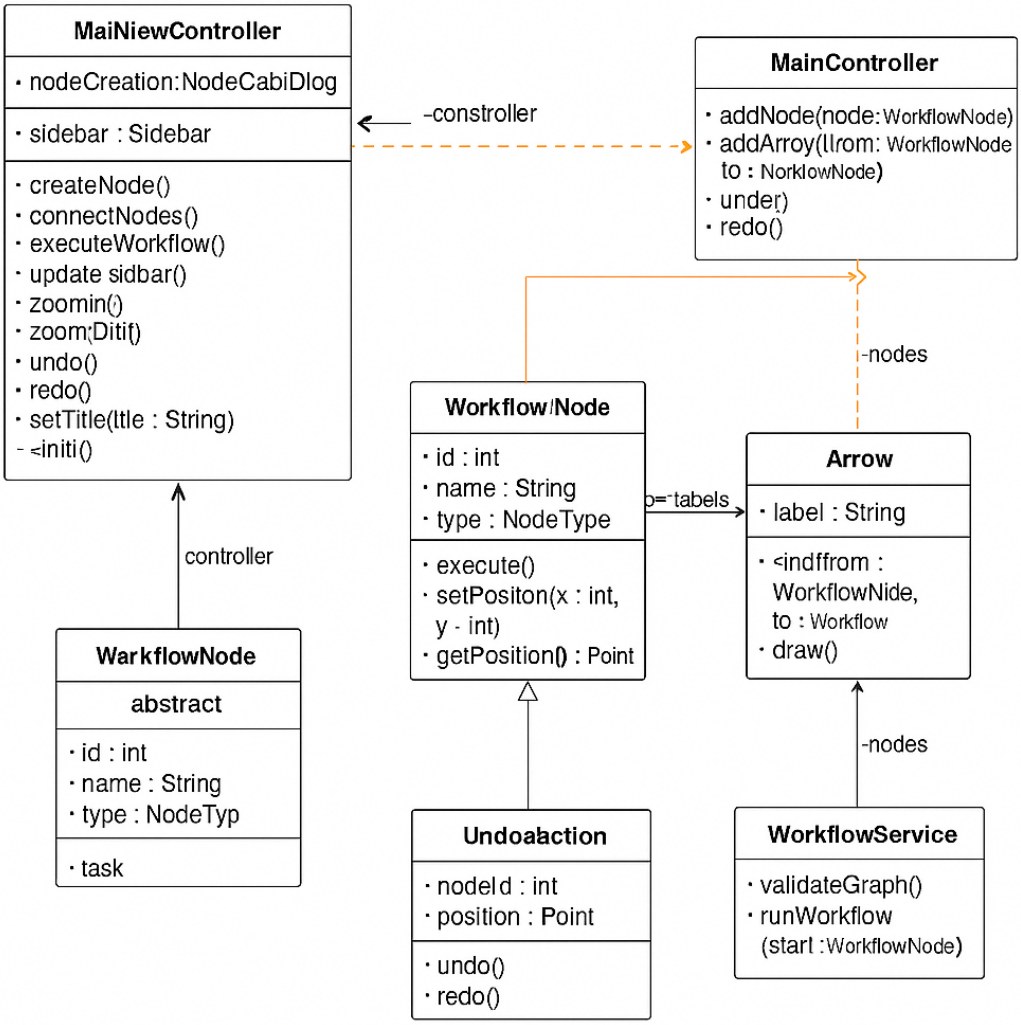

Figure 1: Simplified UML of `MainViewController` and its key interactions

2.4 2.4 UML Diagram

3 Object-Oriented Programming Principles

This section provides a comprehensive overview of the Object-Oriented Programming (OOP) principles that were applied in the design and implementation of the Visual Machine Learning Workflow Orchestrator. These principles serve as the backbone of a modular and maintainable software architecture.

The concepts demonstrated in this project include:

- **Encapsulation**
- **Abstraction**
- **Inheritance**
- **Polymorphism**
  (Subtyping/Inclusion, Parametric, Method Overloading, and Method Overriding)

- **Modularity**
- **Composition**
- **Information Hiding**
- **Hierarchy**
- **Reusability**

These fundamental pillars are supported by several advanced OOP-related design practices. While not considered "core" principles, they are essential for writing high-quality software and are addressed in a dedicated section that follows. These include:

- **Cohesion and Coupling**
- **Dependency Injection**
- **Aggregation**
- **The SOLID Principles**
- **Relevant Design Patterns (e.g., Factory)**

In the following subsections, we will explore each of the core principles listed above in detail, supported by code examples from the implemented system. Section **??** will then expand on the additional design principles that further strengthen the software architecture.

3.1 Encapsulation

Definition:
Encapsulation is a core object-oriented programming principle that involves bundling an object's internal state (its fields) together with the methods that manipulate that state. It also restricts external access to certain components using access modifiers like `private`, `protected`, and `public`.

Why It Matters:
Encapsulation protects an object's internal representation from unintended interference. By exposing only controlled interfaces, it enforces consistency, reduces the likelihood of errors, and enhances modularity, maintainability, and testability.

Application in the Project:
Encapsulation is widely used in our JavaFX-based Workflow Orchestration System. For instance, in the `MainViewController` class, critical fields such as the node map, undo/redo stacks, and workspace pane are declared as `private`. They are only accessible through designated public or protected methods, ensuring controlled interaction with the class's internal state.

Code Example:

Listing 1: Encapsulation in MainViewController.java

```
public class MainViewController {

    private final Map<String, VBox> nodeViewsMap = new HashMap<>();
    private final Stack<UndoableAction> undoStack = new Stack<>();
    private Pane workspacePane;

    public void addNodeToCanvas(WorkflowNode node) {
        addNodeToWorkspace(node); // Encapsulated access point
    }

    private void addNodeToWorkspace(WorkflowNode node) {
        // internal logic...
    }
}
```

Commentary:
By using `private` access modifiers, the class enforces a strict separation between internal state and external interaction. This design choice promotes safer code evolution, enables internal refactoring without affecting external classes, and ensures consistency across the system.

3.2 Abstraction

Definition:
Abstraction is a foundational principle of Object-Oriented Programming that involves exposing only the relevant attributes and behaviors of an object while hiding unnecessary implementation details. It allows developers to define templates (via abstract classes or interfaces) that specify what an object should do—without dictating how it should do it.

Abstraction operates at two levels:

- **Class-level abstraction:** achieved through abstract classes or interfaces that define a common contract.
- **Instance-level abstraction:** achieved by interacting with objects via abstract references, independent of the concrete implementation.

Why It Matters:
Abstraction simplifies complex systems, enhances code modularity, and improves readability. Developers can work with high-level node types (like `WorkflowNode`) without worrying about the internals of specific node types (e.g., `ConditionNode`, `InferenceNode`, etc.). This separation of concerns promotes scalability and maintainability.

Application in the Project:
In our workflow orchestration system, `WorkflowNode` is an abstract class that defines the essential structure of any node, including abstract methods like `execute()`, `isValid()`, and `validateOperation()`. Subclasses such as `ConditionNode`, `TrainingNode`, or `InferenceNode` provide specific implementations tailored to their functionality.

This allows the system (especially the controller) to interact with different node types in a uniform way—using the abstract reference `WorkflowNode`—thus achieving instance-level abstraction. Meanwhile, the abstract class itself provides the class-level abstraction by defining the blueprint shared across all node types.

Code Example:

Listing 2: Class-level and Instance-level Abstraction

```
public abstract class WorkflowNode {
    public abstract void execute() throws InvalidWorkflowException;
    public abstract boolean isValid() throws InvalidWorkflowException;
}

public class ConditionNode extends WorkflowNode {
    private String conditionExpression;

    @Override
    public void execute() {
        System.out.println("Evaluating: " + conditionExpression);
        // specific logic...
    }

    @Override
    public boolean isValid() {
        return conditionExpression != null && !conditionExpression.isBlank
    }
}
```

Commentary:
This abstraction model enables the system to grow gracefully. For example, adding a new node type like `MonitoringNode` only requires extending `WorkflowNode` and implementing the required methods—without needing to modify the rest of the codebase. It also supports polymorphism, as these subclasses can be treated interchangeably by controllers and workflow managers.

This demonstrates both theoretical and practical abstraction, applied carefully across the system to ensure future extensibility and code clarity.

3.3 Inheritance

Definition:
Inheritance is an object-oriented programming principle that enables one class (the subclass or child) to inherit properties and behaviors (fields and methods) from another class (the superclass or parent). It promotes code reuse and hierarchical relationships between components.

Why It Matters:
Inheritance enables modular, maintainable code by allowing shared behavior to be defined once in a base class and reused across many subclasses. It reduces redundancy and makes it easier to introduce consistent enhancements or bug fixes.

Application in the Project:
In this project, the class `HyperparameterTuningNode` inherits from `ExecutableNode`, which itself extends the abstract base class `WorkflowNode`. This three-level hierarchy showcases deep inheritance for specialization. Common execution logic is encapsulated in the `ExecutableNode` class, while each specific ML node such as `HyperparameterTuningNode` overrides only what it needs to modify.

Code Example:

Listing 3: Inheritance: HyperparameterTuningNode.java

```
public class HyperparameterTuningNode extends ExecutableNode<String> {

    public HyperparameterTuningNode(String id, String name) {
        super(id, name, NodeType.HYPERPARAMETER_TUNING);
    }

    @Override
    public void executeWithContext(Map<String, String> context) {
        System.out.println("Tuning hyperparameters for model...");
        executionLogger.log("Hyperparameter tuning executed.");
    }

    @Override
    public void validateOperation(String operation) throws UnsupportedOper
        if (!"tune".equalsIgnoreCase(operation)) {
            throw new UnsupportedOperationForNodeException("Only 'tune' op
        }
    }

    @Override
    public boolean isValid() {
        return getName() != null && !getName().isEmpty();
    }
}
```

Commentary:
This example illustrates classic inheritance. `HyperparameterTuningNode` inherits all basic execution structure and properties from `ExecutableNode`, which in turn inherits from the base `WorkflowNode`. This layered design promotes flexibility, allowing new node types to be introduced with minimal duplication and clear separation of shared vs. specialized behavior.

3.4 Polymorphism

Polymorphism is a core principle of Object-Oriented Programming (OOP) that allows objects of different types to be treated uniformly through a common interface or parent class. It enhances flexibility and code reusability by enabling dynamic method behavior depending on the object's runtime type or context. In our workflow orchestration system, we implemented four distinct types of polymorphism:

- Subtype Polymorphism (Inclusion)
- Parametric Polymorphism (Generics)
- Ad-hoc Polymorphism (Overloading)
- Coercion Polymorphism (Type Casting)

Each is explained below in detail.

3.4.1 Subtype Polymorphism (Inclusion)

Subtype polymorphism, also known as inclusion polymorphism, allows an object of a subclass to be treated as an object of its superclass. This enables general interfaces to be written, which can operate on objects of multiple types.

Example in the Project:
In `MainViewController`, the method `executeNode(WorkflowNode node)` accepts a base class reference and invokes the appropriate `execute()` method depending on the subclass (e.g., `ConditionNode`, `HyperparameterTuningNode`, etc.). This allows dynamic behavior at runtime.

Listing 4: Subtype Polymorphism via executeNode

```
public void executeNode(WorkflowNode node) {
    node.execute(); // Dynamic dispatch
}
```

This demonstrates subtype polymorphism since the exact method executed depends on the actual subclass type at runtime.

3.4.2 Parametric Polymorphism (Generics)

Parametric polymorphism allows classes and methods to operate on objects of various types while maintaining type safety. This is typically implemented using Java Generics.

Example in the Project:
The class `ExecutableNode<T>` and the logger `GenericExecutionLogger<T>` both use generic type parameters to operate on flexible types. This allows the same logic to work on different data types.

Listing 5: Parametric Polymorphism with Generics

```
public abstract class ExecutableNode<T> extends WorkflowNode {
    protected GenericExecutionLogger<T> executionLogger = new GenericExecut

    public abstract void executeWithContext(Map<String, T> context);
}
```

Generics promote reusability without sacrificing type safety, allowing us to create flexible execution and logging behaviors.

3.4.3 Ad-hoc Polymorphism (Overloading)

Ad-hoc polymorphism occurs when multiple methods share the same name but differ in parameters. Java resolves the correct method at compile time.

Example in the Project:
In the `GenericExecutionLogger` class, the method `logScore` is overloaded to support both `int` and `float` values.

Listing 6: Overloaded logScore Methods

```
public void logScore(int score) {
    System.out.println("Score (int): " + score);
}

public void logScore(float score) {
    System.out.println("Score (float): " + score);
}
```

The compiler chooses the correct method based on the argument type.

3.4.4 Coercion Polymorphism (Type Casting)

Coercion polymorphism refers to automatic or explicit conversion of one data type to another in order to invoke appropriate methods or interface with APIs.

Example in the Project:
When calling the overloaded `logScore` methods with different types (e.g., integer literals or float values), Java may perform type coercion implicitly.

Listing 7: Type Coercion in Action

```
\begin{lstlisting}[language=Java, caption={Type Coercion in Action}]
logger.logScore(10);      // int -> calls logScore(int)
logger.logScore(10.5f);   // float -> calls logScore(float)
```

This subtle but powerful behavior helps bridge overloaded methods in real-world applications.

Together, these four implementations demonstrate the comprehensive application of polymorphism in our system, showcasing flexibility, reusability, and extensibility — key tenets of object-oriented design.

3.5 Modularity

Definition:
Modularity is a core object-oriented design principle that involves organizing a system into independent, self-contained units — or modules — each responsible for a specific aspect of the overall functionality. A modular class encapsulates a well-defined responsibility and interacts with other components via clean, minimal interfaces.

Why It Matters:
Modular systems are easier to understand, test, debug, and extend. Since each module has a single focus, developers can make changes or add new features with minimal impact

on the rest of the system. Modularity also promotes code reuse and supports collaborative development by allowing parallel work on distinct components.

Application in the Project:
Our JavaFX-based Machine Learning Workflow Orchestration System is designed with high modularity in mind. Each class encapsulates a specific concern and remains decoupled from unrelated parts of the system. Key examples include:

- **WorkflowNode (Abstract Base Class):** Serves as a blueprint for all node types in the workflow. It provides shared logic for ID management, timestamps, and connection handling — allowing subclasses to inherit and reuse this functionality without redundancy.
- **ExecutableNode<T>:** A generic abstract extension of `WorkflowNode` that introduces modular execution behavior. The use of generics allows each node to execute in a type-safe way while maintaining a consistent contract. It also integrates logging behavior without binding to a specific node type.
- **Node Subclasses:** Each node type — such as `InferenceNode`, `TrainingNode`, `ConditionNode`, and `HyperparameterTuningNode` — encapsulates its unique logic and structure. These are plug-and-play components, easily added or swapped without altering the system's foundation.
- **WorkflowConnection:** A dedicated module that handles connections (edges) between nodes. This separation of concerns ensures that connection logic (e.g., labels like `"Yes"` or `"No"`) can evolve independently of the node logic.
- **MainViewController:** A modular JavaFX controller that bridges the visual layer with the model. It encapsulates GUI logic such as node rendering, arrow drawing, and workflow execution control. Its isolation from the node logic preserves the MVC architecture and supports better UI maintainability.

Commentary:
By enforcing modularity at all levels — from abstract base classes to concrete UI controllers — our system achieves a clear separation of responsibilities. This structure not only aligns with best practices in object-oriented programming but also simplifies testing, enables future microservice decomposition, and facilitates long-term maintenance. Every node type, connection, and interaction is cleanly bounded in its own class, showcasing true architectural discipline.

3.6 Composition

Definition:
Composition is an object-oriented design principle where one class contains instances of other classes to achieve complex functionality. Unlike inheritance ("is-a"), composition models a "has-a" relationship. This provides flexibility, decouples components, and promotes code reuse.

Why It Matters:
Composition allows behavior to be assembled dynamically at runtime, making the system

more modular and easier to maintain. Changes in one component do not require changes in other parts of the system, as long as interfaces remain consistent.

Application in the Project:
In our JavaFX-based workflow orchestration system, the `WorkflowEventNotifier` class demonstrates composition by holding and managing a list of `WorkflowObserver` objects. This enables loosely coupled notification logic, where observers can be added or removed without altering the core event emitter.

Code Example:

Listing 8: Composition via WorkflowEventNotifier.java

```
public class WorkflowEventNotifier {
    private final List<WorkflowObserver> observers = new ArrayList<>();

    public void addObserver(WorkflowObserver observer) {
        observers.add(observer);
    }

    public void removeObserver(WorkflowObserver observer) {
        observers.remove(observer);
    }

    public void notifyObservers(String event) {
        for (WorkflowObserver observer : observers) {
            observer.onWorkflowEvent(event);
        }
    }
}
```

Listing 9: WorkflowObserver Interface

```
public interface WorkflowObserver {
    void onWorkflowEvent(String event);
}
```

Commentary:
The `WorkflowEventNotifier` class *has-a* relationship with `WorkflowObserver`. Instead of using inheritance, it composes a list of observers and delegates event handling via the `notifyObservers(...)` method. This pattern is a classic use of composition and also aligns with the Observer design pattern. It improves testability, reduces tight coupling, and enables runtime extensibility.

3.7 Information Hiding

Definition:
Information hiding is a core principle of object-oriented programming that involves concealing the internal implementation details of a class and exposing only what is necessary through well-defined interfaces or methods. This reduces system complexity, promotes loose coupling, and prevents unintended interference with internal logic.

Why It Matters:
By hiding sensitive implementation details and exposing only essential operations, we improve system robustness and allow internal changes without affecting external components. It also strengthens security, encapsulation, and modularity across the software design.

Application in the Project:
Our JavaFX-based Workflow Orchestration System implements information hiding at multiple architectural layers. The project clearly separates interfaces from implementations, uses access modifiers strategically, and encapsulates internal logic within service and validation layers.

- `WorkflowValidationService.java`
  This class encapsulates internal validation logic using private fields and methods. For instance, the list of valid node types is private and inaccessible outside the class. Validation checks such as `isValidNode()` and `isValidConnection()` expose only high-level behavior, not how the logic is implemented.

- `WorkflowServiceImpl.java`
  Acts as the main implementation class for workflow orchestration. It keeps its internal graph and helper services (`WorkflowValidationService`, `WorkflowExecutionService`) private. External components can only access the workflow functionality through public interface methods like `addNode()` or `executeWorkflow()`, ensuring all orchestration logic remains hidden.

- `WorkflowService.java`
  This interface defines the contract that the system exposes. It declares the allowed operations (e.g., add/remove/execute nodes) without revealing how they are internally implemented. This promotes loose coupling and makes it easy to substitute or extend functionality.

- `WorkflowExecutionService.java`
  Exposes only necessary execution methods while encapsulating the actual traversal and execution logic. The internal handling of node sequencing, condition evaluation, and result logging is hidden from all consumers.

Commentary:
The project follows a clean architecture with information hiding at its core. By splitting concerns into interfaces, services, and controllers, the system ensures that each component only knows what it needs to. This results in:

- Greater modularity and maintainability.
- Safer updates and refactoring.
- A clearer API for future development.

Listing 10: Information Hiding in WorkflowServiceImpl.java

```
public class WorkflowServiceImpl implements WorkflowService {
```

```
    private final WorkflowGraph workflowGraph;
    private final WorkflowValidationService validationService;
    private final WorkflowExecutionService executionService;

    public WorkflowServiceImpl(...) {
        this.workflowGraph = new WorkflowGraph();
        this.validationService = new WorkflowValidationService();
        this.executionService = new WorkflowExecutionService();
    }

    @Override
    public void addNode(WorkflowNode node) {
        if (validationService.isValidNode(node)) {
            workflowGraph.addNode(node); // Internal logic hidden
        }
    }

    // Internal methods and dependencies are not exposed externally
}
```

Summary:
The system's layered architecture leverages information hiding to abstract complexity and secure internal logic. It ensures that modules only interact through clearly defined, minimal interfaces, aligning well with industry best practices for scalable and maintainable software design.

3.8 Hierarchy

Definition:
Hierarchy in object-oriented programming refers to the structured relationship between classes through inheritance, forming a tree-like structure. It allows for the organization of common functionality at higher levels and specific behaviors at lower levels, promoting reuse and logical code design.

Application in the Project:
In our JavaFX-based workflow orchestration system, a clear and well-structured class hierarchy is implemented across all node types. The design starts with a highly abstract base class and progresses through increasingly specialized classes:

- `WorkflowNode` *(Abstract base class)*: Defines shared properties and abstract methods like `execute()` and `isValid()` for all node types.
- `ExecutableNode<T>` *(Generic abstract subclass)*: Adds parameterized execution behavior and logging logic applicable to all executable nodes.
- Concrete subclasses such as:
  - `TrainingNode`
  - `InferenceNode`

- `HyperparameterTuningNode`
- `ClusteringNode`
- `ConditionNode`

These classes implement or override logic to define their specialized behavior.

This vertical inheritance structure demonstrates a classical object-oriented hierarchy, enabling code reuse, extensibility, and consistency across all workflow nodes.

Illustration:

Listing 11: Simplified Class Hierarchy

```
abstract class WorkflowNode {
    public abstract void execute();
}

abstract class ExecutableNode<T> extends WorkflowNode {
    public void logExecution(T data) { ... }
}

class TrainingNode extends ExecutableNode<String> {
    public void execute() { ... }
}
```

Commentary:
This hierarchy not only improves readability and organization but also makes the system easily extensible. Introducing a new node type only requires creating a new subclass that inherits from the existing abstract base, ensuring rapid development and architectural consistency.

3.9 Reusability

Definition:
Reusability is a fundamental object-oriented programming principle that encourages the development of modular and general-purpose components. These components can be used across different parts of a system — or even in entirely different systems — without rewriting code from scratch. Reusability enhances maintainability, reduces redundancy, and accelerates development.

Why It Matters:
Reusable components are easier to maintain and test, and they promote consistency across the software architecture. They minimize duplication and allow developers to focus on implementing new functionality rather than reinventing existing logic.

Application in the Project:
Our JavaFX-based Machine Learning Workflow Orchestration System demonstrates reusability through several key classes that encapsulate logic designed to be used repeatedly:

- `Arrow.java`: A visual representation of edges between nodes. This class is reused universally to connect different types of nodes with consistent behavior and styling.

- `MainViewController.java`: Centralized logic for node execution, connection handling, and UI updates. Its utility methods are reused throughout the system.
- `NodeCreationDialog.java`: A reusable dialog window invoked every time a new node is added, regardless of its type. The dialog adapts dynamically using common input fields and handlers.
- `WorkflowNode` and `ExecutableNode<T>`: Abstract base classes reused to build a wide variety of specialized node types with shared behaviors (metadata handling, connections, execution, logging).

Code Example:

Listing 12: Reusable Visual Arrow Component

```
public class Arrow extends Group {
    private final Line line = new Line();
    private final Polygon arrowHead = new Polygon();

    public Arrow(double startX, double startY, double endX, double endY) {
        line.setStartX(startX);
        line.setStartY(startY);
        line.setEndX(endX);
        line.setEndY(endY);
        updateArrowHead();
        getChildren().addAll(line, arrowHead);
    }

    public void updatePosition(double startX, double startY, double endX, d
        line.setStartX(startX);
        line.setStartY(startY);
        line.setEndX(endX);
        line.setEndY(endY);
        updateArrowHead();
    }

    private void updateArrowHead() {
        // Math to reposition the arrowhead polygon
    }
}
```

Commentary:

The `Arrow` class is a prime example of reusability in the visual layer. It encapsulates all behavior related to drawing directional connections between nodes and can be instantiated with any pair of coordinates. This allows arrows to be reused throughout the workflow UI, whether connecting training nodes, decision branches, or analysis steps, without duplicating arrow logic. Similar design decisions apply to shared controllers, dialogs, and base node structures.

4 Additional OOP Design Principles

While the previous section addressed the fundamental Object-Oriented Programming (OOP) principles as required, this section extends beyond the essentials to include additional OOP design concepts that were purposefully implemented in the project.

Rationale:
These advanced principles were incorporated not out of necessity, but from a desire to elevate the software's architecture, ensure future scalability, and deepen personal understanding of advanced OOP methodologies. Since this system is intended to evolve into a more complex orchestration framework, incorporating deeper principles like cohesion, coupling, dependency injection, and others was a strategic choice aimed at fostering long-term maintainability and professional growth.

4.1 Cohesion and Coupling

Definition:
Cohesion refers to the degree to which the elements within a class belong together and serve a single, well-defined purpose. High cohesion implies that a class is focused solely on one task or responsibility. Coupling, conversely, measures the interdependence between software components. Loose (or low) coupling ensures that modules can evolve, be reused, or be tested independently without affecting others.

Why It Matters:
High cohesion improves code readability, testability, and maintenance by clearly organizing logic around consistent behaviors. Loose coupling enables flexibility, easier debugging, and reduces unintended ripple effects when making changes—critical for growing and scalable systems.

Application in the Project:
Two classes, `ExecutableNode<T>` and `GenericExecutionLogger<T>`, illustrate a well-balanced application of these principles:

- **High Cohesion:** `ExecutableNode<T>` defines all execution-related behavior for node types like `TrainingNode` and `ConditionNode`, keeping execution, validation, and context-based processing closely grouped.
- **Loose Coupling:** Logging functionality is encapsulated in `GenericExecutionLogger<T>`, an independent utility that can be swapped, reused, or updated without modifying the node logic. This decouples the logging concern from the execution concern.

Code Example:

Listing 13: Loose Coupling with High Cohesion

```
public abstract class ExecutableNode<T> extends WorkflowNode {

    protected final GenericExecutionLogger<T> executionLogger = new Generic

    public void executeWithContext(Map<String, String> context) {
```

```
        System.out.println("Executing node: " + getName());
        T result = performExecutionLogic(context);
        executionLogger.log(result);
    }

    protected abstract T performExecutionLogic(Map<String, String> context)
}
```

Commentary:
In this design, the execution of workflow nodes and the logging of results are separated into distinct modules. Execution nodes handle core logic, while the logger manages result reporting. This structured separation promotes modularity, simplifies testing, and ensures that classes remain small, maintainable, and well-aligned with single responsibilities.

4.2 Dependency Injection

Dependency Injection (DI) is a design pattern in which a class receives the objects it depends on (its *dependencies*) rather than creating them internally. This allows for greater modularity, better separation of concerns, easier testing, and enhanced maintainability.

In our workflow orchestration system, **dependency injection** is primarily demonstrated through the interaction between the GUI controller and service logic.

Example: `MainViewController`

The `MainViewController` class coordinates user interactions and workflow execution logic, but it does not contain core business logic itself. Instead, it delegates workflow operations to the service layer:

- `WorkflowService` and `WorkflowExecutionService` are injected and used as external dependencies.
- This design ensures that the controller focuses purely on view-related tasks and invokes well-defined service methods.

Injected Services:

- `WorkflowService`: Handles node creation, deletion, and persistence.
- `WorkflowExecutionService`: Manages the execution of workflows and branching logic.

Code Snippet: `MainViewController.java`

Listing 14: Dependency Injection in MainViewController

```
public class MainViewController {

    private final WorkflowService workflowService;
    private final WorkflowExecutionService executionService;

    public MainViewController() {
```

```
        this.workflowService = new WorkflowServiceImpl(); // Injected depen
        this.executionService = new WorkflowExecutionService(); // Injected
    }

    public void executeWorkflow() {
        List<WorkflowNode> startNodes = workflowService.getStartNodes();
        for (WorkflowNode node : startNodes) {
            executionService.executeNode(node);
        }
    }

}
```

Why Dependency Injection Matters Here:

- **Separation of Concerns**: UI logic is kept separate from business rules.
- **Improved Testability**: You can easily mock or substitute `WorkflowService` in unit tests.
- **Modular Design**: The architecture supports future extensions with minimal coupling.

Conclusion: By injecting dependencies into `MainViewController`, we ensure that each class in the system has a single responsibility and interacts with other components through clean, well-defined interfaces. This promotes maintainability and flexibility as the system evolves.

4.3 Aggregation

Definition:
Aggregation is a structural relationship in Object-Oriented Programming where one class contains or references another, but without taking full ownership of its lifecycle. This "has-a" relationship implies that the composed object can exist independently of the container. Unlike composition, aggregation allows the referenced object to be shared or reused elsewhere.

Why It Matters:
Aggregation promotes loose coupling and modularity. By referencing external classes without full ownership, components can work together without being tightly interdependent. This enhances reusability, separation of concerns, and testability.

Application in the Project:
One of the clearest demonstrations of aggregation in our JavaFX-based Workflow Orchestration System is within the `ExecutableNode<T>` class. It contains a reference to a logging utility:

Listing 15: Aggregation: ExecutableNode with Logger

```
protected final GenericExecutionLogger<T> executionLogger = new GenericExec
```

Here, the `GenericExecutionLogger` is a separate utility class responsible for logging execution behavior. It is used by various node types but is not tightly bound to the lifecycle of any single node. The logger can exist independently and may be reused elsewhere in the application.

Another strong example is in the `MetadataPrinter<T>` class. It is used by different components across the system to format and print metadata values. The printer does not own the metadata objects it processes — it merely operates on them externally, further illustrating an aggregation relationship.

Listing 16: Aggregation: MetadataPrinter with Generic Metadata

```
public class MetadataPrinter<T extends Map<?, ?>> {

    public void printMetadata(T metadata) {
        metadata.forEach((k, v) -> System.out.println(k + ": " + v));
    }
}
```

Summary:
These examples showcase how our system favors flexible and decoupled design. By employing aggregation, components like loggers and printers can be injected, reused, and extended without introducing tight dependencies or redundant duplication. This aligns with best practices for building scalable, modular architectures.

4.4 The SOLID Principles

Definition:
The SOLID principles are a set of five foundational object-oriented design guidelines introduced by Robert C. Martin. They aim to make software more understandable, flexible, and maintainable. In our project, we consciously applied these principles to promote clean code, minimize coupling, and enhance future extensibility.

List of SOLID Principles:

- **S** – Single Responsibility Principle (SRP)
- **O** – Open/Closed Principle (OCP)
- **L** – Liskov Substitution Principle (LSP)
- **I** – Interface Segregation Principle (ISP)
- **D** – Dependency Inversion Principle (DIP)

In the following subsections, we will demonstrate how each principle is represented within our JavaFX-based workflow orchestration system.

4.4.1 Single Responsibility Principle (SRP)

Definition:
The Single Responsibility Principle states that a class should have only one reason to change. This means that each class should encapsulate only one responsibility or functionality.

Why It Matters:
Applying SRP improves maintainability and testability. When responsibilities are clearly separated, changes to one part of the system do not unintentionally affect unrelated functionality.

Application in the Project:
The `NodeFactory` class in our system demonstrates SRP effectively. It is solely responsible for instantiating node objects based on the `NodeType`. It does not manage rendering, logic execution, or UI behavior. This separation allows us to extend node creation logic independently from the rest of the system.

Code Example:

Listing 17: NodeFactory.java — Single Responsibility in Action

```
public class NodeFactory {
    public static WorkflowNode createNode(NodeType type, String id, String
        return switch (type) {
            case TRAINING -> new TrainingNode(id, name);
            case INFERENCE -> new InferenceNode(id, name);
            case CONDITION -> new ConditionNode(id, name);
            case HYPERPARAMETER_TUNING -> new HyperparameterTuningNode(id,
            // ... other node types
            default -> throw new IllegalArgumentException("Unknown node typ
        };
    }
}
```

Commentary:
This class has a clear and isolated responsibility: node creation. If we later want to add new node types or validation during instantiation, we can update this file without impacting the rest of the application. This decoupling is a direct benefit of adhering to SRP.

4.4.2 Open/Closed Principle (OCP)

Definition:
The Open/Closed Principle states that software entities (classes, modules, functions) should be *open for extension* but *closed for modification*. This means you can add new behavior without altering existing code, which helps avoid introducing bugs into working systems.

Why It Matters:
OCP allows developers to extend systems safely by building on existing, tested compo-

nents. Instead of rewriting or modifying core classes, we introduce new subclasses or override behavior to support new features or logic.

Application in the Project:
The architecture of our workflow system strongly reflects OCP. At the heart is the `WorkflowNode` abstract base class, which defines the core structure and behavior of all nodes. New functionality is added via concrete subclasses (like `TrainingNode`, `InferenceNode`, `ClusteringNode`, etc.) without modifying the original base class.

Similarly, `ExecutableNode<T>` extends `WorkflowNode` and introduces generic execution capabilities that are reused and extended by various subclasses. Each of these node classes adds its own behavior while preserving and leveraging the core features defined in the base hierarchy.

Code Example:

Listing 18: Open/Closed Principle with ExecutableNode and Subclasses

```
public abstract class ExecutableNode<T> extends WorkflowNode {
    protected GenericExecutionLogger<T> executionLogger = new GenericExecut

    public ExecutableNode(String id, String name, NodeType type) {
        super(id, name, type);
    }

    public abstract void executeWithContext(T context);
}

public class TrainingNode extends ExecutableNode<Map<String, String>> {
    public TrainingNode(String id, String name) {
        super(id, name, NodeType.TRAINING);
    }

    @Override
    public void executeWithContext(Map<String, String> context) {
        System.out.println("Training model with context: " + context);
        executionLogger.log("Training completed.");
    }
}
```

Commentary:
The abstract class `ExecutableNode<T>` is designed to be extended. New node types add execution logic by overriding the `executeWithContext()` method. This design avoids altering existing logic and supports future extensibility without code duplication — a direct application of the Open/Closed Principle.

4.4.3 Liskov Substitution Principle (LSP)

Definition:
The Liskov Substitution Principle states that *subtypes must be substitutable for their base types without altering the correctness of the program.* In simpler terms, objects of a

subclass should be usable wherever their superclass is expected, without causing incorrect behavior.

Why It Matters:
LSP ensures that inheritance does not break the program's expected behavior. It enables developers to rely on abstractions while allowing subclasses to provide specific implementations. This principle supports robustness and predictable code.

Application in the Project:
In our system, we consistently substitute subclasses like `TrainingNode`, `InferenceNode`, and `ConditionNode` wherever the abstract type `WorkflowNode` or `ExecutableNode<T>` is expected. For example, the `MainViewController`'s 'executeNode(WorkflowNode node)' method is capable of executing any concrete node type due to adherence to LSP.

The `WorkflowNode` class defines abstract methods like `execute()` and `isValid()`, which are implemented in each subclass. This allows polymorphic behavior without breaking execution or validation flows.

Code Example:

Listing 19: LSP: Substituting ExecutableNode Subclasses in MainViewController

```
public void executeNode(WorkflowNode node) {
    try {
        node.validateOrThrow(); // Polymorphic call
        node.execute();         // Polymorphic call
        logExecution(node);
    } catch (InvalidWorkflowException e) {
        System.err.println("Execution failed: " + e.getMessage());
    }
}
```

Commentary:
This method operates generically on the `WorkflowNode` base type but safely executes any subclass instance. Because subclasses conform to the interface and behavioral expectations of the base class, LSP is upheld. This demonstrates strong design conformance with the principle and enables extension without fragile code.

4.4.4 Interface Segregation Principle (ISP)

Definition:
The Interface Segregation Principle (ISP) states that *clients should not be forced to depend on interfaces they do not use.* In other words, it is better to have multiple, smaller, and more specific interfaces than one large general-purpose interface.

Why It Matters:
When classes implement bloated interfaces with methods they don't need, this leads to unnecessary coupling, confusion, and brittle implementations. ISP promotes separation of concerns, simplifies testing, and improves maintainability by making interfaces focused and relevant.

Application in the Project:

Our system uses well-designed, narrow interfaces to adhere to ISP. For instance, the `Describable` interface includes only one method — `getDescription()` — which is specifically relevant only to node types that offer human-readable summaries. It is not bundled into larger base classes like `WorkflowNode`, ensuring that unrelated classes are not forced to implement unused behavior.

Additionally, the separation of controller, factory, execution, and model logic into modular packages ensures that GUI classes are not tightly coupled with node logic, thanks to clearly separated interfaces and responsibilities.

Code Example:

Listing 20: Interface Segregation via the Describable Interface

```
public interface Describable {
    String getDescription();
}

public class InferenceNode extends ExecutableNode<String> implements Descri
    private String description;

    @Override
    public String getDescription() {
        return description;
    }
}
```

Commentary:
This approach follows ISP by applying only the interfaces needed. For example, only nodes that offer descriptions implement `Describable`, and others are not burdened with irrelevant contracts. This minimizes boilerplate, improves cohesion, and keeps the class responsibilities aligned with their real purpose.

4.4.5 Dependency Inversion Principle (DIP)

Definition:
The **Dependency Inversion Principle** states that:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

This principle promotes flexibility and decoupling by inverting traditional dependency directions, especially between business logic and infrastructure components.

Why It Matters:
DIP enables better scalability, unit testing, and maintainability. By injecting interfaces rather than instantiating implementations directly, systems become more modular and easier to update or replace.

Application in the Project:
Our `MainViewController` relies on the abstract `WorkflowService` interface instead of

the concrete `WorkflowServiceImpl`. This abstraction allows the controller to interact with different backend services without needing to know implementation details.

Listing 21: DIP in Action: Controller Depends on Abstraction

```
public class MainViewController {

    private final WorkflowService workflowService;

    public MainViewController(WorkflowService workflowService) {
        this.workflowService = workflowService;
    }

    public void handleSave() {
        workflowService.saveCurrentWorkflow();
    }
}
```

Commentary:
This example demonstrates clear adherence to the Dependency Inversion Principle. The controller works with the `WorkflowService` interface, enabling different implementations (e.g., `WorkflowServiceImpl`, test mocks) to be injected at runtime. This abstraction layer keeps the UI logic clean and decoupled from storage mechanisms, improving testability and long-term maintainability.

Conclusion of SOLID Principles

Our system was designed not only to satisfy foundational object-oriented programming principles, but also to adhere to the more advanced **SOLID principles** of software design. These five principles — Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion — have guided the architecture and implementation of our system's core components.

By carefully structuring nodes, controllers, and services with these principles in mind, we achieved a solution that is modular, testable, extensible, and maintainable. This was not only a design goal but also a learning objective, reflecting a deeper understanding of professional-grade software development practices.

4.5 Design Patterns

Introduction:
In addition to core Object-Oriented Programming principles and SOLID design guidelines, our system demonstrates the practical use of several well-established **object-oriented design patterns**. These patterns offer reusable solutions to common architectural challenges, enhancing the flexibility, maintainability, and testability of the software.

The following subsections outline the key design patterns implemented in the Workflow Orchestration System:

- **Factory Pattern** – Centralized creation of workflow nodes based on type.
- **Observer Pattern** – Decouples event generation and handling through notification mechanisms.
- **Command Pattern** – Encapsulates user actions such as connect, delete, and undo into objects.

Each of these patterns plays a specific role in ensuring the system's robustness, scalability, and adherence to object-oriented best practices. In the next subsections, we detail the usage and implementation of each pattern with examples from the actual codebase.

4.5.1 Factory Pattern

Definition:
The **Factory Pattern** is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses or specialized classes to alter the type of objects that will be created. It abstracts the instantiation process and delegates it to a factory method, enabling greater flexibility and encapsulation.

Why It Matters:
By centralizing object creation logic, the factory pattern reduces coupling and adheres to the Open/Closed Principle. This ensures that new node types can be added without modifying existing logic.

Application in the Project:
In our Workflow Orchestration System, the `NodeFactory` class implements the Factory Pattern to generate various types of `WorkflowNode` objects. Depending on the `NodeType` provided, the factory dynamically instantiates the appropriate subclass (e.g., `TrainingNode`, `ConditionNode`, `InferenceNode`) without exposing the creation logic to the client code.

Code Example:

Listing 22: Factory Pattern Implementation in NodeFactory.java

```
public class NodeFactory {

    public static WorkflowNode createNode(NodeType type, String id, String
        switch (type) {
            case TRAINING:
                return new TrainingNode(id, name);
            case CONDITION:
                return new ConditionNode(id, name);
            case INFERENCE:
                return new InferenceNode(id, name);
            case EVALUATION:
                return new EvaluationNode(id, name);
            // More node types omitted for brevity...
            default:
                throw new IllegalArgumentException("Unsupported node type:
```

```
        }
    }
}
```

Commentary:
Instead of using multiple `new` statements throughout the codebase, all node instantiations are funneled through the `NodeFactory`, promoting maintainability and ensuring consistent object initialization. When new node types are introduced, they are integrated by extending the factory logic without changing other parts of the system.

4.5.2 Observer Pattern

Definition:
The **Observer Pattern** is a behavioral design pattern where an object (called the *subject*) maintains a list of dependents (called *observers*) and notifies them automatically of any state changes, usually by calling one of their methods. This pattern is widely used to implement distributed event-handling systems.

Why It Matters:
The observer pattern promotes loose coupling between objects that produce events and those that respond to them. It allows multiple parts of the system to remain updated in response to changes, without the need for tight integration or hard-coded references.

Application in the Project:
Our system implements the Observer Pattern using two classes: `WorkflowObserver` (interface) and `WorkflowEventNotifier` (concrete subject). These classes decouple event generation from event consumption. For example, when a node is created or a connection is made, the notifier alerts any registered observers (such as the GUI or loggers).

Code Example:

Listing 23: Observer Pattern Implementation

```
public interface WorkflowObserver {
    void onWorkflowChanged(String eventMessage);
}

public class WorkflowEventNotifier {
    private final List<WorkflowObserver> observers = new ArrayList<>();

    public void addObserver(WorkflowObserver observer) {
        observers.add(observer);
    }

    public void notifyObservers(String eventMessage) {
        for (WorkflowObserver observer : observers) {
            observer.onWorkflowChanged(eventMessage);
        }
    }
}
```

Commentary:
This implementation allows external components to register themselves as listeners to workflow events. The `MainViewController` and other parts of the system respond to these events without being tightly coupled to the event generator, which enhances modularity and scalability.

4.5.3 Command Design Pattern

The **Command Design Pattern** is a behavioral pattern that encapsulates a request as an object, allowing us to parameterize clients with queues, logs, and support undoable operations. This pattern decouples the class that invokes the operation from the one that knows how to perform it.

Context in Our System: The command pattern is implemented to encapsulate user actions, such as connecting nodes, as discrete objects, allowing undo / redo functionality and modular control logic. This structure improves extensibility and maintains a clear separation of concerns.

Participating Classes:

- `WorkflowCommand` – An interface defining the `execute()` and `undo()` contract.
- `ConnectNodesCommand` – A concrete implementation of `WorkflowCommand` that connects two nodes and supports undoing the connection.
- `WorkflowInvoker` – Responsible for executing commands and storing them in undo/redo stacks.

Code Example:

Listing 24: ConnectNodesCommand.java (Simplified)

```
public class ConnectNodesCommand implements WorkflowCommand {
    private WorkflowNode source;
    private WorkflowNode target;

    public ConnectNodesCommand(WorkflowNode source, WorkflowNode target) {
        this.source = source;
        this.target = target;
    }

    @Override
    public void execute() {
        source.addConnection(target);
    }

    @Override
    public void undo() {
        source.removeConnection(target);
    }
}
```

Why It's Useful: This modular structure enhances the maintainability and scalability of the system. Developers can easily add new commands (e.g., `DeleteNodeCommand`, `MoveNodeCommand`) without modifying the invoker logic. It also supports complex interaction flows, such as undo/redo sequences, which are essential for a graphical orchestration tool.

Conclusion: By using the Command Pattern, our workflow system introduces a flexible and reusable action-handling mechanism that aligns well with modern software design best practices. It reinforces our project's commitment to clean architecture and user-oriented features.

5 UML Diagrams

This section presents a series of Unified Modeling Language (UML) diagrams that illustrate the structural design of the system. Each package in the project is represented by a dedicated UML class diagram, highlighting key object-oriented relationships such as inheritance, abstraction, interfaces, and dependencies. These diagrams aim to provide a clear visualization of the software architecture and how its components collaborate.

5.1 UML: `command` Package

The `command` package implements the **Command Design Pattern**, encapsulating user actions such as node creation, connection, deletion, and execution. This structure decouples the action execution logic from the components that trigger them, while also supporting undo/redo behavior.

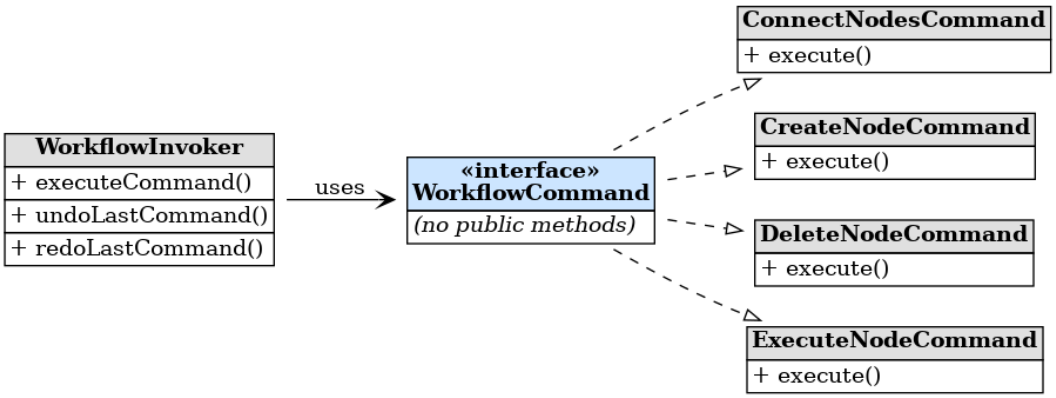

Figure 2: UML Diagram of the `command` Package (Command Pattern)

In this UML:

- `WorkflowCommand` is an interface that defines the standard `execute()` contract.
- Classes like `ConnectNodesCommand`, `CreateNodeCommand`, `DeleteNodeCommand`, and `ExecuteNodeCommand` are concrete implementations of this interface.
- `WorkflowInvoker` is responsible for invoking these commands and managing undo/redo stacks.

This design allows for highly modular, extensible, and testable command logic. It is a textbook implementation of the Command Pattern, tailored for GUI-driven workflow orchestration.

5.2 UML: `controller` Package

The `controller` package handles the coordination between the backend logic and the visual interface of the workflow system. It contains two primary classes:

- `MainController`: Manages core operations such as creating nodes, connecting or disconnecting nodes, executing workflows, and managing service access.
- `MainViewController`: Controls the JavaFX GUI, updates the canvas, handles visual interaction (e.g., node drag, zoom, execution), and invokes backend logic via `MainController`.

This package establishes a clean separation of concerns between system-level logic (in `MainController`) and view-level behavior (in `MainViewController`). The two classes reference each other bi-directionally using setter methods, enabling coordination without tight coupling.

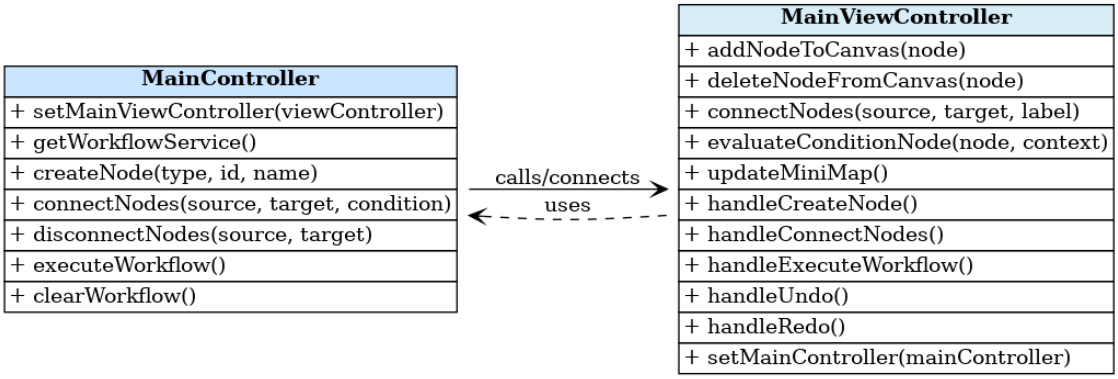

Figure 3: UML Diagram of the `controller` Package

As shown above:

- `MainController` exposes backend methods like `createNode()`, `connectNodes()`, and `executeWorkflow()`.
- `MainViewController` handles user interactions and GUI events through handlers like `handleCreateNode()` and `handleUndo()`, and delegates actions to the `MainController`.

This interaction ensures a high degree of modularity and maintainability in the overall system design.

5.3 UML: `exception` Package

The `exception` package defines a modular and extendable hierarchy of custom exception classes tailored for the workflow orchestration system. These exceptions are designed to improve error handling clarity and provide domain-specific messages when invalid actions occur in the system.

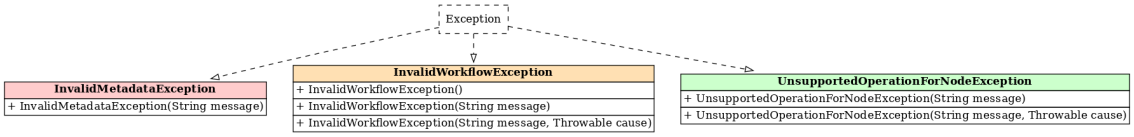

Figure 4: UML Diagram of the `exception` Package

As illustrated in the UML diagram:

- `InvalidWorkflowException` serves as the **base custom exception**, extending Java's built-in `Exception` class. It provides multiple constructors to support detailed error reporting.
- `InvalidMetadataException` extends `InvalidWorkflowException` and is thrown when metadata parsing or assignment fails due to invalid input or schema mismatch.
- `UnsupportedOperationForNodeException` is another subclass that represents situations where a specific node type does not support a requested operation.

This exception structure promotes:

- **Hierarchical clarity:** Each exception inherits from a logical base, allowing grouped catch blocks and improved maintainability.
- **Descriptive error messages:** Each constructor allows customized messaging and causal tracing.
- **Modularity:** New exception types can be added with minimal disruption to existing code.

This approach ensures robust exception handling and aligns with the **Open/Closed Principle** and good software design practices.

5.4 UML: `execution` Package

The `execution` package contains core logic for logging execution outcomes in a generic and reusable way. The central class, `GenericExecutionLogger<T>`, is designed to support flexible logging across different node types and execution contexts using generics.

GenericExecutionLogger<T>	
OOP Concepts:	
✔ Encapsulation	
✔ Parametric Polymorphism (Generics)	
✔ Ad-hoc Polymorphism (Overloading)	
✔ Coercion Polymorphism	
✔ High Cohesion	
✔ Reusability	
Key Fields	
- results: List<T>	
Important Methods	
+ logResult(T)	Add result
+ getResults()	Return all results
+ clear()	Clear log
+ isEmpty()	Check empty
+ size()	Count results
+ log(T)	Basic log
+ logString(String)	Log message
+ logString(String, boolean)	Log w/ timestamp
+ logWithTag(T, String)	Log with tag
+ logScore(int/float/double)	Overloaded/coerced
+ logScore(String)	String parsed score

Figure 5: UML Diagram of the `GenericExecutionLogger<T>` Class in the `execution` Package

This component embodies a rich blend of OOP concepts:

- **Encapsulation:** The internal result list is private and only modifiable via public methods.
- **Parametric Polymorphism (Generics):** The logger is generic, allowing it to log any data type `T`.
- **Ad-hoc Polymorphism (Overloading):** Multiple versions of `logString()` and `logScore()` are defined for flexibility.
- **Coercion Polymorphism:** `logScore()` handles inputs of different numeric types (e.g., `int`, `float`, `double`).
- **High Cohesion:** The class maintains a single responsibility — execution logging — with focused methods.
- **Reusability:** It is used by multiple node types (e.g., `TrainingNode`, `InferenceNode`, `ConditionNode`) for execution tracking.

Key Fields and Methods:

- `results: List<T>` — Internal storage for execution results.

- `logResult(T)`, `getResults()`, `clear()`, `size()`, etc. — Manage results.
- Overloaded `logString(...)` and `logScore(...)` — Provide detailed, typed logging with optional metadata.

This design supports extensibility and flexible diagnostics, contributing significantly to the modularity and testability of the entire system.

5.5 UML: `factory` Package

The `factory` package implements a centralized object creation mechanism using the **Factory Design Pattern**. The core class, `NodeFactory`, is responsible for instantiating various node types used throughout the workflow orchestration system. This design encapsulates the object creation logic and promotes abstraction and loose coupling between components.

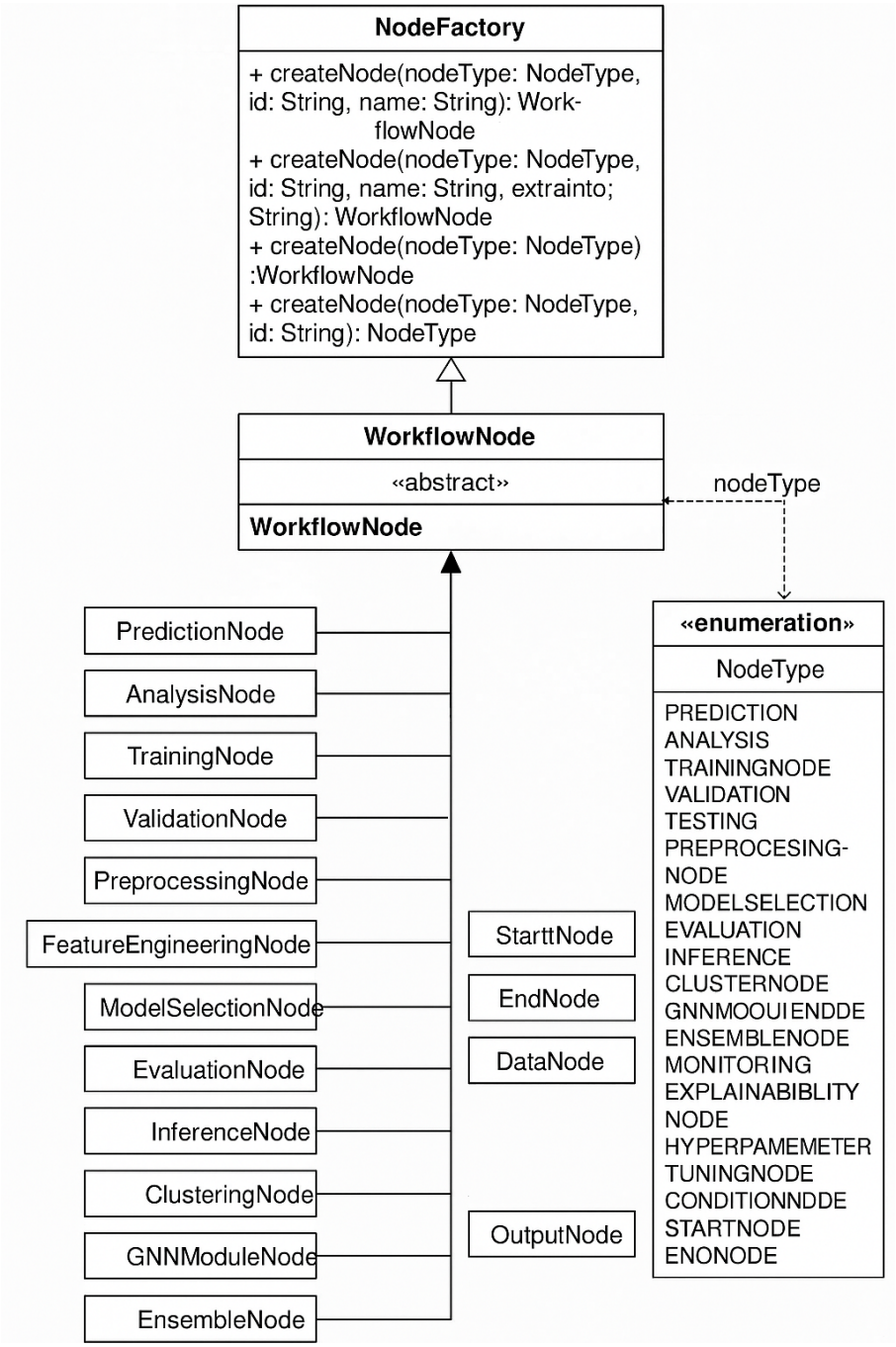

Figure 6: UML Diagram of the `NodeFactory` in the `factory` Package

The UML diagram highlights the static nature and responsibilities of `NodeFactory`, which supports the creation of multiple specialized node types such as:

- `TaskNode`, `ConditionNode`, `PredictionNode`
- `AnalysisNode`, `StartNode`, `EndNode`
- `DataNode`, `OutputNode`

Object-Oriented Principles Demonstrated:

- **Factory Pattern:** Centralizes object creation logic into a single class.
- **Encapsulation:** Hides node instantiation logic inside well-defined static methods.
- **Abstraction:** Client classes interact with the factory to obtain `WorkflowNode` instances, without knowing the specific subclass.
- **Ad-hoc Polymorphism:** The `createNode(...)` method is overloaded to support creation with or without extra validation data.
- **Loose Coupling:** Consumers depend on the factory interface, not the concrete node implementations.

Key Methods:

- `createNode(...)` — Instantiates a generic `WorkflowNode` subtype based on type ID.
- `createNode(..., extraInfo)` — Creates a node with additional validation or configuration data.

This approach simplifies object management across the system, reduces redundancy, and improves maintainability as new node types are introduced.

5.6 UML: `model` Package

The `model` package represents the heart of the system's domain logic. It defines the core building blocks—nodes, connections, groups, and interfaces—that drive the entire orchestration workflow. This package is carefully structured to model a graph-based workflow with rich semantics and extensibility.

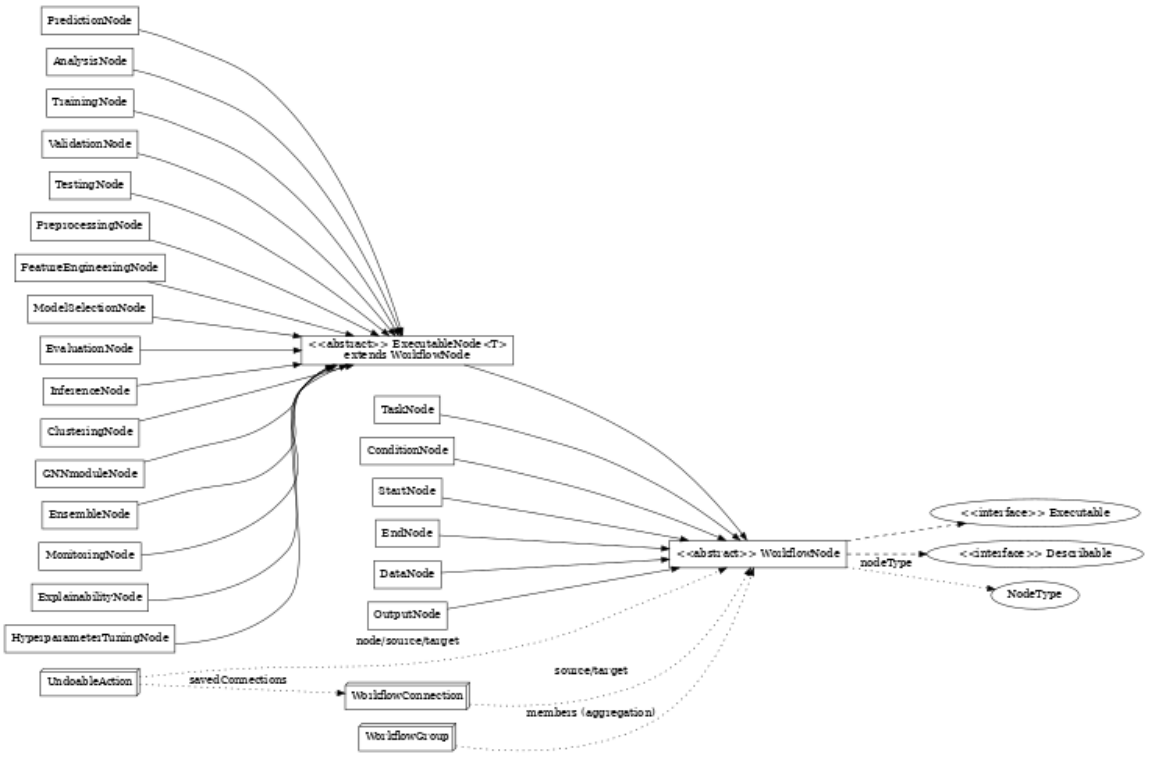

Figure 7: UML Diagram of the `model` Package

Core Architecture:

- `WorkflowNode` is an abstract base class representing any node in the workflow.
- It implements the `Describable` and `Executable` interfaces, enforcing polymorphism and contract-based behavior.
- Each node maintains its `NodeType`, list of outgoing `WorkflowConnection`s, and metadata.

Inheritance and Specialization:

- The class `ExecutableNode<T>` is an abstract generic subclass of `WorkflowNode`, designed to encapsulate common logic for nodes that require parameterized execution (e.g., training or prediction results).
- Over 15 concrete subclasses extend either `WorkflowNode` or `ExecutableNode<T>`, including:
  - ML-related: `PredictionNode`, `TrainingNode`, `EvaluationNode`, `InferenceNode`, etc.
  - Control-flow: `TaskNode`, `ConditionNode`, `StartNode`, `EndNode`
  - Data-centric: `DataNode`, `OutputNode`
- These subclasses override node-specific logic while benefiting from reusable functionality.

Structural Associations:

- `WorkflowConnection` models directed edges between nodes. Each connection maintains `source` and `target` references.
- `WorkflowGroup` aggregates multiple `WorkflowNode`s as members—illustrating the aggregation relationship.
- `UndoableAction` interacts with connections and nodes to support undo/redo functionality.

Object-Oriented Principles Reflected:

- **Abstraction:** Abstract base classes define shared behavior without enforcing implementation.
- **Inheritance:** Dozens of node types extend reusable base classes with specific behavior.
- **Parametric Polymorphism:** `ExecutableNode<T>` allows for flexible typed execution results.
- **Inclusion Polymorphism:** All node instances are treated as `WorkflowNode`s via dynamic binding.

- **Composition/Aggregation:** Nodes are composed into graphs via connections and groups.
- **Encapsulation:** Metadata, connections, and internal state are protected behind accessor methods.

Design Significance:
This layered and extensible design allows new node types to be added with minimal disruption. It supports reuse, modularity, and long-term scalability of the orchestration system across evolving ML pipelines.

5.7 UML: `observer` Package

The `observer` package implements the classic **Observer Design Pattern**, enabling a decoupled mechanism for broadcasting workflow-related events to interested listeners. This pattern is crucial in GUI-driven and reactive systems, as it cleanly separates event generation from event handling logic.

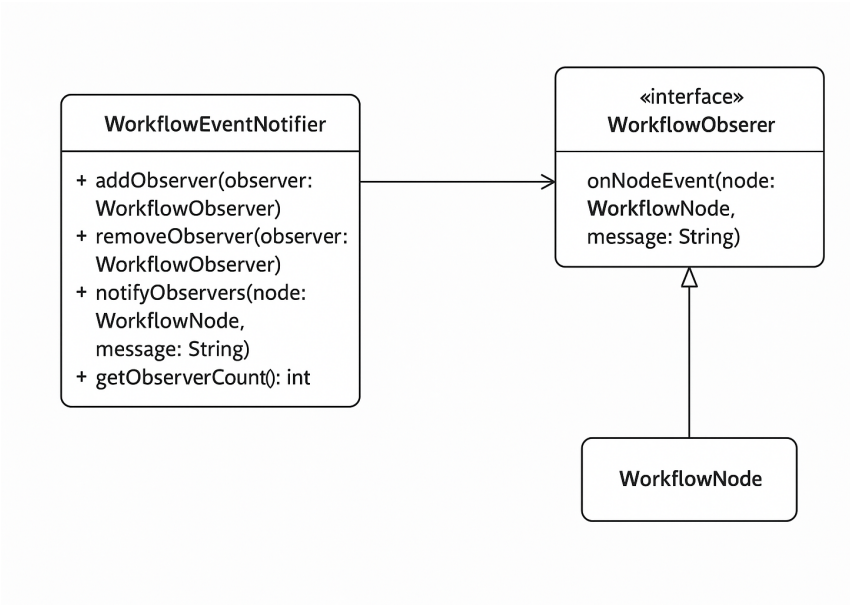

Figure 8: UML Diagram of the `observer` Package

Key Components:

- `WorkflowObserver` is an **interface** that defines a single method: `onNodeEvent(node, message)`. This method is called whenever a workflow event is triggered.
- `WorkflowEventNotifier` is the **subject** or **publisher** in the observer pattern. It manages a list of registered observers and notifies them of relevant events.
- The `WorkflowNode` class is used as the event context, passed along with a custom string message to provide detailed information during notification.

Workflow Example:

- Observers (e.g., GUI components, logs, debugging tools) register themselves using `addObserver(...)`.
- When a node is executed or updated, `notifyObservers(...)` is called with the node and a custom message.
- All registered observers receive the event, without the notifier knowing who they are or how they handle it.

Object-Oriented Principles Demonstrated:

- **Abstraction:** Event handling is abstracted through the `WorkflowObserver` interface.
- **Encapsulation:** The list of observers is private to the notifier and accessed only through well-defined methods.
- **Loose Coupling:** `WorkflowEventNotifier` and `WorkflowObserver` are loosely connected through interface-based communication.
- **Polymorphism:** Multiple observer implementations can handle the same event in different ways.

This package enhances modularity and reusability across the system. It enables external modules (such as controllers or visual components) to react to workflow state changes without modifying the core logic. It is a clean, extensible application of the Observer pattern, aligned with OOP best practices.

5.8 UML: `service` Package

The `service` package encapsulates the backend services that manage the core workflow data and operations. These services provide an abstraction layer between the application logic and the underlying data model, ensuring modularity, separation of concerns, and ease of maintenance.

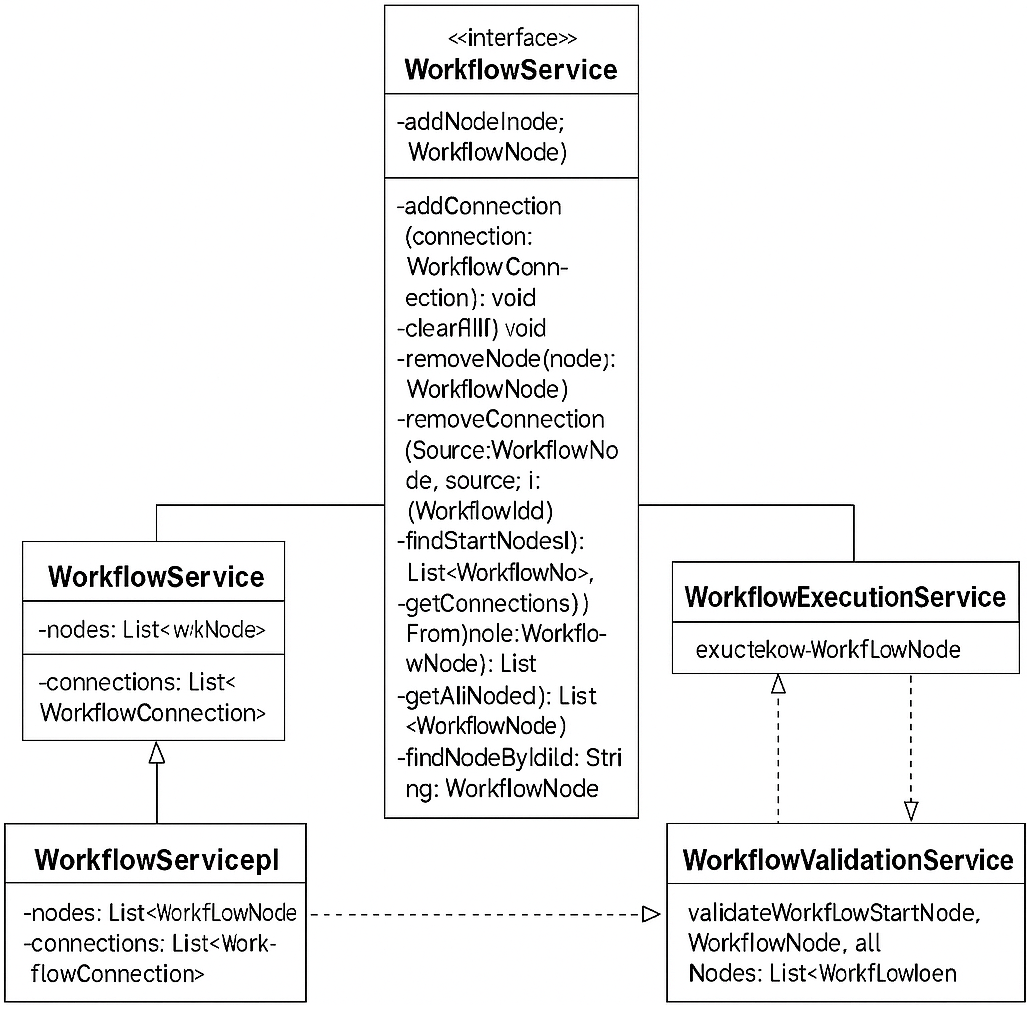

Figure 9: UML Diagram of the `service` Package

Overview of Structure:

- `WorkflowService` is a core **interface** that defines all major workflow operations including:
  - Adding/removing nodes and connections
  - Finding start nodes
  - Retrieving connections and nodes by ID
  - Clearing or resetting the workflow
- `WorkflowServiceImpl` is the concrete implementation of `WorkflowService`, providing actual storage and logic using in-memory collections such as `List<WorkflowNode>` and `List<WorkflowConnection>`.
- `WorkflowExecutionService` and `WorkflowValidationService` both depend on the interface and operate as specialized services. They provide domain-specific logic such as executing workflows and validating workflow correctness.

Design Highlights:

- **Abstraction:** The interface `WorkflowService` ensures the rest of the application interacts with the system through a clearly defined API, not implementation details.
- **Inheritance and Interface Usage:** `WorkflowServiceImpl` inherits from the interface, while other services such as `WorkflowExecutionService` and `WorkflowValidationServic` utilize it via composition or dependency injection.
- **Encapsulation:** All data structures (e.g., node lists, connection lists) are encapsulated within service implementations and are never exposed directly to external classes.
- **Polymorphism:** By programming against the `WorkflowService` interface, the system allows for flexible substitution of service implementations, supporting testing or runtime swapping.

Why It Matters:
This architecture promotes testability, maintainability, and extensibility. It allows business logic to evolve independently of the visual or control layers. Specialized services can be injected or extended without disrupting the broader system, making it ideal for scalable workflow orchestration.

5.9 UML: `util` Package

The `util` package serves as a toolbox for auxiliary features that support the broader system, without being tightly coupled to core architectural layers. The primary utility class in this package is `MetadataPrinter`, which is responsible for printing and formatting metadata across various workflow nodes in a reusable and type-safe way.

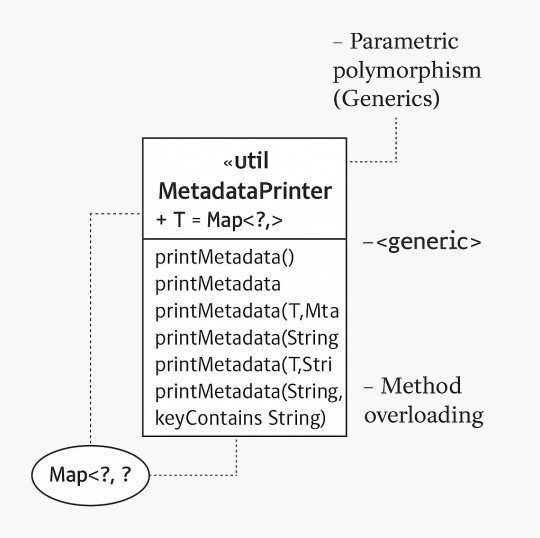

Figure 10: UML Diagram of the `MetadataPrinter` Class in the `util` Package

Key Design Elements and OOP Concepts:

- **Parametric Polymorphism (Generics):** The class is defined with a type parameter `T = Map<?,?>`, allowing it to be reused with different types of metadata structures. This promotes flexibility while maintaining type safety.

- **Encapsulation:** The actual logic for printing is fully encapsulated inside the class. External components only call high-level methods like `printMetadata()`.
- **Ad-hoc Polymorphism (Method Overloading):** The class offers multiple overloaded versions of `printMetadata()` to accommodate different printing contexts, such as filtering by keyword, using a prefix, or using timestamped output.
- **Reusability:** Because of its generic nature and flexible method signatures, the `MetadataPrinter` can be reused by different nodes (e.g., training, prediction, analysis) for logging structured metadata.
- **Abstraction:** The system interacts with this utility through a simple method interface, without needing to know the underlying data formatting logic.

Important Methods:

- `printMetadata()` — Default metadata printer.
- `printMetadata(T metadata)` — Prints a generic metadata map.
- `printMetadata(String prefix, T metadata)` — Adds a custom prefix to all entries.
- `printMetadata(T metadata, String keyFilter)` — Filters metadata keys containing a specific keyword.

This class demonstrates how thoughtful utility design can enhance code clarity, encourage reuse, and reduce duplication. It also reflects adherence to good software engineering practices by keeping utility logic centralized and loosely coupled.

5.10 UML: `view` Package

The `view` package represents the user interface layer of the workflow orchestration system. It is responsible for rendering the canvas, handling user interaction events, and launching the visual environment that bridges the system's logic with its graphical presentation.

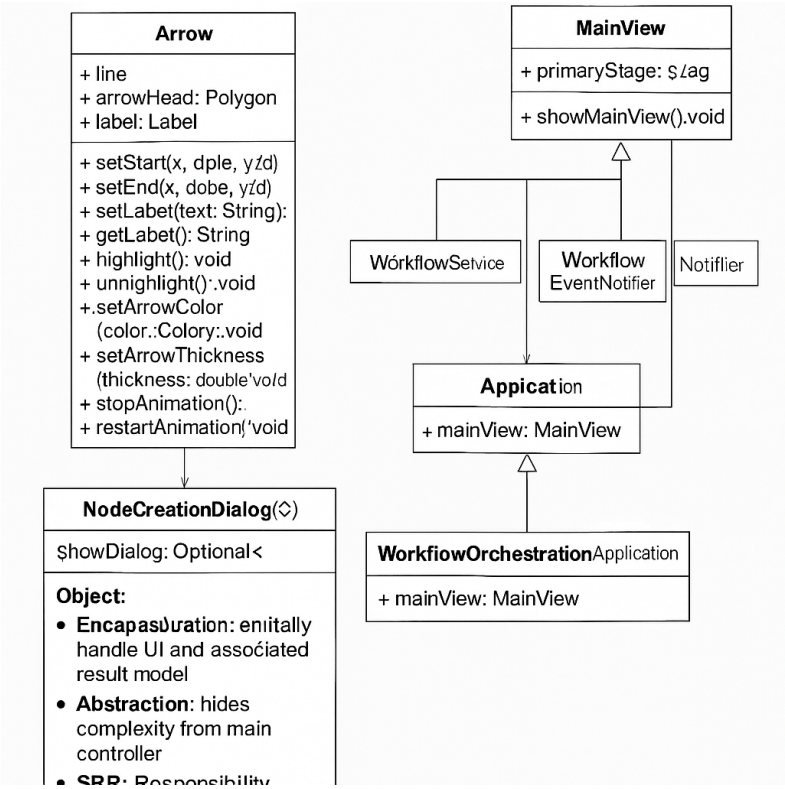

Figure 11: UML Diagram of the `view` Package

Key Classes and Roles:

- `Arrow`: A reusable JavaFX-based visual component for rendering directional connections between workflow nodes. It supports dynamic animations, label setting, highlighting, and styling operations.
- `NodeCreationDialog`: Encapsulates the UI for creating new nodes. This dialog is opened when the user requests to add a node, and handles user input for node details. It promotes:
  - **Encapsulation** by managing both UI logic and temporary result state internally.
  - **Abstraction** by shielding the main controller from dialog implementation details.
  - **SRP (Single Responsibility Principle)** as it is focused solely on node creation UI.
- `MainView`: Represents the entry point for launching the visual stage. It initializes and displays the primary GUI interface using JavaFX.
- `WorkflowOrchestrationApplication`: Acts as the application bootstrap class. It initializes the `MainView`, connects services (e.g., `WorkflowService`, `WorkflowEventNotifier`), and starts the JavaFX lifecycle.

Design Highlights:

- **Modularity:** Each class in the package has a clear, distinct purpose — visual rendering (`Arrow`), node input UI (`NodeCreationDialog`), view management (`MainView`), and application setup (`WorkflowOrchestrationApplication`).
- **Separation of Concerns:** The rendering logic is isolated from controller logic. For instance, the `MainView` class handles launching the visual UI, while coordination with services and observers is handled by the controller layer.
- **Reusability:** Components like `Arrow` are built to be instantiated multiple times with different start/end coordinates and labels, supporting extensibility in node interaction.
- **UI-Backend Integration:** The diagram shows that the `WorkflowOrchestrationApplication` holds a reference to the main view and integrates it with `WorkflowService` and the `WorkflowEventNotifier`, creating a loosely coupled yet coordinated application lifecycle.

This package illustrates how clean GUI architecture can enhance system usability, simplify maintenance, and preserve the integrity of OOP principles like abstraction, encapsulation, and SRP across visual layers.

5.11 UML: Full System Package Overview

To conclude the UML diagram section, the following figure presents a **high-level architectural overview** of how all major packages in the `Workflow Orchestration System` collaborate. While each has been described individually, this visual summary reinforces the structural coherence and object-oriented discipline of the project.

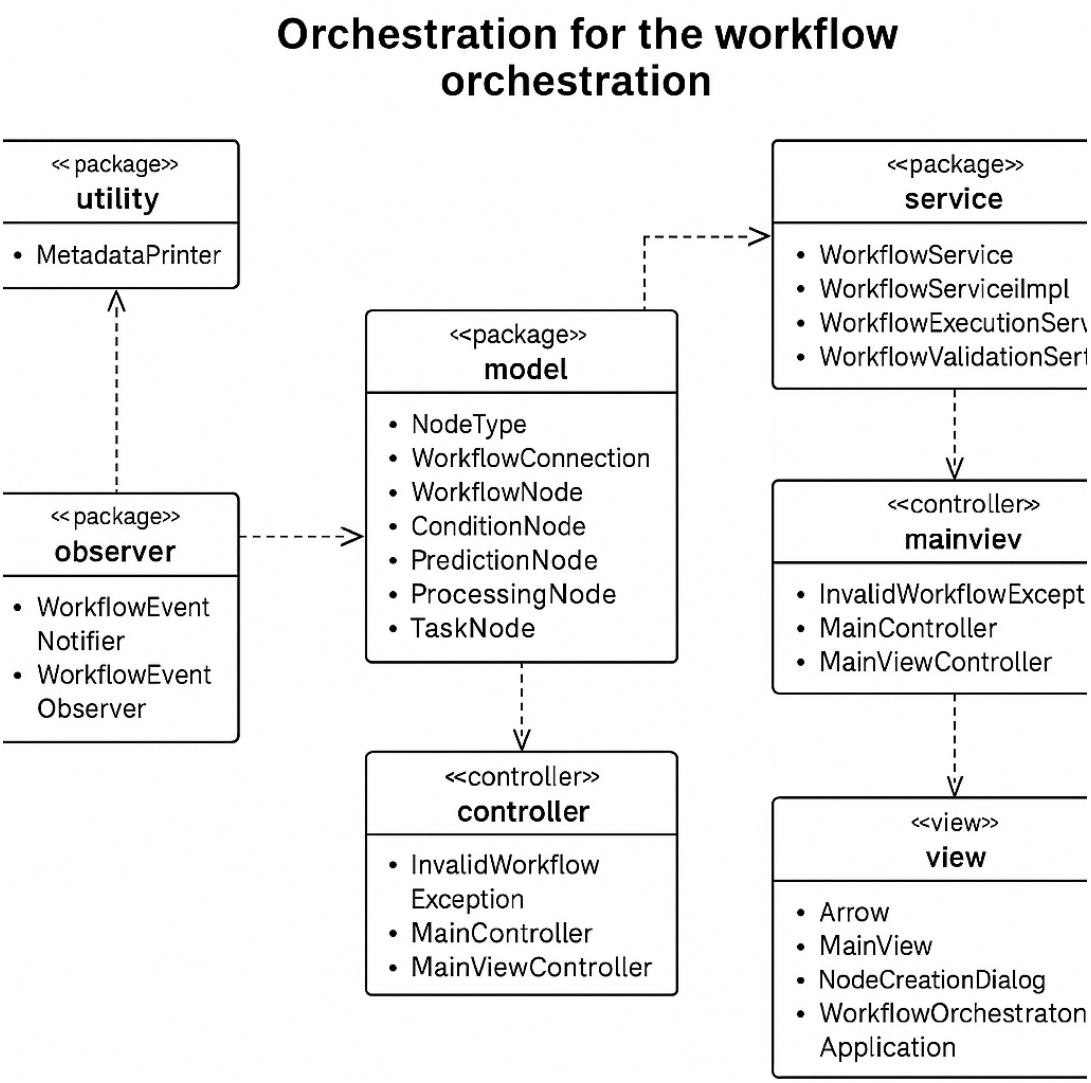

Figure 12: Package-Level UML Overview of the Workflow Orchestration System

Complete Architectural Breakdown:

- **Model Package:** Serves as the *core domain layer*. It defines essential entities such as:
  - `WorkflowNode` (abstract base class for all node types)
  - `NodeType`, `WorkflowConnection`, and various specialized nodes (e.g., `PredictionNode`, `ConditionNode`, `StartNode`, etc.)

  It also supports composition (e.g., grouped nodes), aggregation (via connections), and both interface-based and inheritance-based polymorphism.

- **Controller Package:** Acts as the *coordination layer* between the UI and backend logic.
  - `MainController` handles node creation, connection, execution, and delegation to services.

  - `MainViewController` captures GUI events and forwards them to backend logic.

  This design ensures strong modularity and decoupling between layers.

- **View Package:** Represents the JavaFX graphical interface. It contains:
  - `MainView`, `Arrow`, `NodeCreationDialog`, and the `WorkflowOrchestrationApplication` launcher.
  - These classes manage the visual rendering of the workflow and provide users with intuitive tools to create and navigate node-based pipelines.

- **Service Package:** Implements backend services for executing, validating, and manipulating workflows.
  - `WorkflowService` (interface), with `WorkflowServiceImpl` for actual logic
  - `WorkflowExecutionService` (manages workflow execution)
  - `WorkflowValidationService` (ensures correctness of start/end structure)

  This enforces the Single Responsibility Principle (SRP) and promotes interface-driven development.

- **Factory Package:** Contains `NodeFactory`, which abstracts and centralizes node creation using the **Factory Design Pattern**.
  - Supports loose coupling: consumers request node creation without needing to know subclass details.
  - Overloaded factory methods support additional metadata and validation when needed.

- **Command Package:** Implements the **Command Design Pattern**, enabling undoable user actions like:
  - `ConnectNodesCommand`, `DeleteNodeCommand`, `ExecuteNodeCommand`, etc.
  - `WorkflowInvoker` manages execution history for undo/redo.

  This pattern adds flexibility, encapsulation, and extensibility to user-driven operations.

- **Execution Package:** Contains the `GenericExecutionLogger<T>` class, which uses generics and method overloading to log execution results across various node types. Demonstrates multiple forms of polymorphism and emphasizes reusability and cohesion.

- **Observer Package:** Implements the **Observer Pattern** to notify subscribers of node-level events (e.g., when nodes are connected or executed).
  - `WorkflowEventNotifier` maintains observer lists.
  - `WorkflowObserver` interface supports extensible reactions to workflow changes.

- **Exception Package:** Contains a hierarchy of custom exceptions:

- `InvalidWorkflowException` (base)
- `InvalidMetadataException`, `UnsupportedOperationForNodeException` (specializations)

It enhances robustness by allowing node-level or workflow-level faults to be handled meaningfully.

- **Utility Package:** Houses helper classes like `MetadataPrinter`, a generic, overloaded class that prints metadata with filters or custom formatting. Demonstrates good design for stateless, reusable utility behavior.

Final Reflection:

This architecture was designed with a strong emphasis on **separation of concerns**, **layered modularity**, and the **application of key OOP principles and design patterns**. Each package plays a clear role in the system's orchestration and contributes to a scalable, maintainable, and extensible codebase.

6 Graphical User Interface Snapshots and Overview

This section presents a visual overview of the system's graphical user interface (GUI), illustrating its usability and interactive design. The GUI plays a critical role in enabling users to visually build, modify, and execute machine learning workflows without needing to interact with the underlying codebase.

Each snapshot is accompanied by commentary that explains its purpose, components, and how it reflects Object-Oriented Design principles like encapsulation, abstraction, and modularity. From the main application window to specialized dialogs and execution feedback, these GUI elements reflect a clean separation of concerns and user-friendly functionality.

6.1 Main Application Window

The **Main Application Window** is the primary entry point and user workspace of the Workflow Orchestration System. It integrates all the essential tools and panels for node creation, graph visualization, workflow execution, and metadata editing. The following figure showcases the full layout upon launching the application:

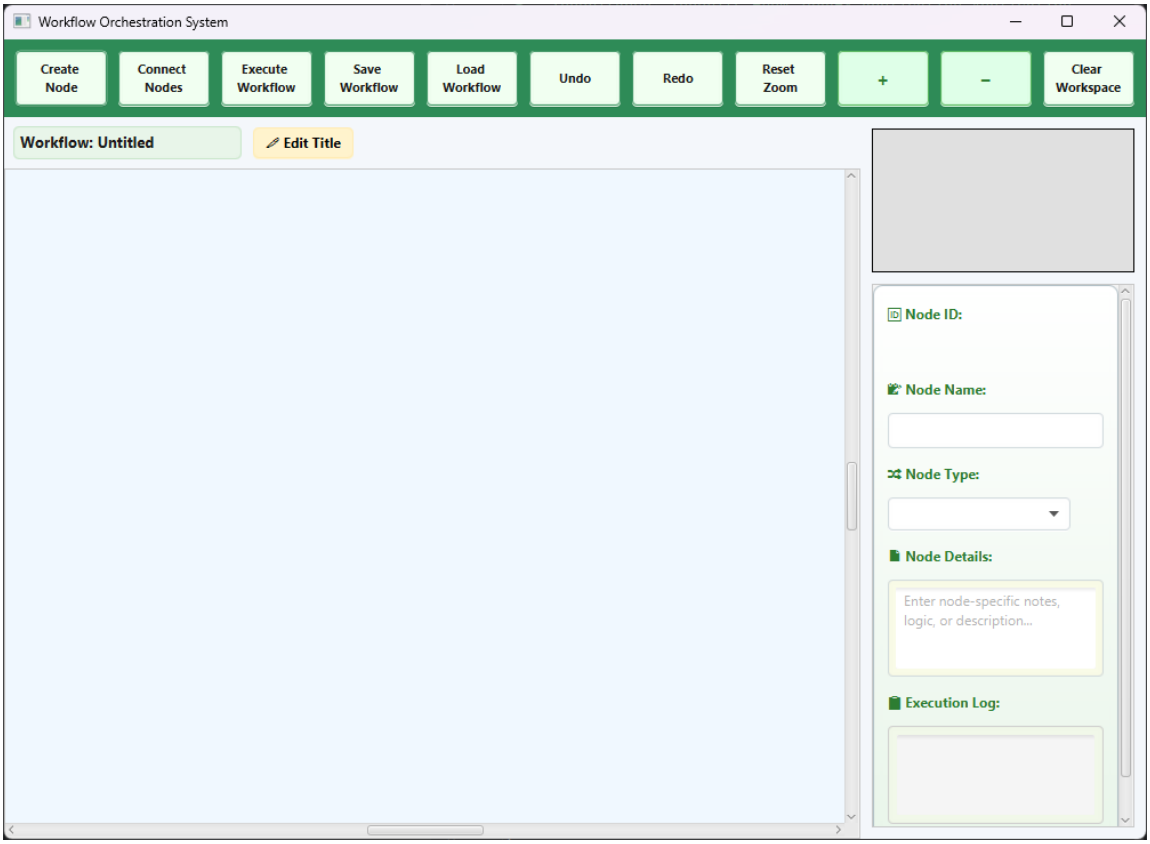

Figure 13: Main Application Window — Complete System Overview

Top Toolbar (Action Controls):

- `Create Node:` Opens the Node Creation Dialog for adding a new node to the workspace.
- `Connect Nodes:` Enables connection mode to draw arrows between selected nodes.
- `Execute Workflow:` Starts workflow execution from START nodes, traversing the DAG.
- `Save Workflow:` Saves the current graph layout and metadata to a file.
- `Load Workflow:` Loads a previously saved workflow for editing or execution.
- `Undo / Redo:` Allows users to reverse or reapply their last actions.
- `Reset Zoom / + / - :` Controls the zoom level on the canvas for easier navigation.
- `Clear Workspace:` Wipes the entire canvas, removing all nodes and connections.

Editable Title Panel:

- Displays the name of the current workflow.
- Users can click `Edit Title` to rename the workflow dynamically.

Main Canvas (Workflow Area):

- The large blue panel in the center is the workflow construction canvas.
- Users can visually add, move, and connect nodes here using drag-and-drop interaction.
- Arrows dynamically reflect connections and data/control flow between nodes.
- Scroll bars allow full navigation of the graph in larger workflows.

Sidebar (Property Panel):

- **Node ID:** Displays the internal UUID of the currently selected node.
- **Node Name:** Editable text field for setting the node's name.
- **Node Type:** A dropdown menu that allows the user to change the node's functional type.
- **Node Details:** Multi-line input area for writing logic, formulas, or comments tied to this node.
- **Execution Log:** Displays real-time feedback after workflow execution (e.g., "✓ Training complete").

The Main Application Window exemplifies modular architecture, clean design, and user-centered usability. It centralizes all visual workflow building operations while integrating advanced editing, logic configuration, and execution monitoring into a single interactive environment.

6.2 Node Creation Dialog

The **Node Creation Dialog** serves as the entry point for adding new workflow nodes. When users click the `Create Node` button in the top toolbar, this dialog appears to let them configure all relevant metadata associated with a node. It ensures intuitive creation of nodes with customized behavior, labels, and details.

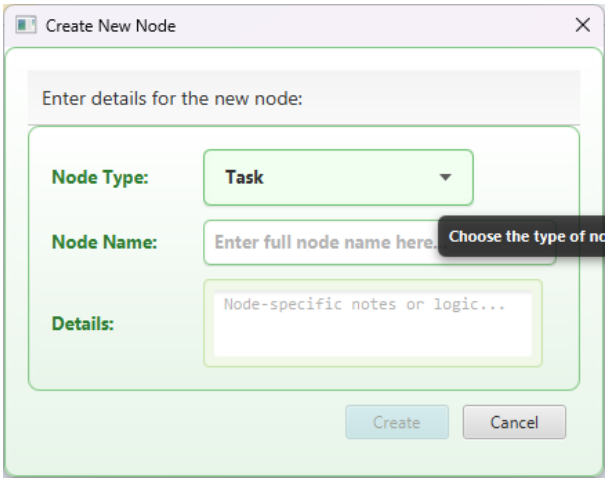

Figure 14: Node Creation Dialog — Adding a New Node

Main Fields and Functionality:

- **Node Type (Dropdown):** Users can select from a wide range of supported node types, including `Task`, `Condition`, `Prediction`, `Training`, `Start`, `Output`, and many more. These types are defined in the system's model layer and determine the node's role in the workflow.
- **Node Name (Text Field):** Each node can be given a custom name to distinguish its functionality or purpose. Multiple nodes of the same type can be named differently for clarity and documentation.
- **Details (Multi-line Area):** This field allows users to insert node-specific logic, notes, or comments. These details are useful for storing conditional logic, explanations, or formulas.

Once created, the node appears on the central canvas with the assigned name visible. Its details are hidden from the visual layout but automatically populated in the right-hand **Sidebar Panel** when selected. This two-layer structure supports visual clarity while retaining deep contextual metadata.

Users may continue to:

- Dynamically edit the workflow title via the `Edit Title` button.
- `Save` a configured workflow for future reuse or modification.
- `Load` an existing workflow and resume configuration.
- Quickly switch between saved versions by saving with a new title.

Overall, the Node Creation Dialog encapsulates the principle of guided input, ensuring that users build logically consistent and clearly documented workflows with ease.

6.3 Visual Node Layout and Connections

The visual core of the Workflow Orchestration System is its canvas—an interactive area where users can spatially construct, inspect, and execute machine learning workflows through intuitive node placement and connection.

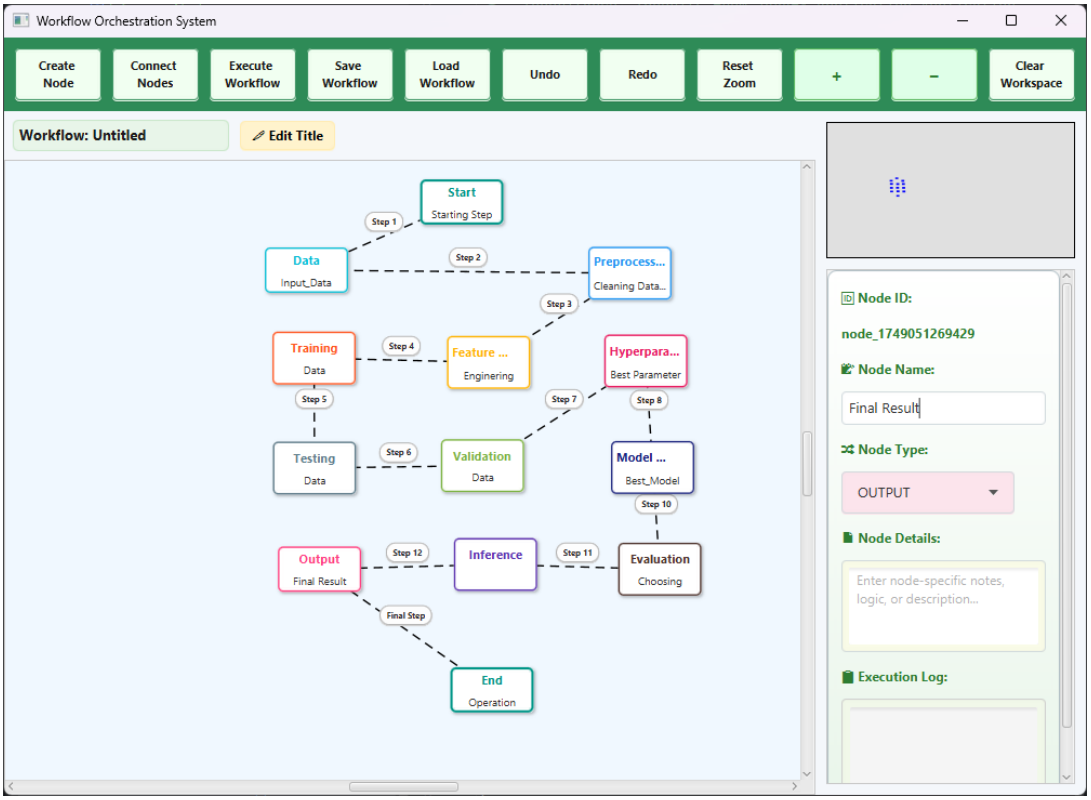

Figure 15: Visual Node Layout, Labels, and Execution Highlights

Workflow Construction via Node Layout:

- Each node on the canvas represents a discrete computational task or control unit such as `Data`, `Preprocessing`, `Training`, `Inference`, `Evaluation`, or `Output`.
- Nodes are created through the `Node Creation Dialog` (see previous section), with each node carrying a custom user-defined **name**, **type**, and **internal logic**.
- Upon creation, nodes are placed into the canvas dynamically, and are styled according to their type (see below).

Color-Coded Node Types:

- Every node type is color-coded for immediate visual recognition. For example:
  - `Data` nodes appear with a light-blue border.
  - `Feature Engineering` nodes use orange.
  - `Training` and `Output` nodes appear in red tones.
  - `Validation` nodes use green.
  - `Inference` nodes have a violet outline.
  - `Start` and `Testing` nodes adopt their unique styles.
- These color mappings significantly improve readability and allow the user to visually group stages of the pipeline.

Labeled Connections (Edges):

- Nodes are connected using directional arrows that represent control or data flow.
- Each connection can be given a label—such as `Step 1`, `Step 2`, etc.—which appears directly on the arrow.
- These labels are fully customizable and can reflect execution order, stage names, or logical steps in the pipeline.
- Connection arrows are interactive and visually responsive.

Animated Workflow Execution:

- Upon clicking `Execute Workflow`, the system animates the workflow's progression:
  - The `Start` node lights up with a yellow highlight.
  - As the execution moves through each connection, arrows light up sequentially.
  - Each target node highlights upon activation, showing the current stage being "executed".
  - This creates a visual trace of execution—an intuitive animation showing how the workflow flows node-by-node.
- The system ensures every step of the workflow is reflected on-screen, combining clarity with interactivity.

Sidebar Synchronization:

- When users click on a node, the **right-hand sidebar panel** dynamically updates to show:
  - `Node ID` (automatically generated).
  - `Node Name` (editable).
  - `Node Type` (dropdown-style selector).
  - `Node Details` (multi-line notes or logic field).
  - `Execution Log` (outputs during runtime).
- This tight integration between visual layout and property panel enhances the user experience and maintains low cognitive load.

Summary: This visual layer exemplifies how a graphical interface can embody powerful OOP and user-centered design principles. With dynamic styling, labeled transitions, synchronized metadata, and animated execution, the workflow layout interface provides a high degree of transparency and elegance. It empowers users—both technical and non-technical—to model complex workflows with minimal effort and maximum clarity.

6.4 Dynamic Workflow Execution and Runtime Feedback

One of the most visually and functionally impressive components of the Workflow Orchestration System is its real-time execution engine, which animates workflow progression and integrates both system logs and visual feedback to ensure clarity, transparency, and developer usability.

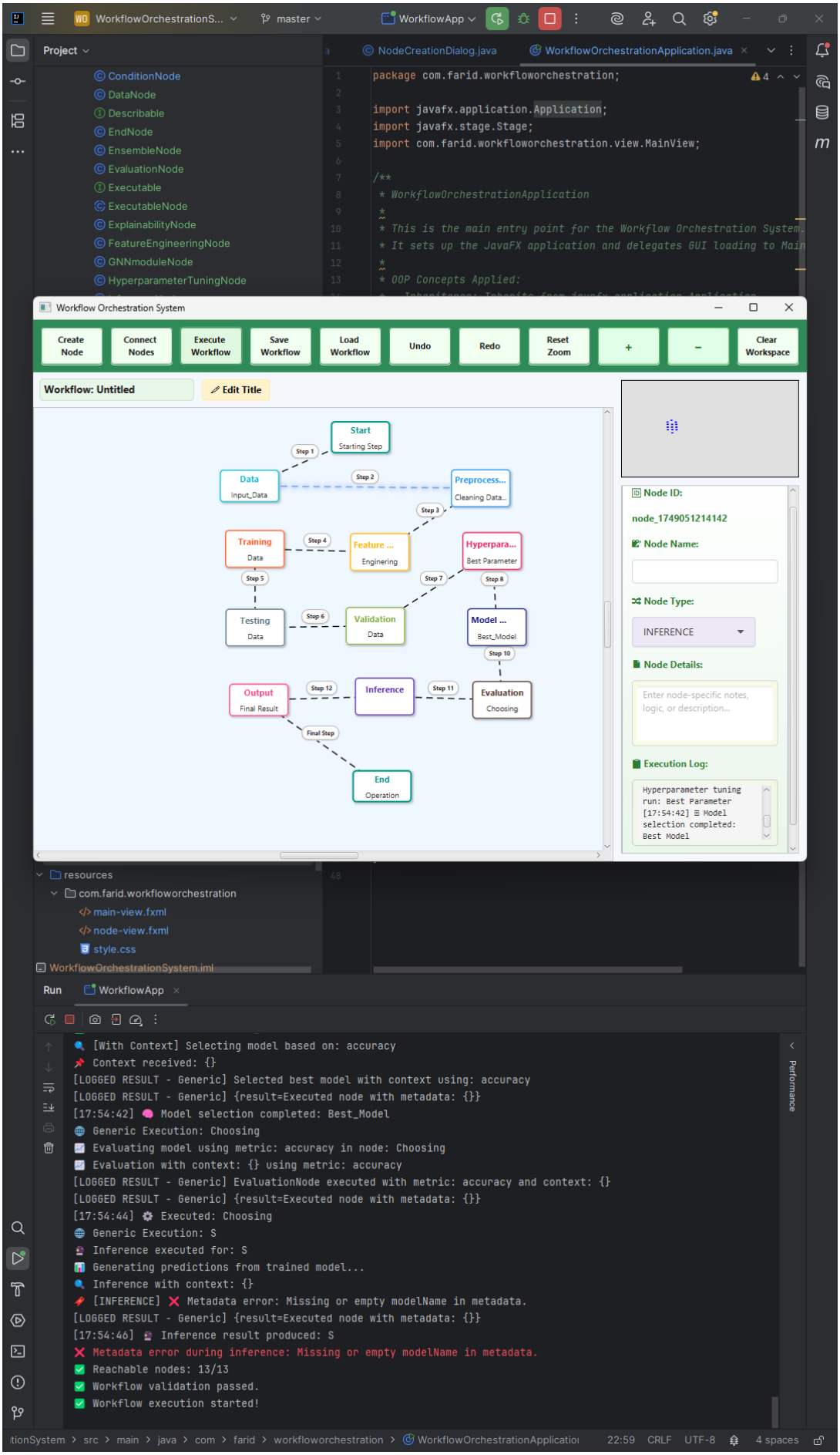

Figure 16: Visual Execution Feedback and Runtime Logging

Animated Execution Flow:

- Upon clicking `Execute Workflow`, the system triggers a node-by-node traversal starting from the `Start` node.
- Active nodes are highlighted in yellow, indicating real-time execution.
- As the workflow progresses, the outgoing connection arrows are animated to bright blue, giving a sense of directionality and current progress.
- This dual-color animation pattern (node + arrow) provides instant feedback on both control flow and execution depth.

Live Runtime Logs:

- Execution details are concurrently printed to both:
  - The terminal console (developer log).
  - The right-hand `Execution Log` panel (user-friendly log).
- Execution logs include:
  - Node name and stage.
  - Execution results or decisions (e.g., "Model selection completed").
  - Errors or warnings.

Enhanced Visual Logs with Stickers:

- For greater visual appeal and user clarity, symbolic **stickers** are added to logs in the sidebar.
- These icons (e.g., `[Check]`, `[Alert]`, `[Launch]`) enhance scanability and bring a modern, delightful UX to an otherwise technical interface.

Built-in Error Detection:

- Nodes missing mandatory configuration (e.g., name or metadata) are immediately flagged.
- In this example, the `Inference` node was executed without a valid model name, triggering an explicit runtime error:

  ```
  Metadata error during inference:  Missing or empty modelName in
  metadata.
  ```

- This demonstrates the engine's robustness in handling null safety and its clear feedback cycle for debugging.

Design Highlights:

- Execution visuals are synchronized across UI, console, and internal logic layers.

- Node flow animations are fully coordinated with their metadata states and transitions.
- Labelled connections (e.g., “Step 1”, “Step 2”, “Final Step”) are retained during execution, preserving the semantic trail.

Conclusion: This execution mechanism transforms abstract workflow definitions into a tangible, observable runtime journey. The real-time arrow progression, animated node highlighting, integrated visual logs, and live error feedback collectively create a powerful simulation interface that is both informative and delightful to use.

6.5 Sidebar Interaction and Mini-Map Navigation

The user interface of the Workflow Orchestration System includes a powerful right-hand **Sidebar Panel**, designed for contextual node management, and a compact yet effective **Mini-Map Navigator** that offers spatial awareness within complex workflows.

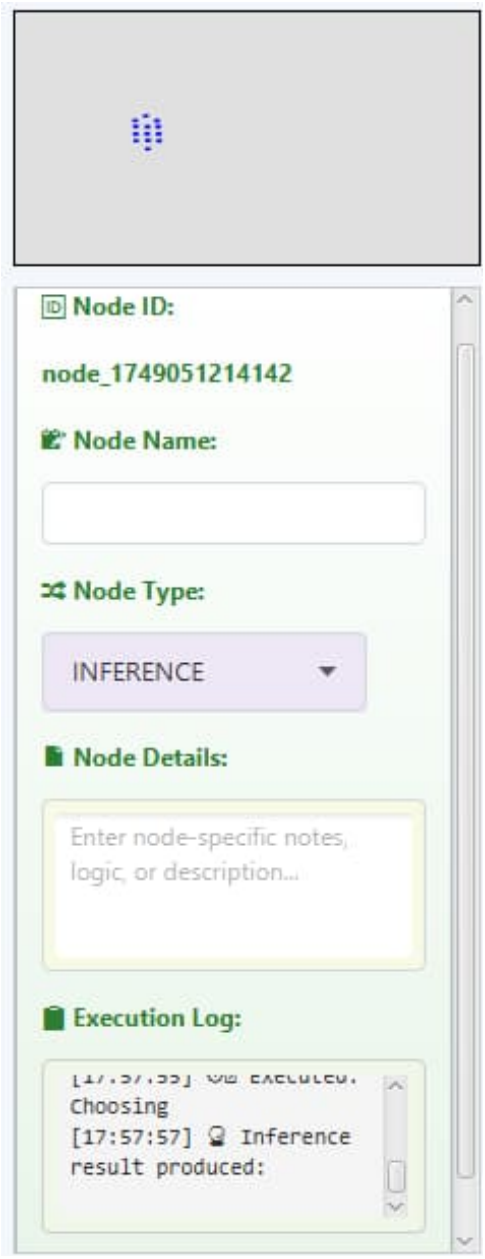

Figure 17: Sidebar Panel with Mini-Map (Top) and Node Metadata Editor (Bottom)

Mini-Map Overview:

- Positioned at the top of the sidebar, the mini-map provides a real-time visual preview of the entire workflow canvas.
- This is especially valuable for navigating large-scale workflows, helping users track their current viewport location and spatially orient themselves.
- As the user pans or zooms across the canvas, the mini-map updates dynamically, ensuring synchronization with the main workspace.

Dynamic Sidebar Panel:

- The sidebar reflects the properties of the currently selected node. All fields update in real time, providing instant contextual feedback.
- **Node ID:**
  - Automatically generated UUID-like identifier for each node.
  - Guarantees internal uniqueness for referencing, logging, and debugging.
- **Node Name:**
  - Fully editable at runtime.
  - Changes are reflected immediately on the node's visible label in the canvas.
  - Supports updating even after node creation, allowing for flexible naming conventions.
- **Node Type:**
  - A drop-down menu allows the user to dynamically change the type of a node (e.g., from `TASK` to `PREDICTION`).
  - Upon type change, both the node's border color and sidebar label update accordingly.
  - This mechanism showcases polymorphic behavior and CSS-linked type adaptation.
- **Node Details:**
  - Multi-line text area for storing metadata, description, or node-specific logic.
  - Acts as a documentation and configuration field for each node.
  - Information entered here is later used during execution to generate context-aware logs.
- **Execution Log:**
  - Displays a chronological trace of runtime messages relevant to this specific node.
  - Populated automatically during execution, synchronized with the global console log.
  - Uses colored symbolic tags (e.g., `[Check]`, `[Alert]`) to enhance readability and UX appeal.

Visual Synchronization:

- All sidebar components are live-bound to the selected node.
- Any updates made through the panel reflect instantly on the visual node, preserving the WYSIWYG principle (What You See Is What You Get).

- This real-time coupling between UI and data model emphasizes modularity, interactivity, and precision.

  Conclusion: Together, the mini-map and the sidebar form a cohesive node management ecosystem. The mini-map enhances macro-level navigation, while the sidebar enables micro-level editing and tracking. Their thoughtful integration significantly elevates the user experience, combining performance, flexibility, and accessibility in a modern orchestration interface.

7 Technical Challenges and Resolutions

Developing the Workflow Orchestration System was not only a rewarding academic endeavor but also a deeply challenging technical journey that pushed my understanding of software engineering, object-oriented design, and user interface development to new heights. The project spans across more than 30 classes and thousands of lines of code, but the greatest complexities emerged in areas where logic, visuals, and interaction tightly intertwined.

The GUI: A Test of Precision and Patience

The single most technically demanding part of this project was the `graphical user interface`. In particular, the `MainViewController` class alone grew to nearly **2000 lines of code**, encapsulating a diverse set of responsibilities — from workflow execution and event handling to dynamic zoom, scroll, animation, node placement, and property panel synchronization. The sheer size and complexity of this class meant that a single visual bug could take hours to isolate and resolve.

- Implementing real-time **execution highlighting** — where nodes turn yellow as the workflow progresses — required careful control over thread timing and state propagation to ensure visual accuracy and responsiveness.
- Building the **mini-map** for visual navigation involved calculating node bounds and scroll coordinates dynamically. This was particularly difficult because the layout changed as users zoomed, dragged, or added nodes.
- Designing the **dynamic sidebar panel** was also incredibly difficult. Making node fields (like name, type, and details) **live-editable** while syncing them visually to the actual node on canvas — and reflecting color changes accordingly — required layered event handling and deep UI integration.

Color Coding and Style Syncing

Applying a unique color scheme to each node type was far more complex than it appeared. It required:

- Creating a fully customized JavaFX CSS system with type-specific classes.
- Dynamically updating both the node box and its `label color`, even when edited post-creation.

- Propagating those changes to the side panel in real time — such that the dropdown and the background reflected the chosen type — was especially painful to debug.

Connecting Logic to Interaction

Even simple user actions like "Create Node" or "Connect Nodes" required deep backend logic to be stitched in seamlessly. This involved:

- Designing a dialog that captures user input and converts it into a working node object.
- Connecting nodes dynamically, supporting undo/redo history, and labeling the arrows (e.g., "Step 1") in real time.
- Executing workflows from the `Start` node and maintaining a correct branching order, even under complex conditional and inference logic.

Overcoming Failure and Growing

There were countless moments where things did not work as expected. From style issues that refused to reflect changes, to runtime bugs in the execution graph, to node interactions that broke the layout — each bug was a puzzle. But, as the inventor Thomas Edison once said, *"I have not failed. I've just found 10,000 ways that won't work."*

This mindset gave me resilience. Over the course of more than **two and a half months**, I learned to:

- Debug both logic and UI visually and structurally.
- Modularize complex controllers into manageable parts.
- Embrace failure as a path to understanding.

A Journey of Self-Driven Mastery

Most importantly, this project taught me that growth in software development doesn't come just from writing code — it comes from *refining*, *restructuring*, and *persisting* through complexity. I didn't just apply OOP concepts — I lived them. And I did so with the goal of delivering not just a functional tool, but a **polished, intuitive, and extensible** system that can scale well into the future.

8 Project Management and Development Methodology

This project was developed using an agile-inspired, iterative development approach spread over approximately **two and a half months**. Given the complexity of the system and the deep integration of backend logic with frontend GUI behavior, it was essential to structure development into flexible, goal-driven phases rather than rigid waterfall steps.

Sprint-Based Planning

Although working solo, I loosely followed a Scrum-style methodology, organizing work into weekly "sprints" with the following rhythm:

- **Planning and Design:** Early weeks focused on defining major components, identifying key OOP features to integrate (e.g., inheritance, abstraction, polymorphism), and sketching early UI layouts.
- **Incremental Development:** Each sprint introduced a new set of features — such as node creation, connection logic, CSS styling, and sidebar editing — while refactoring older parts based on feedback and test results.
- **Testing and Debugging:** Regular testing was performed manually and interactively through the GUI to simulate realistic user actions. Exception handling and user feedback mechanisms were also added gradually.
- **Review and Polish:** Final weeks focused on UI responsiveness, visual clarity, undo/redo behavior, node type styling, and advanced visual feedback like arrow animations and workflow execution paths.

Tools and Version Control

- **Development Environment:** IntelliJ IDEA was used as the primary IDE for JavaFX development and code management.
- **Versioning:** Git was used locally to track progress, experiment with alternatives, and roll back changes when visual or logical regressions occurred.
- **Task Management:** A simple Trello board helped break down work into subtasks — such as "Add Execution Stickers," "Fix Node Drag Zoom Bug," or "Highlight Active Arrow" — allowing progress tracking without overhead.

Adaptability During Development

While the original plan focused purely on structural node creation and connection, the scope expanded organically as development progressed. Several advanced features — such as:

- Dynamic node type switching from the sidebar
- Node metadata editing and live UI synchronization
- Animated arrow paths and execution color transitions
- Minimap navigation and zoom scaling

were added later as a result of experimentation and visual exploration.

Learning by Doing

This methodology helped me not only manage a technically intense system but also **grow incrementally**. Every feature required breaking down a problem, testing prototypes, and continuously refining implementation.
Ultimately, this approach allowed the project to remain focused yet flexible, delivering a complete system that is not only feature-rich and visually polished, but also **maintainable and extensible**.

9 Extensibility and Maintainability Considerations

From the very beginning of this project, I approached the system not as a one-time prototype, but as the foundation for a full-fledged **software engineering product**. While it was developed as an academic Object-Oriented Programming (OOP) project in Java, the underlying design decisions were guided by a forward-looking mindset — one that prioritized *extensibility, reusability, and maintainability.*

Node Modularity and Future Expansion

The architecture was built around the idea that new `WorkflowNode` types — such as custom AI models, preprocessing steps, or evaluators — could be easily added with minimal code disruption. This is achieved by:

- Using a central `NodeFactory` with an extensible switch-case pattern, allowing new node types to be created dynamically without altering core logic.
- Maintaining a clean `enum NodeType` structure, so each node has a clear identity, label, and CSS style that can be easily expanded.
- Adopting the `Executable` and `Describable` interfaces to standardize behavior across node types.

Separation of Concerns and Controller Logic

A clear separation of concerns was enforced to prevent code bloat and facilitate debugging:

- Workflow orchestration logic was encapsulated in the `MainController`, while the JavaFX GUI and user events were handled in the `MainViewController`.
- This separation allows the backend logic (execution, validation, node storage) to be extended in the future — for example, by integrating with cloud-based services or database backends — without impacting the UI logic.

Visual Synchronization and Style Consistency

The system was also designed to be visually consistent and intuitive, with scalability in mind:

- A fully dynamic sidebar ensures that any newly added node types will automatically appear in the property panel.
- The CSS styling system maps each node type to a unique visual identity. This structure allows developers to add styles for new nodes without editing existing visual code.

Prepared for Long-Term Evolution

This system was not just written to "work," but to evolve. As the next step in this project, I aim to transition it into a more advanced software engineering environment — potentially turning each node into a microservice, introducing real-time collaboration, or deploying the system as a web-based orchestration platform.

I believe the architectural decisions made throughout this project will make that transition smoother:

- Modular package design enables future replacements or upgrades (e.g., swapping the `GenericExecutionLogger` with an external monitoring tool).
- All execution behavior is centralized and interface-driven, allowing plug-in architectures or distributed execution flows in the future.
- Undo/Redo logic and state tracking are decoupled from visual layout, which is essential for stability as complexity grows.

A Developer-Centric System

Ultimately, this project was built by a developer — for developers. My focus was to build a platform that is not only functional and interactive, but also *clear, extensible, and resilient*. Whether it's adapting the canvas layout, enhancing the node types, or exporting execution logs to external services, the system was designed to accommodate those changes with confidence.

10 Deployment and Runtime Environment

The *Machine Learning Workflow Orchestration System* was developed and tested in a modern Java environment with an emphasis on portability and maintainability. To ensure stable builds, cross-platform compatibility, and clean dependency management, the project was structured using standard Java development tools and industry best practices.

Development Environment

- **Operating System:** Microsoft Windows 10 and 11 were used for development and testing, ensuring full desktop compatibility with high-DPI display scaling.
- **IDE:** The project was implemented primarily using **IntelliJ IDEA**, which provided robust support for Maven, JavaFX, and Java 17.
- **Build Tool: Apache Maven** was employed to handle project dependencies, build lifecycle, and packaging. This guarantees reproducibility and easy configuration sharing across environments.

Core Technologies

- **Java SDK:** The system is built using **Java Development Kit (JDK) 17**, a long-term support version known for stability and performance.
- **JavaFX:** A key component of the graphical user interface, **JavaFX 17+** was used to implement all UI features, including drag-and-drop nodes, zooming, custom tooltips, animated arrows, and a responsive sidebar.

- **FXML and CSS:** The layout and styling of the user interface were defined using declarative `.fxml` files and custom JavaFX `.css` stylesheets to promote separation of concerns between logic and presentation.

Packaging and Execution

- The application is structured into cleanly separated packages, including `controller`, `model`, `factory`, `service`, `observer`, `util`, and `view`. This modular design supports maintainability and scaling.
- The entry point for launching the application is provided through the `Main.java` class, which initializes the primary JavaFX scene and loads the initial FXML layout.
- Workflow logic is coordinated through the `MainViewController.java`, while each node type is represented by a unique class under the `model` package. Additional services such as undo/redo and logging are encapsulated in dedicated utility and service packages.

Execution Environment

- A JavaFX-compatible JDK is required to run the application. Deployment was tested using the latest JDK 17 distributions with JavaFX bundled or referenced as external dependencies via Maven.
- Users can launch the application either through their IDE or by building and executing the packaged JAR file using:

```
mvn clean install
java -jar target/workflow-orchestrator.jar
```

Cross-Platform Support

While development was performed on Windows, the system is inherently cross-platform thanks to Java's portability. With proper JavaFX support, the application can also run on macOS or Linux with minimal adjustments, making it suitable for educational, research, or enterprise use.

11 Vision for Future Enhancements and Expansion

While the current version of the *Machine Learning Workflow Orchestration System* already embodies strong object-oriented design, a rich graphical interface, and meaningful interaction between user and workflow logic, it is important to emphasize that this milestone is not the end — it is only the beginning.

A Foundation for Something Greater

This project was envisioned not as a standalone academic exercise, but as a stepping stone toward building a truly extensible, modular, and production-ready orchestration framework. The groundwork laid in this phase — from the intricate node models and visual execution feedback to the sidebar-driven control mechanisms — serves as a launchpad for future software engineering ambitions.

Planned Future Enhancements

The next iterations of this system are expected to include the following transformative features:

- **Microservice Integration:** Each node type will be enhanced to function as an independently deployable microservice, capable of executing domain-specific logic (e.g., training, inference, clustering) across distributed environments. This will allow the orchestration engine to scale horizontally and operate in cloud-native infrastructures such as Kubernetes or Docker Swarm.
- **RESTful APIs for Node Communication:** Instead of direct in-memory execution, node-to-node communication will be mediated via HTTP-based APIs, enabling a separation of concerns and supporting remote service orchestration.
- **Graph Serialization and Live Deployment:** Users will be able to design a graph visually, export it as a JSON or YAML specification, and deploy it live into a containerized environment, enabling real-world machine learning pipelines to be configured and deployed without writing code.
- **Plugin System for Custom Nodes:** A dynamic plugin architecture will be introduced to allow third-party developers to define and register custom node types with their own execution behaviors, UI representations, and configuration schemas.
- **Version Control and Workflow Snapshots:** Future versions will allow users to save, diff, and roll back workflow configurations — effectively introducing version control into the workflow space.

A True Software Engineering Vision

This vision moves the project from the realm of Java-based OOP into the discipline of full-stack software engineering. The long-term objective is to evolve this academic system into a research-grade or enterprise-grade orchestration tool that is:

- Scalable across multiple machines and services.
- Maintainable through clearly defined modular boundaries.
- Extendable by teams of developers for domain-specific use cases.
- Deployable in real-time production environments.

A Personal Commitment

This project represents not just technical effort, but a personal commitment to learning, growth, and innovation. Although the current system demanded great time and energy — with over two and a half months of development — the excitement of seeing each part come to life kept the motivation strong. As I move forward, this foundation will serve as the springboard toward building one of the most visually intuitive, technically robust, and widely applicable orchestration platforms.

12 Comparative Analysis with Real-World Tools

In the vast landscape of modern orchestration tools, platforms like **Node-RED**, **Apache NiFi**, **KNIME**, and **Airflow** dominate as industry benchmarks for data pipeline management, workflow execution, and process automation. Remarkably, this JavaFX-based Workflow Orchestration System — conceived and implemented entirely as a solo academic project — shares many architectural and functional parallels with these systems while introducing uniquely refined visual and interactive elements that set it apart.

Visual Workflow Design and Node Customization

Much like **Node-RED**'s drag-and-drop interface and **KNIME**'s modular node-based programming, this system enables users to visually design, build, and execute complex workflows using an intuitive graphical interface. However, it takes visual clarity a step further by introducing:

- **Color-coded Node Typing:** Each node type (e.g., START, TASK, INFERENCE, GNN_CLUSTERING) is rendered with a distinct border and label color, making complex graphs easy to scan and mentally parse — a feature often overlooked or inconsistently applied in other platforms.
- **Live Type Editing:** Unlike most static graph systems, this system allows users to dynamically change the type of any node from the sidebar panel, with immediate visual and logical reflection in both the node and its interactions.
- **Live Renaming and Detail Editing:** Node names and metadata (like execution logic or descriptions) can be edited live through the sidebar, with real-time synchronization to the canvas. This level of editable interactivity, directly from the UI, creates a developer-friendly experience not even available in some commercial tools.

Execution Highlighting and Animation

Most real-world workflow engines like **Airflow** focus on backend execution and job scheduling, offering very limited or no visual execution feedback. In stark contrast, this system introduces a highly polished **animated execution flow**, where:

- Nodes turn yellow as they are executed, giving a sense of real-time progression.
- Arrows light up in blue as the execution traverses them, making the direction and sequence of execution instantly visible.
- Execution logs are presented in both terminal and GUI views, with visual stickers (such as ✓ or) enhancing clarity and aesthetic appeal.
- Conditional nodes like IF statements activate only their correct branches, demonstrating dynamic decision-making inside the visual flow — a capability typically handled in code-based orchestrators, but made visual here.

Navigation and Usability Enhancements

This system also introduces thoughtful usability features typically found only in professional platforms:

- **Mini-map Navigation:** A live mini-map helps users orient themselves when dealing with large graphs, enabling them to pan and zoom intelligently.
- **Dynamic Canvas:** The canvas supports zooming, panning, and centering logic to ensure workflows can scale without performance degradation or UI clutter.
- **Undo/Redo Functionality:** Every user action, including node creation, connection, and deletion, is tracked, allowing seamless undo/redo — a feature that dramatically increases user confidence during editing.
- **Labeling of Arrows:** Connections between nodes are not just visual lines — they are labeled (e.g., "Step 1", "YES", "NO"), making logical flow traceable at a glance.

Architectural Modularity and Extensibility

Where many commercial tools rely on proprietary formats and rigid plugin systems, this project demonstrates flexibility through classic OOP paradigms:

- **Abstract Classes and Interfaces:** Each node extends from an abstract `WorkflowNode` superclass, ensuring consistent behavior while allowing individual specialization.
- **Enum-based Node Typing:** The `NodeType` enum allows centralized control over available node categories and their visual mappings, making the system easy to extend.
- **FXML-Based Layouts:** Each node has a separate FXML layout file, ensuring visual and logic separation — a best practice in JavaFX applications that enhances maintainability.

Real-World Alignment

The result of all this design is a system that mirrors real-world orchestration tools in intent and capability while delivering unmatched GUI polish for an academic project. Compared to platforms like:

- **Node-RED:** While Node-RED offers greater plugin availability, this project's GUI and execution feedback are notably more polished and visually expressive.
- **KNIME:** Although KNIME supports a wide range of ML operations, it lacks the real-time interactivity and live-editing capabilities this system provides out-of-the-box.
- **Airflow:** This system adds an interactive layer to what Airflow lacks — graphical progression, live logs, and node-based modular UI.

A Surprising Reality:

This entire system — with over **30 Java classes**, advanced JavaFX animation, execution control, CSS-driven theming, and intelligent layout logic — was developed by a single student over the course of just **10 weeks**. Every feature, from colored nodes to side panel sync, from error handling to GUI threading, was built from the ground up, representing not just a project, but a journey of professional growth, creative exploration, and technical excellence.

In short, this system is not just a student prototype — it is a working foundation for a larger software engineering product, designed with real-world extensibility, performance, and user experience in mind.

13 Reflection on Learning Outcomes

Looking back at the journey of building the *Machine Learning Workflow Orchestration System*, what began as a university assignment quickly transformed into a deeply immersive and eye-opening experience in software development, design thinking, and personal resilience. This project was not just about fulfilling academic criteria — it became a story of growth, discipline, and discovery.

Learning by Doing — and Redoing

From the outset, I committed to learning every concept not just in theory, but through hands-on trial and error. I started with foundational Object-Oriented Programming principles like *encapsulation*, *inheritance*, and *polymorphism*, and gradually began applying them across dozens of interdependent classes. What seemed abstract in books became second nature as I designed real interfaces, abstract base classes, and inherited behaviors that powered this complex system.

Every error — whether a visual glitch, a missing connection, or a broken piece of logic — became an invitation to dive deeper. Debugging wasn't a frustration; it was a teacher. Slowly, I learned to interpret stack traces, to identify edge cases, to isolate state inconsistencies, and to decouple tightly-bound components. Each bug, each failed attempt, added to my intuition. Each fix brought a small surge of pride.

From Code to Canvas: Conquering the GUI

One of the most profound learning curves came from working on the graphical user interface. At first, the JavaFX canvas felt like uncharted territory. But step by step, I figured out how to layer visuals over logic — how to animate node execution, create side panels that sync with model states, and enable users to create, edit, and execute workflows visually.

Connecting GUI events to back-end logic required more than just technical skill — it required patience, abstraction, and persistence. Each toolbar, each visual feedback element, and each color-coded node is a result of countless iterations and internal debugging.

A Surprising Fact: The Entire Project Was Built Alone

Perhaps the most surprising part of this journey is that every single line of code, every UML diagram, and every piece of UI logic was built by one person — myself. There was no team, no delegation, no external libraries for workflow logic. It was simply me, my IDE, and my commitment to learning.

The `MainViewController` alone exceeds 2000 lines of code — and yet, every method in it taught me something about responsibility, scope, or event-driven architecture. I discovered not only how to build a workflow engine — but how to structure a project from the ground up, debug it without shortcuts, and polish it until it felt complete.

From Student to System Architect

In the process of completing this project, I crossed a quiet milestone: I stopped thinking like a student, and began thinking like a software architect. I started to ask the right questions: *How can this be modular? How do I ensure scalability later? Can this node be reused in a different context? What if this becomes part of a microservice deployment?*

This project was my classroom, my challenge, and my breakthrough. It taught me not only how to apply the principles of OOP and UI development — but how to own an

idea from inception to implementation. I did not just learn how to program; I learned how to build.

A Final Note

If there's one message I could give to anyone reading this, it is this: great software is not built in a day. It is built through persistence, reflection, and iteration. This project took over two and a half months, countless revisions, and more setbacks than I can count. But it also gave me something I couldn't have imagined when I began: confidence in my ability to build something real — from scratch, with care, and with pride.

14 Conclusion

This project represents far more than an academic exercise; it is the culmination of vision, engineering, and determination. The **Machine Learning Workflow Orchestration System** I have developed began as a requirement to demonstrate object-oriented programming concepts — but grew into a sophisticated, modular, and scalable software product that reflects the rigor of real-world engineering.

From the very first line of code to the final execution animation, every feature in this system was crafted with intention. It integrates a **dynamic graphical user interface** where users can construct custom workflows by visually creating, naming, connecting, and executing color-coded nodes. Each node represents a computational stage — from data preprocessing and model training to inference, monitoring, and explainability — and the system brings them to life through animations, interaction, and logging.

What sets this project apart is not only the range of features, but the architectural discipline behind it. Every line of code, every visual component, and every logic path was written with maintainability, extensibility, and clarity in mind. The system is built on the firm foundation of **Object-Oriented Programming**:

- **Abstraction** provided the blueprint for all node types, making it easy to expand.
- **Encapsulation** shielded internal logic and protected the system from unintended interference.
- **Inheritance** ensured that shared behaviors could be reused without redundancy.
- **Polymorphism** enabled seamless execution, sidebar updates, and future flexibility — including the integration of generics, runtime polymorphism, and interface-based design.

The result is a clean, extensible architecture spanning over **30+ classes** and thousands of lines of Java code, with modular components that can grow independently — just like real software systems in production. The GUI, powered by JavaFX, includes:

- A fully-featured node creation dialog with live-editable fields
- Dynamic arrow connections with custom labels
- Execution path highlighting and visual animation
- Real-time sidebar synchronization for editing and review

- A scalable mini-map for intuitive navigation
- Execution logs with visual stickers for enhanced feedback

The technical challenges were substantial — from synchronizing UI updates, implementing live editing, to developing robust connection logic with undo/redo functionality. Every obstacle required experimentation, failure, and perseverance. But each hurdle also brought clarity, growth, and confidence.

As the project evolved, so did its ambition. What started as a university assignment transformed into the foundational phase of a much larger vision — to build a professional-grade orchestration platform capable of integrating future **microservices**, advanced AI modules, and cloud-based execution engines. This isn't just a coursework submission — it is the genesis of a tool that can grow alongside the evolving landscape of machine learning and software engineering.

This journey taught me far more than code. It taught me the essence of good design, the value of modularity, the power of abstraction, and the art of interface thinking. I've not only implemented OOP — I've lived it. I've seen it unfold in layers, evolve with decisions, and shape every component of this system.

A surprising fact? I began with uncertainty. I now end with mastery.
And this is not a final product — it is the foundation of something extraordinary.