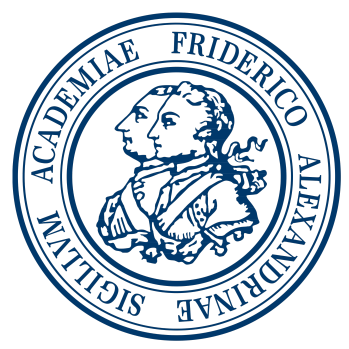
Engineering Workflow Orchestration Interfaces: A Full-Stack Approach to Scalable Pipeline Management, Validated through User-Centered Evaluation

MASTER THESIS

Timo Ricardo Meinhof

Submitted on 30 April 2026



Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Thomas Wolter, M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 30 April 2026

License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see `https://creativecommons.org/licenses/by/4.0/`

Erlangen, 30 April 2026

Abstract

As data processing pipelines grow in complexity, effective tools for monitoring and controlling their execution become increasingly important. Within the MECOIS platform, the Apache Airflow scheduler provides a user interface that is difficult for non-technical users to navigate and requires interaction through a separate interface rather than integrating into the system directly. This thesis aims to address these limitations by developing a custom user interface within the MECOIS system for orchestrating data-intensive tasks. We present the design and implementation of a user interface based on React and TypeScript to enable direct interaction with the Airflow API and the MECOIS backend. The solution supports triggering workflows, monitoring execution states, accessing logs, and managing task execution through retry and cancellation mechanisms. These functionalities are complemented by a graphical representation of workflow structures as directed acyclic graphs, along with mechanisms for efficient data retrieval, caching, and synchronization. We evaluate usability and effectiveness of the approach through a qualitative user study. This includes interactive demonstrations and interviews conducted to gather user feedback and identify areas for refinement. The results show improved user interaction and workflow management capabilities. Furthermore, the study revealed shortcomings in the initial implementation and informed subsequent improvements to the user interface.

Contents

1 Introduction **1**

2 Literature Review **5**
2.1 Workflow orchestration and pipeline automation research 5
2.2 User interfaces for workflow automation tools 8
2.3 Industry standards, accessibility requirements, and frontend best practices for workflow interfaces 11

3 Requirements **19**
3.1 Requirements context and derivation 20
3.1.1 System context and objective 20
3.1.2 Sources and derivation of requirements 20
3.1.3 Classification of requirements 21
3.2 Functional requirements 22
3.2.1 Workflow execution and control 22
3.2.2 Monitoring, inspection, and visualization 22
3.2.3 Optional product extensions 24
3.3 Quality requirements 24
3.3.1 Usability and accessibility 24
3.3.2 Performance and responsiveness 26
3.3.3 Maintainability and extensibility 26
3.4 Technical and architectural constraints 27
3.4.1 Platform and integration constraints 27
3.4.2 Architectural constraints 28
3.4.3 Dependency and version constraints 29
3.5 Security, quality assurance, and operational requirements 29
3.5.1 Security and authentication considerations 29
3.5.2 Testing and quality assurance requirements 30
3.5.3 Deployment, environment, and evaluation requirements . . 31

4 Architecture **33**
4.1 Architectural Context and Overview 33

4.1.1 Role of the Frontend within MECOIS 33
4.1.2 Overall System Architecture Overview 34
4.2 Frontend and Backend Integration Architecture 35
4.2.1 Interaction Between Frontend, Backend, and Airflow . . . 35
4.2.2 Backend Structure and Airflow Task Architecture 36
4.3 Authentication and API Client Architecture 37
4.3.1 Authentication and Authorization Flow 37
4.3.2 API Abstraction and Client Design 38
4.4 State Management and Data Flow Architecture 39
4.4.1 State Management with TanStack Query 39
4.4.2 Component Data Flow and Backend Communication . . . 39
4.4.3 Caching and Request Optimization 40
4.5 Technology Integration and Supporting Architectural Concerns . 40
4.5.1 Integration of Core Frontend Technologies 40
4.5.2 Error Handling and Logging Architecture 41
4.6 Architectural Decisions and Trade-Offs 41

5 Design and Implementation 43
5.1 Design Objectives and Implementation Approach 43
5.1.1 UI/UX Design Principles and Guidelines 43
5.1.2 General Implementation Strategy 45
5.2 Cross-Cutting Frontend Infrastructure 46
5.2.1 Reusable Component Architecture 46
5.2.2 Data Fetching and State Synchronization with TanStack Query 47
5.2.3 API Client Generation and Request Handling 48
5.2.4 Localization and Internationalization Support 49
5.3 Workflow Management Interfaces 49
5.3.1 Workflows Page 50
5.3.2 Workflow and Workflow Run Pages 53
5.3.3 Task Instances and Workflow Runs Tables 58
5.3.4 Variables Management Interface 63
5.3.5 Logs Interface 66
5.4 Advanced Interaction and Execution Support 68
5.4.1 DAG Visualization with ReactFlow 68
5.4.2 Handling Asynchronous Data and Polling 69
5.4.3 Error, Warning, and Status Representation 69
5.5 Testing and Backend Support 70
5.5.1 Frontend Testing Strategy and Implementation 70
5.5.2 Backend Adjustments Supporting the Frontend 70
5.5.3 Example Workflows and Demonstration Pipelines 70
5.6 Implementation Decisions and Trade-Offs 71

6 Evaluation **73**
6.1 Evaluation Goals and Research Questions 73
6.2 Evaluation Methodology 73
6.2.1 Study Design 73
6.2.2 Participant Selection and Recruitment 74
6.2.3 Task-Based Evaluation Scenarios 74
6.2.4 Data Collection and Analysis Approach 75
6.3 Comparative Evaluation Setup 76
6.3.1 Comparison with Airflow's Native Interface 76
6.3.2 Quantitative and Qualitative Evaluation Metrics 76
6.4 Evaluation Results 77
6.4.1 Usability and Effectiveness Results 77
6.4.2 Participant Feedback and Observed Themes 78
6.4.3 Interpretation of Findings 78
6.5 Threats to Validity and Limitations 80

7 Conclusion **83**
7.1 Limitations 85
7.2 Future work 85

Appendices **87**
A Tool Disclaimer 89
B Interview Summaries 89
B.1 Participant 1 90
B.2 Participant 2 92
B.3 Participant 3 95
B.4 Participant 4 98
B.5 Participant 5 101
B.6 Participant 6 104
B.7 Participant 7 107
B.8 Participant 8 110
B.9 Participant 9 113
B.10 Participant 10 115
B.11 Participant 11 118
B.12 Participant 12 120
B.13 Participant 13 123
B.14 Participant 14 125
B.15 Participant 15 127
B.16 Participant 16 129
B.17 Participant 17 132
B.18 Participant 18 134

References **137**

List of Figures

4.1 Overall System Architecture of the MECOIS Workflow Integration Frontend, Illustrating Frontend Components, Backend Tasks, Airflow Services, and Their Communication Paths 35
4.2 Backend and Airflow Task Architecture Showing Modular Task Composition, Pipeline Groupings, and Execution Coordination Through Airflow Services 37
4.3 Token-Based Authentication and Authorization Architecture Integrating Supabase Credential Storage, Persistent Client Tokens, and Airflow Application Programming Interface (API) Access . . 38

5.1 Compact Overview of the Workflows Page Showing Accordion-Based Workflow Listing and High-Level Scheduling Information . 50
5.2 Expanded Workflow View Providing Embedded Workflow Metadata, Reusable Workflow Details, and Direct Execution Controls 51
5.3 Workflow Detail Page Integrating DAG Visualization, Reusable Workflow Details, and Historical Workflow Run Overview 54
5.4 Interactive DAG Visualization Embedded in the Workflow Page for Structural Inspection of Task Dependencies 55
5.5 Reusable Workflow Details Component Embedded in the Dedicated Workflow View 56
5.6 Embedded Workflow Run History Supporting Navigation from Workflow Structure to Execution Inspection 57
5.7 Workflow Run Detail View Combining Task-State Visualization and Execution Metadata Inspection 58
5.8 Structured Workflow Run Metadata View Showing Timing, Trigger, State, and Configuration Information 59
5.9 Workflow Runs Table Supporting Filter-Driven Inspection of Historical Execution Instances 60
5.10 State-Based Filtering Interaction in the Workflow Runs Interface . 60
5.11 Date-Range Filtering for Temporal Restriction of Workflow Run Queries 61

5.12 Task Instances Table Supporting Task-Level Monitoring and Direct Log Access . . . 62
5.13 Variables Management Interface Supporting Structured Inspection and Editing of Airflow Variables . . . 63
5.14 Variable Creation Dialog Supporting Structured Input and JSON-Aware Value Entry . . . 64
5.15 Inline Validation Feedback for Invalid JSON Input During Variable Creation . . . 65
5.16 Selection-Based Bulk Actions Supporting Variable Export and Batch Operations . . . 66
5.17 Structured Logs Interface with Severity-Based Highlighting and Chronological Presentation . . . 67
5.18 Reusable Status Indicator Combining Iconography, Color, and Textual Semantics . . . 69
5.19 Snackbar Notification for Immediate Feedback on User Actions . . . 70
5.20 Icon with Tooltip Providing Contextual Information . . . 71
5.21 Copy-to-Clipboard Interaction for Efficient Identifier Handling . . . 71

6.1 Age distribution of the participants in the user study. The majority of participants were 25 years old, while the remaining participants were distributed across higher age groups. . . . 74
6.2 Comparison of participants' general software development experience and Apache Airflow familiarity across defined proficiency levels. While development experience is concentrated in the intermediate and advanced categories, familiarity with Airflow is more evenly distributed, with a noticeable proportion of participants reporting no or low prior exposure. . . . 75
6.3 Distribution of interview durations grouped into ten-minute intervals. Most interviews lasted between 40 and 79 minutes, with only one interview reaching the 80-minute interval. . . . 76
6.4 Participant perception of the stock and custom interfaces across navigation ease, cognitive load, and learnability. Aggregated scores show that the custom interface is consistently perceived as superior across all evaluated dimensions. . . . 77
6.5 Preferred user interface for daily use among participants. The majority of participants expressed a clear preference for the custom interface, while a small subset indicated that their preference depends on the specific context. No participant preferred the stock interface. . . . 78

6.6 Frequency of emergent themes across all participant interviews. Learnability and efficiency were identified in all interviews, while visibility and trust were also prominent. Debugging emerged less consistently but remained a significant theme. The dashed line indicates the total number of participants ($n = 18$). 79
6.7 Distribution of observed interaction styles during the user study. Exploratory and confident interaction patterns were most common, while hesitant and trial-and-error behaviors occurred less frequently. No participant exhibited sustained frustration during task execution. The dashed line indicates the total number of participants ($n = 18$). 79
6.8 Distribution of participant sentiment toward the custom interface. The majority of participants reported very positive or positive impressions, while the remaining participants expressed neutral sentiment. No negative or very negative perceptions were observed. 80

Acronyms

API	Application Programming Interface
DAG	Directed Acyclic Graph
UI	User Interface
ETL	Extract, Transform, Load
AWS	Amazon Web Services
HCI	Human-Computer Interaction
SUS	System Usability Scale
TAM	Technology Acceptance Model
SART	Situation Awareness Rating Technique
QUIS	Questionnaire for User Interaction Satisfaction
EAA	European Accessibility Act
BFSG	Barrierefreiheitsstärkungsgesetz
BFSGV	Verordnung zum Barrierefreiheitsstärkungsgesetz
WCAG	Web Content Accessibility Guidelines
W3C	World Wide Web Consortium
JSON	JavaScript Object Notation
WAI	Web Accessibility Initiative
ARIA	Accessible Rich Internet Applications
HTML	HyperText Markup Language
UX	User Experience
URL	Uniform Resource Locator

CI/CD Continuous Integration and Continuous Delivery/Deployment

1 Introduction

As software systems increasingly rely on recurring workflows for data extraction, transformation, and analysis, the orchestration of these processes becomes a central engineering concern. Modern data processing environments typically rely on workflows that coordinate multiple dependent steps, execute them in a defined order, and provide mechanisms for scheduling, monitoring, and failure handling. In such contexts, workflow orchestration is not just an operational convenience, but a prerequisite for building reliable and maintainable pipeline-based systems. This is particularly relevant in platforms such as MECOIS, where data must be collected from heterogeneous sources, processed through multiple stages, and made available for analysis in a reproducible and traceable manner. The increasing complexity of such workflows creates the need for interfaces that allow users to understand the pipeline structure, inspect execution states, and intervene when problems occur.

Within MECOIS, Apache Airflow has recently been introduced as the backend scheduling engine for managing these workflow-based processes. Airflow is a widely used orchestration system that models workflows as directed acyclic graphs (Directed Acyclic Graph (DAG)) and provides a programmable execution environment for defining, scheduling, and observing task-based pipelines. In the context of MECOIS, Airflow assumes the role of the operational backbone for orchestrated data processing: it schedules workflow runs, manages dependencies between tasks, tracks execution states, and exposes relevant workflow information through its Application Programming Interface (API). This makes Airflow a suitable technical foundation for backend orchestration, especially because it offers a mature scheduling model and a structured interface for interacting with workflows and their executions. However, at the same time, integrating such a scheduler into an existing software platform requires more than just backend functionality. It also requires a frontend that presents orchestration capabilities in a way that aligns with the surrounding system, the needs of its users, and the specific workflow scenarios of the project.

Although Airflow provides its own native user interface, this interface is not sufficient for the project-specific requirements of MECOIS. The native Airflow user

interface (User Interface (UI)) is designed as a general-purpose administration and monitoring tool for Airflow installations, but it is not tailored to the interaction patterns, terminology, integration requirements, and usability expectations of end users interacting with a dedicated application frontend. Relying on a separate external interface would fragment the user experience and force users to leave the MECOIS' environment whenever they need to trigger workflows, inspect executions, or review logs. In addition, the default UI does not encapsulate the full functionality desired by the MECOIS team, particularly with regard to seamless integration, workflow-specific visualizations, reusable project-wide frontend patterns, and an interaction model that can be adapted to the needs of data engineers and other technical stakeholders. This creates a disconnect between the robust orchestration capabilities provided by the backend scheduler and the user-facing interaction model available within the MECOIS platform.

The motivation for developing a custom frontend component follows directly from this gap. A dedicated MECOIS-integrated interface can consolidate workflow orchestration tasks within the existing platform, thus reducing context switching, improving usability, and providing functionality that is better aligned with project requirements. Instead of exposing users to the full complexity of Airflow's native interface, a custom frontend can focus on operationally relevant actions, such as triggering workflows, monitoring task and workflow states, inspecting logs, and managing failed executions. In addition, it can incorporate orchestration functionality into the design language, routing structure, and architectural conventions of the existing system. In an engineering context, this is particularly important because maintainability, consistent component design, state synchronization, accessibility, and testability are essential quality attributes of the final solution.

The practical relevance of this work lies in improving workflow orchestration for data engineers, pipeline owners, and other users of the MECOIS platform by providing clear visibility into dependencies, execution states, failures, and recovery options within an integrated frontend. By embedding these capabilities directly into the system, the solution enhances usability, consistency, and efficiency in day-to-day interaction with orchestration functionality, thereby extending beyond mere API integration. Accordingly, this thesis makes both research-oriented and engineering-oriented contributions, with a primary emphasis on the latter. From a research perspective, it examines how the functionality of a general-purpose workflow scheduler can be effectively integrated into a domain-specific frontend in a usable manner. From an engineering perspective, it presents the design and implementation of a React and TypeScript-based prototype that enables seamless interaction with the Airflow API and the MECOIS backend, supporting workflow triggering, execution monitoring, log access, and dependency visualization. Overall, the work constitutes an engineering thesis focused on the development, integration, and evaluation of a practical software artifact.

The scope of the thesis is bounded accordingly. It focuses on the conception, implementation and evaluation of a prototype frontend for the orchestration of workflow within MECOIS. The work does not seek to replace Apache Airflow as the orchestration backend nor does it attempt to redesign the entire MECOIS platform. Instead, it concentrates on the frontend-facing integration layer between users, the MECOIS system, and the Airflow-based scheduling infrastructure. Likewise, while the work includes evaluation and draws conclusions regarding usability and effectiveness, it does not aim to establish generalizable scientific laws about workflow interfaces. Its findings are grounded in a concrete system context and are primarily intended to inform the design and assessment of the implemented solution.

Methodologically, the thesis follows a typical engineering process consisting of design, implementation, and evaluation. It begins with an analysis of the problem context and requirements for workflow orchestration within MECOIS. Based on these requirements, a frontend concept and a supporting architecture are developed that define how workflow information should be presented, how user interactions should be structured, and how the frontend should communicate with backend services and the Airflow API. This concept is then realized as a prototype implementation using the selected frontend technologies and libraries. Finally, the resulting artifact is evaluated with regard to usability and effectiveness, including comparison with the native Airflow interface and the collection of user feedback to identify strengths, shortcomings, and opportunities for refinement.

The remainder of this thesis is structured as follows. Following the literature review in Chapter 2, the thesis addresses the requirements, architecture, design and implementation, and evaluation of the developed frontend in Chapters 3 to 6. The final chapter summarizes the main findings, discusses limitations, and outlines implications for future work.

2 Literature Review

2.1 Workflow orchestration and pipeline automation research

Workflow orchestration engines are software systems designed to coordinate, monitor, and recover complex workflows in distributed environments. Recent literature describes these tools as a response to the increasing complexity of modern software systems and the corresponding need to manage recurring processes with reliability, scalability, and fault tolerance [1]. In practice, they enable engineers to define, execute, and monitor dependent tasks in a structured manner while abstracting recurring infrastructural concerns such as state management, dependency handling, retries, and failure recovery [1]. This makes workflow orchestration especially relevant in data-intensive environments, where recurring extraction, transformation, and analysis processes must be executed in a reproducible and operationally manageable form.

The practical relevance of workflow orchestration extends beyond data pipelines alone. Comparative research shows that orchestration engines are used in Extract, Transform, Load (Extract, Transform, Load (ETL)) scenarios, microservice coordination, business process automation, event-driven applications, and cloud-native deployment settings [1]. In the context of data engineering, they are particularly useful because they structure recurring processing steps, reduce manual intervention, and provide explicit mechanisms for monitoring and recovery. As a result, workflow orchestration has become a core infrastructural concern in modern data platforms rather than a peripheral implementation detail.

A central concept in many orchestration systems is the directed acyclic graph. According to the official Apache Airflow documentation, Airflow represents a workflow as a DAG composed of individual tasks and their dependencies [2]. The DAG specifies the order in which tasks execute and defines the retry and dependency structure of the workflow [3]. This graph-based model is well suited to recurring processing pipelines because it makes control flow explicit, prevents circular dependencies, and supports both execution planning and visual inspection.

In this sense, the DAG is not merely a visual abstraction, but the operational core of many orchestration systems.

The DAG model is closely connected to scheduling. Airflow describes its own architecture as consisting of multiple components, including a scheduler, an executor, a webserver, DAG files, and a metadata database [2]. The scheduler is responsible for triggering scheduled workflows and submitting tasks for execution, while the metadata database stores workflow and task state information [2]. This architecture shows that workflow orchestration is not limited to sequencing tasks. Rather, it combines workflow definition, execution control, persistence, observability, and user interaction within a broader operational framework. Such a combination is one reason why orchestration platforms have become foundational tools in production data systems.

An important development in this field is the shift toward ”workflows as code”. Instead of relying exclusively on graphical modeling environments, modern orchestration tools define workflows programmatically. In Airflow, DAGs are written in Python and can express tasks, dependencies, scheduling, retries, and additional operational parameters in code [3]. This code-centric approach is significant from an engineering perspective because it supports version control, code review, testing, parameterization, and integration into existing software development processes. Workflow definitions thereby become first-class software artifacts rather than external configurations.

Within this broader landscape, Apache Airflow occupies a particularly prominent position. Airflow is widely used in data engineering because of its mature DAG-based model, broad ecosystem, and extensive integration support. At the same time, comparative work points out that workflow orchestration has diversified considerably, and that Airflow now coexists with several alternatives that reflect different design priorities [1]. This diversification is important for the present thesis because it shows that Airflow is not the only possible orchestration backend and that its strengths and limitations should be understood relative to other platforms.

One notable alternative is Prefect. According to the official Prefect documentation, flows are defined as decorated Python functions that behave largely like ordinary Python functions while adding workflow-specific capabilities such as execution tracking, retries, timeout handling, and deployment support [4]. Compared with Airflow, this model places stronger emphasis on a lightweight Python-first experience and on dynamic workflow behavior. Prefect therefore illustrates a design direction that retains code-based workflow definition while attempting to reduce some of the operational and conceptual overhead traditionally associated with Airflow.

Dagster represents a different design orientation. Its official documentation de-

scribes Dagster as a data orchestrator built for data engineers, with integrated lineage, observability, a declarative programming model, and strong testability support [5]. The documentation foregrounds assets as a central abstraction and illustrates workflows in terms of asset definitions and dependencies rather than only as generic task chains [5]. This shifts the focus from task execution alone toward the structure and lifecycle of data products. In contrast to Airflow's largely task-centric view, Dagster therefore reflects a more explicitly data-centric approach to orchestration.

Luigi provides yet another perspective on the design space. The Luigi documentation describes tasks as the place where execution occurs and states that tasks depend on each other and produce output targets [6]. Dependencies are declared programmatically through the `requires()` method, after which Luigi resolves the dependency structure and executes the workflow accordingly [6]. This makes Luigi a comparatively lightweight and straightforward orchestration framework, especially for Python-based environments. However, its conceptual and operational model is generally more minimal than that of Airflow, particularly with regard to broader observability, ecosystem breadth, and modern platform capabilities.

Beyond these Airflow-adjacent alternatives, the research literature also discusses orchestration systems based on substantially different execution models. A recent comparative study examines Apache Airflow, Netflix Conductor, Temporal, and Amazon Web Services (AWS) Step Functions together and highlights that these systems are shaped by distinct architectural assumptions [1]. In that study, Airflow is presented as a DAG-based orchestration engine optimized for ETL and batch operations, Conductor as a state-machine-based orchestrator for microservice coordination, Temporal as a durable execution system, and AWS Step Functions as a managed orchestration service built on state machines [1]. This comparison is important because it demonstrates that the broader workflow automation field extends well beyond DAG scheduling for batch data pipelines.

Overall, the literature presents workflow orchestration as a mature but still evolving field. Apache Airflow remains one of its most influential systems because its DAG-based and code-centric model has become highly established in data engineering practice [2], [3]. At the same time, alternatives such as Prefect, Dagster, and Luigi show that different orchestration platforms prioritize different concerns, including dynamic execution, data-centric abstractions, simplicity, testability, and observability [4], [5], [6]. For the present thesis, this body of work provides the conceptual foundation for positioning Airflow as a suitable backend scheduler while also motivating the need to examine how such orchestration functionality can be integrated into a project-specific frontend context.

2.2 User interfaces for workflow automation tools

Research on workflow automation systems indicates that the user interface should not be treated as a secondary layer added after process logic has already been defined. Instead, interface design is closely intertwined with process modeling, user roles, and the execution context of workflow information systems. Guerrero-García argues that the development of user interfaces for workflow information systems should be approached as an evolutionary process in which workflow, task, domain, and context-of-use models jointly inform the resulting interface [7]. Likewise, Yongchareon et al. propose a framework that derives user-interface flow models directly from artifact-centric business process models, explicitly linking business processes, user roles, and interface structures [8]. Taken together, this literature suggests that workflow frontends are not merely presentational shells, but integral operational components through which users inspect, understand, and influence process execution.

This is particularly relevant for workflow automation tools because they expose complex operational states to users. Unlike conventional information systems that primarily support data entry or retrieval, orchestration interfaces must support monitoring, exception handling, dependency inspection, and interaction with long-running asynchronous processes. Existing workflow-oriented interface research therefore places emphasis on model-driven design, role awareness, and adaptation to incomplete or evolving requirements [7], [8]. For the present thesis, this means that the frontend cannot be reduced to a thin wrapper around backend endpoints. Rather, it must be understood as a process-aware interface whose structure and behavior are shaped by the nature of workflow execution itself.

A second strand of literature concerns dashboards and monitoring-oriented interfaces more broadly. Systematic reviews in the dashboard literature show that usability problems often arise not because data are unavailable, but because the interface presents them in a form that is difficult to navigate, interpret, or align with concrete work tasks. Rabiei and Almasi identify functional requirements such as reporting, customization, reminders, alert creation, and tracking, as well as non-functional requirements such as speed, ease of use, integration, and up-to-date data presentation [9]. In a systematic review of dashboards in aged care, Siette et al. likewise report that user acceptance is influenced less by dashboard type than by how information is displayed and how well the capabilities of the interface match user needs; simple visual elements, interactive displays, and reporting capabilities were associated with comparatively high acceptability, whereas more complex visual forms were less well received [10]. These findings are directly relevant to orchestration frontends, which similarly need to surface operationally important information without overwhelming users with unnecessary detail.

A central challenge in such interfaces is cognitive load. Ke et al. show that ex-

cessive information density in dashboards can increase cognitive load, distract attention from critical information, and impair performance during monitoring tasks [11]. Their study further demonstrates that users differ in how efficiently they process dashboard information and that information load interacts with cognitive style, which implies that interface complexity should not be treated as a purely aesthetic issue [11]. In workflow automation interfaces, where users often need to diagnose failures, inspect dependencies, and react under time pressure, this is especially important. The design implication is not simply that less information is always better, but rather that information should be structured in a way that supports selective attention, prioritization, and rapid orientation.

Related work on adaptive and guided interfaces provides further support for this view. Smereka et al. investigate adaptive user interfaces for workflow management systems and show that interface-level recommendations can guide users toward appropriate next actions, thereby improving efficiency, facilitating training, and supporting decision-making [12]. Akiki's CHAIN approach similarly argues that adaptive interfaces require contextual help that remains useful even when the interface changes at runtime, because static help materials quickly become misaligned with adaptive or dynamically reconfigured interfaces [13]. More general Human-Computer Interaction (Human-Computer Interaction (HCI)) research on onboarding also points in the same direction: Lee et al. show that walkthroughs, product tours, and tooltips function as visual guides that help first-time users understand application features and adapt to the interface more effectively [14]. Although such studies are not specific to workflow orchestration, they indicate that user guidance mechanisms are not optional embellishments. In complex technical systems, they can be a necessary means of reducing friction, clarifying available actions, and improving the learnability of the interface.

Another important issue is visual hierarchy. Monitoring interfaces must continuously communicate which information deserves immediate attention and which information is merely contextual. Still's empirical study of web page visual hierarchy shows that initial entry points in an interface are predicted most strongly by position, color, and text style, rather than by size or images alone [15]. This is directly relevant to orchestration dashboards, in which failed runs, blocked dependencies, retries, and actionable warnings should be visually prioritized over stable background information. In practical terms, the literature suggests that visual hierarchy should be used deliberately to guide the user's focus toward the most critical workflow states instead of presenting all states with equal salience.

Color is one of the most common means of creating such salience, but the literature advises caution. Andersen and Maier show that individual colors differ significantly in their ability to guide attention and that these differences become more pronounced as visual complexity increases [16]. At the same time, Ma et al. demonstrate that an excessive quantity of colors can itself contribute to per-

ceptual overload and increased cognitive burden in complex interfaces [17]. This implies that color can be effective for communicating status, urgency, or warning conditions, but only when used selectively and within a coherent visual scheme. In a workflow interface, overuse of many competing status colors may weaken rather than strengthen a user's ability to recognize critical states quickly.

For this reason, the literature also supports the use of redundant cues such as icons, labels, and textual explanations. Jin et al. show that icon semantics significantly affect visual search performance and that the relationship between icons and text depends on layout and semantic familiarity [18]. High semantic familiarity improves search performance, while the presence or absence of text can either support or hinder recognition depending on context [18]. The broader implication is that color should not function as the sole carrier of meaning. Instead, effective technical interfaces should combine color with semantically clear icons, labels, or textual state descriptions so that important states remain interpretable under different viewing conditions, experience levels, and cognitive strategies.

The literature also points toward customization and task-focused interaction as important design principles. Rabiei and Almasi identify customization as a recurring functional requirement in dashboard systems [9]. Siette et al. similarly find that interactive displays and reporting capabilities contribute to dashboard acceptability [10]. In a more concrete dashboard study, Dowding et al. describe an iterative design process in which nurses were involved through contextual inquiry, rapid prototype feedback, and formal usability evaluation; the resulting dashboard supported switching between display options and viewing trends over time in ways that users found immediately usable and useful [19]. This literature suggests that interfaces become more usable when they allow users to adapt views to their tasks, reduce irrelevant information, and access the most operationally relevant perspective without unnecessary navigation overhead.

Finally, the literature provides useful guidance for evaluation methodology. A systematic review by Almasi et al. shows that dashboard studies commonly use established instruments such as the System Usability Scale (System Usability Scale (SUS)), the Technology Acceptance Model (Technology Acceptance Model (TAM)), the Situation Awareness Rating Technique (Situation Awareness Rating Technique (SART)), the Questionnaire for User Interaction Satisfaction (Questionnaire for User Interaction Satisfaction (QUIS)), and Health-ITUES [20]. At the same time, qualitative methods remain central. Rossi et al. explicitly recommend iterative dashboard design and evaluation, including qualitative interviews and think-aloud exercises, as part of a human-centered development process [21]. Concrete dashboard studies support this recommendation. Dowding et al. combine contextual inquiry, prototype feedback, usability evaluation, SUS, and QUIS in an iterative design process [19], while de Lusignan et al. use think-aloud evaluation with predefined tasks and show that verbalized interaction data can

reveal actionable usability problems in dashboard design [22]. For the present thesis, these methods are especially relevant because they align well with the evaluation of a workflow orchestration frontend intended for technical users performing concrete monitoring and control tasks.

Overall, the literature shows that user interfaces for workflow automation tools should be understood as process-aware and cognitively demanding systems that require deliberate design. Workflow-specific interface research emphasizes the tight coupling of process models, user roles, and interface flows, while broader HCI and dashboard research highlights the importance of visual hierarchy, guided interaction, controlled information density, semantically meaningful cues, customization, and iterative evaluation [7], [8], [9], [10], [11], [15], [20]. These findings provide the conceptual basis for designing an orchestration frontend that is not only functionally connected to workflow execution, but also usable, comprehensible, and effective in operational practice.

2.3 Industry standards, accessibility requirements, and frontend best practices for workflow interfaces

For an engineering thesis that develops and evaluates a concrete frontend artifact, official standards, regulations, and framework documentation are not merely supplementary sources, but part of the primary normative basis for design decisions. This is especially true when the artifact must satisfy legal accessibility expectations, align with established web standards, and integrate correctly with specific frontend technologies. In this context, official guidance is relevant not because it replaces academic literature, but because it defines the requirements, constraints, and implementation semantics that a production-oriented solution must actually satisfy.

Within the European Union, the central regulatory backdrop is Directive (EU) 2019/882, commonly referred to as the European Accessibility Act (European Accessibility Act (EAA)). The directive aims to harmonize accessibility requirements for certain products and services in order to reduce internal-market barriers caused by divergent national legislation [23], [24]. According to the official EUR-Lex summary, the legislation has applied since 28 June 2025 and covers, among other things, e-commerce as well as certain website and mobile-service elements in transport-related services [23]. For web-based workflow interfaces, this is highly relevant because accessibility is therefore not only a matter of usability or inclusiveness, but also part of a broader regulatory environment for digital services in Europe.

In Germany, this European framework is implemented through the *Barrierefreiheitsstärkungsgesetz* (Barrierefreiheitsstärkungsgesetz (BFSG)) and the associated *Verordnung zum Barrierefreiheitsstärkungsgesetz* (Verordnung zum Barrierefreiheitsstärkungsgesetz (BFSGV)). The BFSG explicitly states that products and services must be accessible, and it defines accessibility in terms of being findable, accessible, and usable for people with disabilities in the generally usual way, without particular difficulty and in principle without external assistance [25]. The BFSGV further specifies that it serves the implementation of Annex I of Directive (EU) 2019/882, entered into force on 28 June 2025, and requires compliance with the *Stand der Technik* while also mandating publication of the most relevant standards by the federal accessibility office [26]. As a consequence, accessibility in the implementation of a workflow frontend should be treated as a binding engineering requirement rather than as an optional quality attribute.

At the technical level, the central reference framework for web accessibility remains the Web Content Accessibility Guidelines (Web Content Accessibility Guidelines (WCAG)) 2.2. World Wide Web Consortium (W3C) describes WCAG as the international standard for making web content more accessible and notes that WCAG 2.2 was published as a W3C Recommendation on 5 October 2023 and later approved as ISO/IEC 40500:2025 [27], [28], [29]. For a workflow orchestration frontend, WCAG 2.2 is therefore the most appropriate benchmark because it is internationally recognized, technology-oriented, and directly applicable to web applications rather than only to static websites [27]. In an implementation-oriented thesis, this makes WCAG the central normative frame for accessibility-related frontend decisions.

W3C organizes WCAG around four foundational principles: content and interface components must be perceivable, operable, understandable, and robust [30]. These principles are especially useful in the present context because they map well to typical workflow-interface concerns. Perceivability affects the presentation of task states, logs, and status messages; operability affects navigation, focus handling, and keyboard usage; understandability affects labels, warnings, and action clarity; and robustness affects compatibility with assistive technologies and future user agents [30]. Rather than functioning as abstract terminology only, these principles provide a practical structure for reasoning about concrete frontend implementation choices.

WCAG further distinguishes between conformance levels A, AA, and AAA. The WCAG 2.2 specification states that Level A is the minimum level, Level AA requires satisfaction of all Level A and Level AA success criteria, and Level AAA additionally includes all AAA criteria; W3C also notes that requiring full Level AAA conformance as a general policy is not recommended for entire sites [31], [32]. For web applications in practice, this makes Level AA the most realistic and defensible target. It combines a strong accessibility baseline with practical feasibility and

is also the conformance level that many organizations aim to meet [32]. Accordingly, Level AA is the most appropriate reference point for the implementation of the workflow frontend in this thesis.

Keyboard accessibility forms one of the most basic requirements in this area. W3C states in Guideline 2.1 that all functionality should be available from a keyboard [33]. This requirement is especially important for technical dashboards and workflow tools, because such interfaces frequently include tables, action menus, filters, dialogs, graph elements, and other interaction patterns that are often implemented in JavaScript-heavy ways. If triggering workflows, inspecting runs, navigating logs, or controlling visualizations requires pointer-only interaction, the interface fails a core accessibility expectation and also becomes less robust for power users and assistive technologies [33], [34].

Keyboard accessibility alone is not sufficient unless focus behavior is also coherent. WCAG requires a focus order that preserves meaning and operability and additionally requires a visible focus indicator for keyboard-operable user interfaces [35], [36]. For workflow frontends, this has direct implications for complex interfaces with sidebars, accordions, data tables, filters, and detail views. Focus movement must follow a logical operational sequence, and users must always be able to determine which control currently has focus [35], [36]. This is particularly important in interfaces where multiple actions such as retry, cancel, inspect, expand, or delete may be available in quick succession.

Focus management becomes even more critical when the interface uses overlays, dialogs, or dynamically inserted content. The Web Accessibility Initiative (WAI)-Accessible Rich Internet Applications (ARIA) Authoring Practices Guide specifies that modal dialogs keep focus within their own tab sequence, that `Tab` and `Shift+Tab` must not move focus outside the dialog, and that focus should move to an element inside the dialog when it opens [37]. The same pattern also recommends that dialogs include a visible control for closing them [37]. In practice, this means that confirmation dialogs for destructive workflow actions, parameter-entry modals for triggering runs, or log-detail overlays must not merely appear visually; they must manage keyboard focus explicitly and return users to a sensible place in the workflow after closing [35], [37].

This is especially relevant for destructive or high-impact actions. Official GOV.UK guidance recommends warning buttons only for actions that should cause users to think carefully before proceeding and advises that they should be used sparingly [38]. The same guidance also addresses duplicate submissions and recommends explicit prevention of accidental double clicks where user research shows that repeated activation causes errors [38]. Applied to workflow orchestration interfaces, this supports a design in which operations such as deleting runs, canceling executions, or triggering mass changes should be clearly labeled, separated from routine actions, and protected by explicit confirmation and recoverability mech-

anisms [37], [38].

When custom widgets are necessary, WAI-ARIA authoring practices become important, but they should not be treated as a substitute for semantic HTML. W3C's "first rule of ARIA" states that if a native HTML element or attribute already provides the required semantics and behavior, it should be used instead of repurposing other elements with ARIA roles and properties [34]. The APG further emphasizes that "no ARIA is better than bad ARIA" because incorrect ARIA can misrepresent the interface to assistive technologies [39]. This is an important constraint for a React-based workflow frontend: ARIA should be used deliberately for genuinely custom controls, but native buttons, tables, form elements, headings, and navigation landmarks should remain the default basis whenever possible [34], [39].

Consistent with this principle, forms and controls require proper labeling and instruction. W3C's forms tutorial states that labels should be provided for all form controls and that, in most cases, this should be done using the native `label` element [40]. A related WAI tutorial further recommends that forms provide instructions covering required or optional inputs, formats, and other relevant guidance, and explains that these instructions must be available in ways assistive technologies can read aloud [41]. For workflow orchestration interfaces, these requirements apply not only to classical forms, but also to trigger-parameter dialogs, filter controls, search inputs, and bulk-action interfaces. Clear labeling and instruction are particularly important where users must provide JavaScript Object Notation (JSON) parameters, select execution ranges, or interpret system-specific options under time pressure.

Tabular structures deserve special attention because workflow interfaces frequently display task states, run histories, variables, and log summaries in table-like views. WAI guidance on tables notes that captions help users find and understand tables and function much like headings for tabular content [42]. At the same time, the APG distinguishes between static tables and interactive grids: when content is tabular but not interactive, native HTML tables are preferred; when keyboard-managed interaction inside the structure is required, the grid pattern becomes appropriate [43], [44]. This distinction is highly relevant for workflow frontends. A simple run-history overview may be best represented as an HTML table, whereas a fully interactive data grid with sorting, selection, or in-cell actions may require more elaborate keyboard management and ARIA roles [43], [44].

Color is another area where official accessibility guidance is directly applicable. WCAG requires that color not be the only visual means of conveying information, indicating an action, prompting a response, or distinguishing a visual element [45]. This has immediate implications for the visual representation of workflow states such as queued, running, failed, succeeded, paused, or warning. Color can still be valuable for rapid scanning, but it must be reinforced through icons,

text labels, patterns, or other cues so that state information remains available to users who cannot reliably perceive color differences [45]. In addition, the WCAG framework and design-system guidance on buttons and interfaces reinforce that sufficient contrast must be maintained for readable and operable controls [31], [38].

Information architecture and navigation efficiency should likewise be understood through official guidance rather than through simplistic rules such as a universal maximum click depth. GOV.UK guidance on service navigation explicitly recommends simplifying the user journey first and only then introducing navigation where the service is used repeatedly, involves multiple tasks, or lacks a single linear journey; it also advises that navigation should expose only the most important top-level sections rather than function as a full site map [46]. Similarly, GOV.UK pagination guidance states that content should only be split across pages if this improves performance or usability and that pagination should not be used for linear journeys where conventional next/back controls are more appropriate [47]. For this thesis, the implication is that "click depth" should be treated as part of broader navigation quality: the goal is not to optimize toward a fixed number of clicks, but to reduce friction, preserve findability, and align navigation structures with user tasks [46], [47].

Several common navigation components also have well-defined accessibility expectations. The APG describes tabs as a pattern in which one panel is displayed at a time and provides keyboard conventions for moving between tab elements and associated tab panels [48]. GOV.UK guidance adds practical advice that tab labels should be clear, the first tab should reflect the most commonly needed section, disabled tabs should generally be avoided, and the current tab should be reflected in the URL fragment so browser navigation behaves predictably [49]. Breadcrumbs, by contrast, are explicitly intended to help users understand their place within a website or web application hierarchy [50]. These patterns are directly applicable to workflow interfaces, where tabs may separate overview, graph, and logs views, and breadcrumbs may help users move between workspace, workflow, run, and task detail levels without losing context [48], [49], [50].

Error prevention, error feedback, and recoverability are similarly central in workflow frontends because user actions can have operational consequences. Official button guidance recommends action-oriented labels such as "Save and continue" and warns against competing primary actions on the same page [38]. For a workflow interface, this suggests that trigger, retry, cancel, and delete actions should be phrased unambiguously and visually prioritized according to their operational importance. Where an action is destructive or expensive, confirmation should not be treated as a decorative pattern, but as part of the interface's error-prevention strategy [37], [38].

Feedback for asynchronous operations requires equally careful treatment. WCAG 2.2's

status-message guidance explains that assistive technologies should be able to notify users about relevant changes in content even when those messages do not receive focus [51], [52]. WAI techniques further show that progress indicators often need an accompanying live region because a `progressbar` role alone is not a live region and may therefore not be announced as it changes [53]. In a workflow orchestration frontend, this affects loading indicators, polling-based run updates, stale-data notices, and completion or failure notifications. Messages such as "loading runs", "workflow triggered", "retry scheduled", or "connection lost" should therefore be exposed not only visually, but also programmatically as status information where appropriate [51], [52], [53].

For the concrete React implementation, official React guidance provides useful structural principles. React's documentation on sharing state emphasizes "lifting state up" and maintaining a "single source of truth" for each unique piece of state [54]. Its guidance on state structure further recommends grouping related state, avoiding contradictions in state, and avoiding redundant state that can be derived during rendering [55]. These principles are highly relevant in a workflow frontend, where multiple views may simultaneously depend on shared workflow, run, task, and selection state. Predictable component structure therefore depends not only on decomposition into reusable components, but also on disciplined state ownership and avoidance of duplicated, divergent client-side state [54], [55].

TypeScript contributes an additional layer of robustness. The TypeScript documentation states that the `strict` compiler option enables a wide range of stricter type-checking behaviors that provide stronger guarantees of program correctness [56]. The handbook on narrowing further explains how control-flow-based type narrowing and exhaustive handling via `never` support safer branch logic in union-typed code [57]. In an API-driven frontend, this matters because workflow states, task states, request results, optional fields, and error payloads often have non-trivial shape variation. Strong typing and disciplined narrowing can therefore reduce whole classes of integration errors and improve maintainability when backend contracts evolve [56], [57].

For server-state management, TanStack Query's official guidance is similarly relevant. The documentation describes its defaults as "aggressive but sane", explains that query data are considered stale by default, and notes that stale queries may refetch automatically when components mount, when the window regains focus, or when the network reconnects [58]. It also explains that failed queries are retried by default and that targeted invalidation with `invalidateQueries` marks data as stale and may trigger background refetching [58], [59]. These behaviors are directly applicable to workflow interfaces that poll execution state or need to react to mutations such as triggering a run or canceling a task. Good implementation practice therefore requires explicit decisions about `staleTime`, polling intervals, invalidation strategy, retry behavior, and the user-facing com-

munication of stale or updating data [58], [59].

React Flow, which is central to graphical DAG visualization in this thesis, also provides specific guidance relevant to both accessibility and performance. Its documentation states that nodes and edges are keyboard-focusable and operable by default, that keyboard selection and movement are supported, and that focused nodes can automatically be panned into view [60]. At the same time, its performance guidance warns that unnecessary re-renders are a major source of performance issues and recommends memoizing components, functions, arrays, and objects passed to `<ReactFlow />` as well as avoiding direct dependence on frequently changing node and edge collections where possible [61]. These recommendations are important because DAG visualizations in workflow tools are not merely decorative; they are interactive and potentially state-heavy components whose accessibility and responsiveness significantly affect the overall usability of the frontend [60], [61].

Finally, consistency and reusability should be treated as architecture-level concerns rather than isolated stylistic preferences. Accessibility patterns, button semantics, tab behavior, breadcrumb structure, and status-message handling all become more reliable when implemented through shared components and design-system conventions rather than through repeated ad hoc solutions [38], [48], [50]. For an engineering thesis, this also justifies drawing explicitly on official framework documentation: such sources define the operational semantics of the technologies used, the normative requirements of the web platform, and the legally and technically relevant constraints under which the artifact is built [27], [34], [54], [56], [58], [60]. In this sense, the implementation of a workflow frontend should be guided by a combination of accessibility law, web standards, and framework-specific best practices that together support a solution that is maintainable, robust, and usable in practice.

3 Requirements

This chapter derives and formulates the requirements for the frontend artifact developed in this thesis. The purpose of the chapter is not merely to restate a feature list, but to translate the problem context of MECOIS, the findings of the literature review, and the technical project environment into a structured set of functional requirements, quality requirements, constraints, and operational expectations. In an engineering thesis, this step is essential because the requirements chapter defines the criteria against which subsequent architectural decisions, implementation choices, and evaluation activities can be justified.

The artifact addressed in this thesis is a frontend component integrated into the existing MECOIS platform that enables users to interact with workflow orchestration functionality backed by Apache Airflow. Consequently, the requirements of the artifact are shaped by two perspectives at the same time. On the one hand, the frontend must provide operationally useful capabilities for triggering workflows, monitoring execution, inspecting failures, and understanding dependency structures. On the other hand, it must satisfy broader engineering expectations concerning usability, accessibility, maintainability, integration, and testing. The requirements defined in this chapter therefore combine user-facing expectations with project-specific technical and organizational constraints.

The chapter is structured accordingly. It first explains the context and derivation of the requirements. It then defines the functional requirements of the frontend, followed by the main quality requirements. After that, the chapter discusses the technical and architectural constraints under which the artifact must be built. Finally, it formulates the security-related, quality-assurance-related, deployment-related, and evaluation-related requirements that complete the overall specification.

3.1 Requirements context and derivation

3.1.1 System context and objective

The objective of the artifact is to provide a project-specific frontend for workflow orchestration within MECOIS. More precisely, the frontend is intended to enable users to interact with Airflow-managed workflows directly inside the existing MECOIS environment instead of relying on the native Airflow interface alone. This objective follows directly from the central problem identified in the introduction: while Airflow provides strong orchestration capabilities at the backend level, its default user interface does not satisfy the integration, interaction, and usability expectations of the MECOIS project context.

The requirements of this chapter must therefore be understood in relation to a specific scope. The thesis does not seek to replace Apache Airflow as orchestration backend, nor does it attempt to redesign the entire MECOIS platform. Instead, it focuses on the design and implementation of a frontend-facing integration layer that exposes relevant orchestration capabilities in a form that is more appropriate for the system context and the intended user group. The resulting requirements must thus be specific enough to guide the construction of a working artifact, while remaining narrow enough to stay within the defined thesis scope.

A second contextual aspect is that the artifact is a prototype intended for realistic use and evaluation rather than an isolated mock-up. This means that the requirements cannot be restricted to visual interface design or nominal feature coverage alone. They must also address qualities such as responsiveness, robustness, accessibility, testability, and compatibility with the existing technical environment. For this reason, the requirements chapter combines product requirements with technical and process-oriented requirements that are necessary for a credible engineering implementation.

3.1.2 Sources and derivation of requirements

The requirements defined in this chapter are derived from several complementary sources. The first source is the problem context established in the introduction, especially the role of Airflow as orchestration backend and the need for an integrated frontend solution within MECOIS. The second source is the literature review in Chapter 2, which showed that workflow frontends should be treated as integral components of workflow information systems rather than as superficial presentation layers [7], [8]. The literature review also highlighted that dashboard-like monitoring interfaces must be designed in a way that supports selective attention, reduces cognitive burden, and aligns visual presentation with operational relevance [9], [10], [11].

The third source is the regulatory and standards-based context discussed in the literature review. Accessibility requirements in particular cannot be treated as optional refinements, because the implementation of digital services within the European and German context is shaped by the European Accessibility Act, the *Barrierefreiheitsstärkungsgesetz*, and the Web Content Accessibility Guidelines (WCAG) as the central technical benchmark [24], [25], [26], [27], [31]. The fourth source is the technical environment of the project itself. The frontend must operate within the existing MECOIS architecture, interact with an Airflow 3.x backend through stable interfaces, and align with the React- and TypeScript-based frontend environment adopted by the project.

Finally, the requirements are also informed by the intended evaluation of the artifact. Since the thesis includes a user-centered evaluation and comparison with the native Airflow interface, the artifact must support realistic usage scenarios, expose operationally meaningful functionality, and be stable enough to serve as the basis for qualitative assessment. Accordingly, the requirements in this chapter are not only design-time aspirations, but also the conditions under which the implemented artifact can later be evaluated in a meaningful way.

3.1.3 Classification of requirements

To make the chapter methodologically clear, the requirements are divided into four categories. The first category comprises *functional requirements*. These describe what the frontend must enable users to do, such as triggering workflows, viewing task states, or inspecting logs. The second category comprises *quality requirements*. These specify the qualities the frontend must exhibit, including usability, accessibility, responsiveness, and maintainability. The third category comprises *technical and architectural constraints*. These do not define new user-visible features, but limit the design space within which the solution must be implemented. The fourth category comprises *security, quality assurance, and operational requirements*, which address safe usage, development quality, deployment readiness, and the support of later evaluation.

This classification also requires two clarifications. First, test-driven development and high test coverage are not treated as functional requirements, because they do not describe externally visible system behavior. Instead, they are classified under quality assurance, where they belong as process-oriented and implementation-oriented expectations. Second, potential changes to backend or Airflow-related structure are not formulated as mandatory product functionality, but as optional extensions. This distinction is important because the thesis primarily concerns the frontend artifact, while backend-facing restructuring may be useful but is not part of the minimal functional core.

3.2 Functional requirements

3.2.1 Workflow execution and control

The primary functional purpose of the artifact is to enable direct interaction with orchestrated workflows from within the MECOIS frontend. At the most basic level, this means that users must be able to trigger workflows without leaving the platform context. Workflow triggering is central because the frontend is not intended merely as a passive monitoring surface, but as an operational interface through which users can initiate relevant processing activity.

Beyond basic triggering, the frontend must expose workflow-control actions that are meaningful in the project context and supported by the surrounding backend and Airflow integration. Such control capabilities may include the initiation of new workflow runs, the passing or configuration of input parameters where applicable, and run-related actions such as retrying or canceling execution if these actions are available through the relevant interfaces. The frontend shall therefore not be limited to displaying workflow metadata, but shall support active interaction with workflow execution in a form consistent with the actual capabilities of the system.

At the same time, workflow control must remain aligned with backend realities rather than promise actions that cannot be executed reliably. Apache Airflow models workflows as DAGs and exposes task- and run-related operational functionality through its API and surrounding components [2], [3], [62]. The frontend must therefore map user actions to these capabilities carefully. This implies that all control mechanisms presented to users shall correspond to supported backend operations and shall respect the distinction between operations that are technically feasible and those that are merely conceivable from a user-interface perspective.

A further requirement concerns the clarity of interaction. Workflow-control actions such as triggering, retrying, canceling, or deleting can have immediate operational consequences. They must therefore be presented in a way that makes their purpose understandable and their scope recognizable. The interface shall not obscure whether an action applies to a workflow, a run, or an individual task instance. Instead, it shall make the target and consequences of each control action sufficiently clear to support deliberate use and reduce the risk of accidental or misdirected interaction.

3.2.2 Monitoring, inspection, and visualization

The second major functional requirement is the ability to monitor and inspect workflow execution. A workflow frontend that only allows triggering would be

operationally insufficient, because users also need to observe execution progress, diagnose errors, and understand dependency structures. The interface shall therefore provide access to workflow-level and task-level state information in a manner that is both technically meaningful and operationally useful.

At the workflow level, the frontend shall display the state of workflow runs and provide sufficient contextual information to distinguish between different executions. At the task level, it shall expose the execution state of individual task instances, including the ability to identify running, successful, failed, queued, or otherwise relevant states. This requirement follows not only from the project context, but also from the general literature on technical dashboards, which emphasizes that interfaces must make relevant operational states visible in a way that supports concrete tasks and reduces interpretive friction [9], [10]. In the context of workflow orchestration, the interface therefore shall make execution status directly accessible rather than forcing users to reconstruct state from multiple disconnected views.

Status information alone, however, is not sufficient for effective diagnosis. The frontend shall also provide access to logs and error information for workflows and tasks. This includes the ability to inspect failure-related details and to navigate from high-level status indications to more detailed technical information where needed. The purpose of this requirement is not to replicate every part of Airflow's native debugging environment, but to expose enough information to support troubleshooting and operational understanding within MECOIS. Since workflows are often long-running, asynchronous, and failure-prone, the interface must enable not only detection of problems, but also inspection of their concrete manifestations.

A further core requirement is the visualization of task dependencies. Workflow orchestration is fundamentally structured by dependencies between tasks, and these dependencies are difficult to understand if presented only in textual or tabular form. The frontend shall therefore include a graphical representation of workflow structure that makes dependencies visible as a directed task graph. In practice, this supports users in understanding execution order, concurrency, branching, and potential bottlenecks. The literature on workflow interfaces and monitoring dashboards suggests that interface structure should reflect underlying process structure rather than treat it as secondary metadata [7], [8]. In this case, a dependency graph is therefore not decorative but functionally central.

Finally, the monitoring functionality shall support movement between overview and detail. Users shall be able to move from a workflow overview to specific runs, from runs to individual tasks, and from status indicators to logs or other diagnostic information. This hierarchical navigation requirement is important because workflow monitoring typically involves transitions between broad situational awareness and focused inspection. The interface must therefore support

both orientations: rapid scanning of overall workflow health and deeper inspection of specific execution problems.

3.2.3 Optional product extensions

In addition to the mandatory functional core, the artifact may include selected optional extensions if these can be implemented without jeopardizing the main thesis goals. Two such extensions are particularly relevant in the present project context.

The first optional extension is localization or internationalization support. If implemented, the frontend should be structured in a way that allows interface text and possibly other locale-sensitive elements to be adapted to different languages. This can improve maintainability and future extensibility, especially in multi-user or multi-project settings. However, localization is not a prerequisite for fulfilling the primary goal of integrating orchestration functionality into MECOIS. It is therefore treated as a beneficial extension rather than as a mandatory success criterion.

The second optional extension concerns backend- or Airflow-related structural adjustments that may become useful in order to support the frontend more effectively. Such adjustments could include interface-related backend refinements, improved data aggregation, or support functions that simplify frontend integration. Nevertheless, these changes remain secondary to the main task of designing and implementing the frontend artifact itself. The requirements chapter therefore treats them as optional architectural support measures rather than as core product requirements. This distinction is important for maintaining a clear thesis scope and for avoiding an unnecessary shift of emphasis from frontend engineering toward backend redesign.

3.3 Quality requirements

3.3.1 Usability and accessibility

Usability is a central quality requirement of the artifact because the frontend is intended to improve day-to-day interaction with workflow orchestration. A technically complete interface would still fail in practice if users could not quickly understand workflow states, locate relevant actions, or navigate efficiently between monitoring and control tasks. The frontend shall therefore present information and interaction possibilities in a way that supports operational clarity, reduces unnecessary complexity, and allows users to focus on the most relevant aspects of workflow execution. This requirement is consistent with the literature on workflow dashboards and monitoring systems, which shows that usability problems

often arise not from a lack of data, but from poor alignment between information presentation and task demands [9], [10], [11].

Usability in this context includes several more specific expectations. The frontend shall support clear orientation between workflows, runs, tasks, and logs. It shall present state information in a visually understandable manner, distinguish important actions from secondary ones, and avoid unnecessary fragmentation of interaction across too many disconnected views. Because users of orchestration interfaces often alternate between overview tasks and focused diagnosis, the frontend shall support both rapid scanning and detailed inspection. It shall therefore make important states salient while still allowing access to deeper technical detail when needed.

Accessibility is equally a mandatory quality requirement. As discussed in the literature review, accessibility is shaped not only by good design practice, but also by legal and normative frameworks such as Directive (EU) 2019/882, the German *Barrierefreiheitsstärkungsgesetz*, and WCAG 2.2 as the central technical benchmark [24], [25], [26], [27], [31]. Consequently, the frontend shall be designed in a manner compatible with relevant accessibility expectations, with Level AA of WCAG as the practical benchmark for web applications [32]. This requirement is not only important for legal and ethical reasons, but also contributes to robustness and broader usability for technical users in diverse working conditions.

Concretely, the interface shall be keyboard-accessible, because all essential functionality should be operable without pointer-only interaction [33]. It shall maintain logical and visible focus behavior, especially in interactive and stateful views [35], [36]. Dynamic elements such as dialogs, overlays, filters, and notifications shall be implemented in a way that preserves meaningful focus management and compatibility with assistive technologies [37], [51]. The frontend shall also avoid conveying critical status information by color alone, since WCAG explicitly prohibits the use of color as the sole carrier of meaning [45]. Accordingly, workflow states, warnings, and errors shall be represented through combinations of color, labels, icons, or text.

Another important accessibility-related requirement concerns semantic structure. Native HyperText Markup Language (HyperText Markup Language (HTML)) elements and labeling mechanisms shall be preferred wherever possible, while ARIA should be used deliberately and only where custom interaction patterns genuinely require it [34], [39]. This is especially relevant for forms, controls, tables, and dynamic widgets. Labels shall be provided for controls, instructions shall be clear where input is required, and interactive tabular structures shall distinguish appropriately between simple tables and more complex grid-like components [40], [41], [43], [44]. In sum, accessibility shall be treated as a first-order quality attribute of the artifact, closely tied to its overall usability and not reducible to isolated compliance checks.

3.3.2 Performance and responsiveness

The frontend must remain responsive under realistic usage conditions. Workflow orchestration interfaces differ from static information pages because they are built around asynchronous, stateful, and potentially frequently updated information. Workflow runs may change state while a user is observing them, logs may need to be retrieved or refreshed, and graphical dependency views may involve non-trivial rendering overhead. The frontend shall therefore be designed to support responsive interaction despite these dynamic conditions.

A first aspect of this requirement concerns data retrieval and updating. The interface shall load workflow, run, task, and log information efficiently and shall avoid unnecessary delays in common usage scenarios. Since orchestration data are often not purely static, the solution must also support controlled refresh strategies. The use of TanStack Query is particularly relevant here, because its official guidance emphasizes the importance of explicit decisions concerning staleness, refetching, retries, and invalidation [58], [59]. The frontend shall therefore manage server state in a way that keeps displayed information sufficiently current without causing excessive network load or disruptive user experience.

A second aspect concerns interactive rendering. Tables, filters, and especially DAG visualizations can become performance-sensitive when data sets grow or when state updates occur frequently. React Flow documentation highlights that unnecessary re-renders are a major source of performance problems and recommends disciplined memoization and careful control over frequently changing state [61]. The frontend shall therefore avoid avoidable rendering overhead and structure state updates in a way that preserves interactive smoothness as far as reasonably possible within the project context.

Responsiveness also has a user-facing dimension. Operations such as page changes, filter application, expansion of details, and transitions between overview and drill-down views shall feel timely and predictable. Where operations take longer, the interface shall communicate loading and updating states clearly rather than leaving users in uncertainty. This is particularly important in workflow interfaces, because the distinction between a genuinely slow backend operation and a seemingly unresponsive frontend must be visible to users if the interface is to remain trustworthy.

3.3.3 Maintainability and extensibility

Because the artifact is integrated into an existing software platform, maintainability is a core requirement rather than a secondary coding preference. The implementation shall be structured in a way that supports understanding, modification, extension, and testing over time. This requirement is especially important because workflow orchestration interfaces are likely to evolve with backend

capabilities, changing project needs, and evaluation findings.

Maintainability begins with component structure. The frontend shall use reusable and clearly delimited components wherever this improves clarity and reduces duplication. Shared interaction patterns, shared state representations, and repeated visual structures should be centralized rather than reimplemented ad hoc. This is consistent with React guidance emphasizing clear state ownership, "lifting state up" where appropriate, and avoiding contradictory or redundant state structures [54], [55]. In practice, this means that the frontend shall be organized in a way that prevents divergence between views that represent related workflow information.

Extensibility requires more than modularity alone. The solution shall separate concerns between data fetching, transformation, presentation, and interaction logic as far as reasonably possible. This is important because the frontend may later need to accommodate additional views, workflow features, or refinements identified in the evaluation. The artifact should therefore not be engineered as a one-off prototype optimized only for immediate demonstration, but as a structured contribution that can serve as a stable basis for future refinement within the MECOIS frontend.

Type safety also contributes directly to maintainability. TypeScript documentation emphasizes the value of stricter type checking and disciplined narrowing for safer program logic [56], [57]. Accordingly, the frontend shall use TypeScript in a way that reduces ambiguity at integration boundaries and supports maintainable handling of workflow-related data structures, states, and error conditions. This is particularly important where backend responses may contain optional fields, multiple result states, or version-sensitive API contracts.

3.4 Technical and architectural constraints

3.4.1 Platform and integration constraints

The artifact must operate within a fixed platform context. Most importantly, it must remain compatible with Apache Airflow 3.x and with the specific orchestration data and control model exposed by the project environment. Airflow documents workflows as DAG-based structures composed of tasks and related execution entities [2], [3]. Since the frontend is meant to visualize and manipulate this operational layer, it shall align its data handling and interaction model with the interfaces actually provided by the Airflow 3.x ecosystem [62]. Compatibility with Airflow is therefore not a generic interoperability wish, but a concrete technical requirement that constrains the design of frontend views, actions, and state representations.

In addition, the frontend must integrate with the existing MECOIS backend and not bypass it in an uncontrolled way. This means that the artifact shall respect the role of backend services as integration and mediation layers where appropriate. It shall neither assume unrestricted direct backend equivalence with Airflow nor ignore project-specific backend abstractions that exist to support the frontend. The requirements therefore imply a coordinated integration model in which the frontend uses available services in a way consistent with the surrounding system architecture.

A further platform constraint is the use of React and TypeScript as the primary frontend technologies. This determines much of the feasible implementation style. The solution shall therefore adopt patterns appropriate to component-based UI construction, typed integration code, and explicit client-side state handling. It should also align with the conventions and technical direction of the existing frontend codebase rather than introducing a fundamentally incompatible application model. In this respect, the artifact is constrained not only by what is technically possible, but also by what can be integrated coherently into the project.

3.4.2 Architectural constraints

The frontend cannot be designed as if it were a stand-alone application. It must fit into a pre-existing architectural environment with established routing, page composition, and service boundaries. Accordingly, the artifact shall respect the existing frontend routing structure and shall integrate its workflow-related pages or components into that structure without unnecessary architectural disruption. This is important because navigation coherence is part of both usability and maintainability. A frontend that introduces an entirely separate interaction island would undermine the integration objective of the thesis.

The same applies on the backend-facing side. The solution shall respect the existing distribution of responsibilities between frontend, backend, and Airflow services. Workflow state retrieval, orchestration control, authentication-related processing, and possible aggregation logic may not all reside at the same architectural level. The artifact must therefore be built in a way that acknowledges these boundaries instead of conflating them. This is a particularly important requirement because frontend architecture decisions in later chapters must be explainable in relation to pre-existing system constraints rather than only to frontend convenience.

Another architectural constraint is proportionality. The implementation shall avoid design decisions that would require disproportionate restructuring of the overall MECOIS system in order to support comparatively narrow frontend functionality. This does not exclude targeted architectural refinement, especially where such refinement substantially improves integration or maintainability. It

does, however, mean that the frontend must be designed with respect for the broader project context and with awareness that thesis scope is centered on the frontend artifact rather than on full-system reengineering.

3.4.3 Dependency and version constraints

The implementation depends on third-party libraries for important parts of the solution, especially for DAG visualization and server-state management. These dependencies create both opportunities and constraints. On the one hand, libraries such as React Flow and TanStack Query make it possible to realize complex functionality in a robust and efficient manner. On the other hand, each such dependency introduces requirements concerning version compatibility, maintenance state, integration effort, and long-term stability.

The frontend shall therefore use third-party libraries only where they provide clear project value and fit the technical requirements of the artifact. For the visualization of workflow dependencies, React Flow is an appropriate candidate because it supports interactive graph-based interfaces and includes guidance on accessibility and performance considerations [60], [61]. For server-state handling, TanStack Query is appropriate because it provides explicit mechanisms for staleness control, invalidation, retries, and synchronization with backend state [58], [59]. The requirement is not simply to use these libraries because they are available, but to use them in a technically disciplined way that justifies their inclusion.

Version constraints also matter. The artifact shall remain compatible with the package-management and dependency structure of the surrounding project. This includes avoiding unnecessary dependency growth, avoiding unstable or poorly maintained packages where feasible, and selecting library usage patterns that do not increase technical fragility. In an engineering thesis, this is important because the value of a frontend artifact depends not only on feature completeness, but also on whether the solution can be maintained and evolved in the actual project environment after the thesis concludes.

3.5 Security, quality assurance, and operational requirements

3.5.1 Security and authentication considerations

Workflow orchestration interfaces expose actions and information that may be operationally sensitive. Triggering runs, retrying tasks, canceling execution, or inspecting logs can affect infrastructure behavior, reveal technical details, or expose system internals. The frontend must therefore be designed with awareness

of authentication and authorization boundaries even where the detailed security architecture is not the primary focus of the thesis.

At a minimum, the frontend shall respect the authentication and authorization model of the surrounding system. It shall not assume that all users are equally permitted to inspect all workflow information or execute all control actions. Instead, the availability of frontend actions must remain compatible with backend-enforced permissions and security decisions. This is essential because security in such systems cannot be guaranteed solely at the presentation layer. The frontend may guide or restrict interaction, but actual enforcement must remain aligned with backend control.

The frontend shall also avoid exposing sensitive actions or technical data in a misleadingly unrestricted way. Where actions have significant operational consequences, the interface shall present them deliberately and clearly. Where data are sensitive, the frontend shall only render them in contexts where backend-provided access is appropriate. This requirement does not imply that the thesis must implement a complete permission model from first principles. It does require, however, that the frontend be designed in a way that remains compatible with secure system operation.

3.5.2 Testing and quality assurance requirements

The artifact must be developed under a systematic testing and quality-assurance strategy. This requirement is central because the frontend is not intended as an informal demonstration only, but as a working integration artifact whose behavior should be stable enough for realistic use and evaluation. Test-driven development and high test coverage are therefore treated as process and quality requirements rather than as product functionality.

A first requirement is that the implementation shall be supported by a clear testing strategy covering the most relevant levels of the frontend. This includes tests for critical utility logic, state-related transformations, and major integration-relevant interface behavior. Since the frontend interacts with asynchronous backend data and complex state transitions, testing must go beyond isolated rendering checks and address realistic behavior of important workflow-related components.

A second requirement is that test coverage shall be sufficiently high for the implementation to support stable development and regression prevention. High coverage is not an end in itself, and the chapter does not reduce quality assurance to a numerical threshold alone. Nevertheless, broad and meaningful coverage is important because the interface contains control logic, state-dependent rendering, and integration-sensitive features that are prone to breakage if left untested.

A third requirement concerns integration testing. The major functional paths of the workflow frontend shall be testable in ways that approximate realistic interaction, especially where data retrieval, state changes, and UI reactions are closely coupled. In practice, this means that the testing approach should support confidence in pages or components that present workflows, runs, logs, and control actions rather than only in isolated helper functions. The overall goal is not exhaustive formal verification, but a level of quality assurance appropriate for a prototype intended to serve as a credible engineering result.

3.5.3 Deployment, environment, and evaluation requirements

The artifact must also satisfy a set of operational requirements concerning deployment environment, demonstration readiness, and later evaluation. Because the thesis includes implementation and evaluation as major phases, the frontend shall be buildable, runnable, and demonstrable in the technical environment available to the project. This includes compatibility with the development setup, the available backend and Airflow environment, and the example workflows or demonstration data needed for realistic interaction.

A further requirement is that the artifact shall support meaningful evaluation scenarios. The evaluation chapter is expected to include user interaction with the interface in concrete tasks and to compare aspects of the developed solution with the native Airflow interface. The frontend must therefore be sufficiently complete and stable to support such scenarios. In particular, it must expose the operational capabilities that users are expected to evaluate and must do so in a form that allows observation of usability, effectiveness, and interaction quality.

This requirement is consistent with the broader literature on dashboard and technical-interface evaluation, which emphasizes iterative, task-based, and user-centered assessment methods [19], [20], [22]. For the present thesis, this means that evaluation support is not an afterthought added after implementation. Rather, the artifact shall be designed and realized in a way that permits realistic user tasks, meaningful qualitative feedback, and a defensible assessment of its strengths and shortcomings.

Overall, the requirements defined in this chapter establish the basis for the subsequent architecture, design, and implementation of the workflow orchestration frontend. They specify what the artifact must enable, which qualities it must satisfy, under which constraints it must be developed, and which operational conditions it must support. The next chapter builds on this basis by translating these requirements into an architectural concept for integrating workflow orchestration functionality into the existing MECOIS system.

4 Architecture

4.1 Architectural Context and Overview

The requirements established in the previous chapter set the constraints that directly influenced how the system was designed, which is presented in this chapter. Building on these requirements, the architecture of the proposed workflow integration frontend is described in terms of how it organizes the interaction between frontend components, backend services, and Apache Airflow. Particular emphasis is placed on the architectural role of the frontend within MECOIS, the integration mechanisms supporting workflow execution and monitoring, and the practical concerns such as authentication, state management, technology integration, and deployment. Rather than focusing on implementation details, which are addressed in the following chapter, this chapter concentrates on the principal architectural structures, design decisions, and trade-offs that guided the overall system design.

4.1.1 Role of the Frontend within MECOIS

The workflow integration frontend developed in this thesis constitutes an orchestration-oriented extension of the broader MECOIS platform rather than a standalone subsystem. Its architectural role is to connect user actions (e.g., triggering workflows) with backend processing logic, and workflow execution capabilities provided by Apache Airflow. Rather than exposing users directly to the native Airflow interface, which exists as an external and generalized orchestration interface, the proposed frontend embeds workflow control (e.g., triggering runs), monitoring (e.g., viewing task states), and execution management directly into the domain-specific environment of MECOIS.

The frontend assumes responsibility for three architectural concerns. First, it provides presentation-layer abstractions for pipeline monitoring and control, including visualization of directed acyclic graph (DAG) structures, workflow state inspection, task-level interactions, and runtime log access. Second, it acts as a mediation layer between users and backend services, encapsulating API interac-

tions, authentication concerns, and state synchronization mechanisms. Third, it establishes a foundation for reusable orchestration-oriented UI components that can be integrated into the existing MECOIS frontend ecosystem. This includes visualization components based on ReactFlow, reusable data-driven interfaces such as pipeline tables and execution views, and shared concerns such as caching, localization, and error reporting.

Architecturally, the frontend is therefore not merely a consumer of backend services but a coordinating layer that binds together user workflows and distributed backend execution. This architectural role also motivated the adoption of a strongly API-driven frontend architecture in which state, control actions, and monitoring data are consistently derived from backend and Airflow APIs rather than maintained through duplicated frontend logic. This principle contributes directly to consistency, maintainability, and alignment with existing MECOIS architectural conventions. The relationship of the frontend to surrounding system components is illustrated in Figure 4.1.

4.1.2 Overall System Architecture Overview

The overall architecture follows a layered client-backend-orchestration model in which the custom frontend operates as the primary interaction layer, the MECOIS backend serves as a mediation and integration layer, and Apache Airflow provides workflow scheduling and execution infrastructure.

At the frontend layer, the implementation is realized using React and TypeScript within the existing MECOIS technology stack. Vite provides build-time tooling and development infrastructure, while TailwindCSS and shadcn/ui support consistent interface composition. TanStack Query serves as the central server-state management layer, coordinating asynchronous data retrieval, caching, polling, and synchronization. ReactFlow is integrated as a specialized visualization component for representing workflow dependencies.

The backend layer provides controlled access to Airflow resources while encapsulating system-specific concerns such as endpoint aggregation, credential handling, and workflow-specific logic. Rather than allowing unrestricted direct coupling between frontend and orchestration engine, the backend acts as a stabilizing abstraction layer that reduces frontend exposure to changes in Airflow internals.

At the orchestration layer, Airflow provides scheduling, task execution, worker coordination, and runtime metadata through its API server, scheduler, and worker processes. The architecture supports both simple linear workflows and more complex task dependency structures, including validation pipelines, ETL workflows, and health monitoring processes.

The overall architectural decomposition and subsystem boundaries are shown in

Figure 4.1.

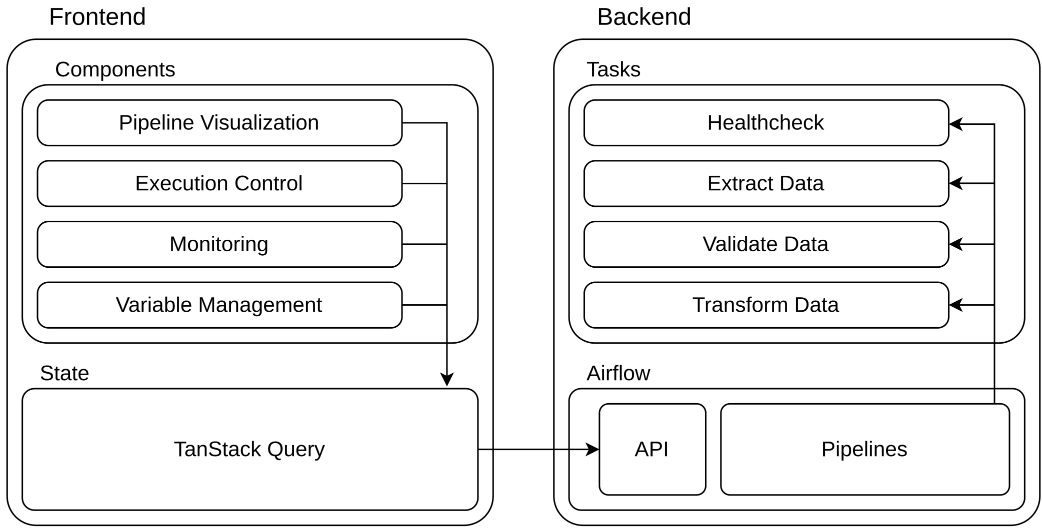


Figure 4.1: Overall System Architecture of the MECOIS Workflow Integration Frontend, Illustrating Frontend Components, Backend Tasks, Airflow Services, and Their Communication Paths

The figure emphasizes two central architectural principles. First, frontend concerns are intentionally separated into component-level interaction logic and centralized state management using TanStack Query. Second, backend workflow tasks are modeled independently from Airflow scheduling primitives while remaining executable through Airflow pipelines. This decoupling permits frontend abstractions to remain stable even when workflow implementations evolve.

4.2 Frontend and Backend Integration Architecture

4.2.1 Interaction Between Frontend, Backend, and Airflow

Communication between the frontend, backend, and Airflow follows an API-centric interaction model based on controlled request routing and layered abstractions. User interactions originating in the frontend, such as triggering workflows, pausing DAGs, retrieving task instances, or accessing execution logs, are translated into structured API operations through dedicated client abstractions. These requests are forwarded to Airflow APIs.

This interaction model separates command-oriented operations from state retrieval operations. Control interactions such as triggering or canceling executions are modeled as explicit mutation requests, whereas monitoring operations rely on query-based polling and state synchronization. This distinction aligns naturally with TanStack Query's architectural model and reduces coupling between view components and asynchronous execution semantics.

The integration architecture also addresses the asynchronous characteristics of workflow orchestration. Since pipeline execution may span long durations and involve multiple distributed workers, frontend interactions are not modeled as synchronous request-response transactions but as state transitions observed through repeated backend communication. Polling intervals, cache invalidation strategies, and refetch triggers thereby become part of the architecture itself rather than implementation details.

Another important characteristic of the interaction architecture is the coexistence of frontend abstractions and Airflow-native concepts. DAGs, DAG runs, task instances, variables, and logs originate as Airflow resources, but the frontend encapsulates these into user-oriented abstractions aligned with MECOIS workflows. This avoids exposing internal orchestration complexity directly to users while preserving access to detailed execution information when required.

4.2.2 Backend Structure and Airflow Task Architecture

The backend architecture combines conventional application service responsibilities with orchestration-specific workflow decomposition. In addition to providing API access, the backend structures processing logic as modular Airflow task pipelines that can be composed into reusable DAGs. This decomposition is represented in Figure 4.2.

The depicted architecture distinguishes individual tasks from orchestration-level pipelines. Tasks such as health checks, data extraction, validation, transformation, and loading represent reusable processing units. These tasks are composed into pipelines such as health monitoring workflows, validation workflows, and ETL pipelines, which are scheduled and coordinated through Airflow.

This distinction supports modularity at two levels. At the task level, individual operations can be reused across workflows and evolved independently. At the pipeline level, task compositions can represent domain workflows without embedding orchestration logic into frontend concerns. This separation proved particularly important for representing task dependencies graphically in the frontend through ReactFlow, since frontend DAG visualizations can map directly onto backend task graphs.

The architectural role of Airflow scheduler and workers is similarly separated. The

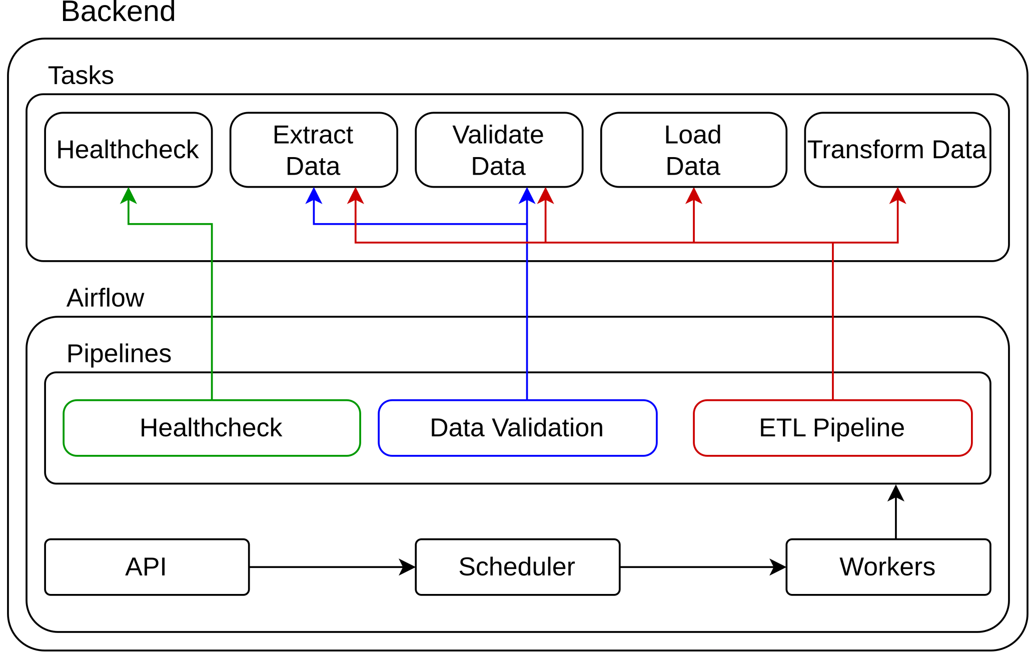


Figure 4.2: Backend and Airflow Task Architecture Showing Modular Task Composition, Pipeline Groupings, and Execution Coordination Through Airflow Services

scheduler governs dependency resolution and execution ordering, whereas workers execute task logic. The frontend interacts with the observable state generated by this backend structure rather than with task execution directly. This preserves a clean separation between monitoring and execution responsibilities.

4.3 Authentication and API Client Architecture

4.3.1 Authentication and Authorization Flow

Authentication follows a layered token-based model designed to reconcile user-level credentials managed within MECOIS with Airflow's API authentication requirements. Rather than requiring users to authenticate independently against Airflow through its native interface, the architecture embeds Airflow credential handling into the frontend-backend interaction model.

The authentication flow is illustrated in Figure 4.3.

As shown, Airflow credentials are associated with users through Supabase-backed storage and used by the client to obtain persistent authentication tokens issued

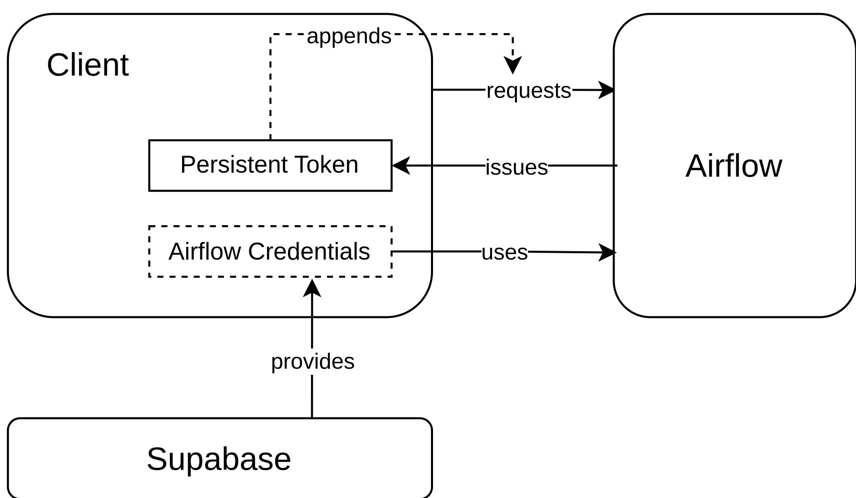


Figure 4.3: Token-Based Authentication and Authorization Architecture Integrating Supabase Credential Storage, Persistent Client Tokens, and Airflow API Access

by Airflow. These tokens are subsequently appended to API requests for authenticated interactions with the orchestration backend.

This architecture was chosen over repeated credential exchange or direct session-bound authentication because it reduces authentication overhead, supports persistent API communication required for polling-heavy workloads, and aligns with Airflow 3.x bearer-token based authentication mechanisms. It also isolates credential management concerns from higher-level workflow interaction logic.

Authorization naturally remains resource-oriented and inherited from backend and Airflow capabilities. The frontend therefore acts as an enforcement and visibility layer rather than the authoritative source of access control decisions.

4.3.2 API Abstraction and Client Design

To prevent tight coupling between UI components and backend endpoint semantics, API access is encapsulated through a dedicated client abstraction built around generated and customized API interfaces. Orval-generated clients provide typed bindings for Airflow endpoints, while Axios-based client configuration centralizes transport-level concerns including authentication headers, base Uniform Resource Locator (Uniform Resource Locator (URL)) resolution, interceptors, and error normalization.

This architecture introduces an intermediate abstraction layer between compon-

ents and remote services. Components consume domain-oriented query and mutation hooks rather than constructing API calls directly. This reduces repetition, improves type safety, and localizes backend integration changes.

The design also aligns strongly with separation-of-concerns principles. Transport concerns remain isolated in client configuration, endpoint semantics are represented through generated interfaces, and application-level data consumption is encapsulated in query hooks and reusable frontend abstractions.

4.4 State Management and Data Flow Architecture

4.4.1 State Management with TanStack Query

A central architectural decision was the treatment of remote orchestration state as server state rather than application-local state. Workflow definitions, DAG runs, task instances, and log streams are externally managed, frequently changing resources and therefore unsuitable for conventional client-side state management approaches.

TanStack Query was adopted as the architectural foundation for server-state management because it combines caching, refetching, mutation coordination, optimistic update support, and polling mechanisms within a single abstraction model. This is particularly suitable for workflow monitoring interfaces, where state freshness and asynchronous consistency are more critical than complex local interaction state.

Queries are organized around stable resource keys and parameterized by workflow identifiers, execution contexts, and filtering parameters. This structure enables predictable cache partitioning while supporting targeted invalidation after mutations such as workflow triggering or pausing.

Polling-based synchronization, rather than ad hoc manual refresh logic, is treated as part of the architectural model. Query-level polling intervals enable continuous monitoring without embedding refresh behavior into presentation components.

4.4.2 Component Data Flow and Backend Communication

Component-level data flow follows a predominantly unidirectional architecture in which data enters components through query abstractions, propagates downward through composition boundaries, and triggers backend mutations through controlled interaction handlers.

Presentation components remain largely stateless with respect to backend resources and consume derived data from query layers. This separation reduces duplication of loading, error, and synchronization logic across views. It also improves testability by separating rendering concerns from backend communication concerns.

The architecture overview in Figure 4.1 illustrates this relationship explicitly through the separation between frontend components and centralized TanStack Query state management.

Particularly for DAG visualizations, this architecture proved important because graph rendering requires transformed backend task metadata while remaining synchronized with underlying execution state.

4.4.3 Caching and Request Optimization

Caching is not just a performance optimization but an architectural mechanism supporting scalability and consistency. Multiple request patterns in the system involve repeated access to slowly changing metadata such as workflow definitions, task dependency graphs, and variable configurations. Other resources, such as run states and logs, exhibit higher volatility.

The architecture therefore differentiates caching strategies according to data volatility. Stable resources employ longer stale times and aggressive cache reuse, whereas runtime-sensitive resources rely on shorter intervals and periodic refetching. Mutation-triggered invalidation ensures consistency after control actions.

Request optimization additionally benefits from deduplication mechanisms, background refetching, and shared cache reuse across components consuming overlapping workflow data. These mechanisms reduce redundant backend load while maintaining responsive interfaces.

4.5 Technology Integration and Supporting Architectural Concerns

4.5.1 Integration of Core Frontend Technologies

The architecture relies on coordinated integration of several frontend technologies, each addressing distinct architectural concerns. React provides component composition and interaction modeling. TypeScript supports type safety across component boundaries and API integrations. Vite contributes lightweight development and build infrastructure aligned with modern frontend workflows.

TailwindCSS and shadcn/ui support a consistent design system while preserving composability and reuse. ReactFlow extends the architecture with graph-based visualization capabilities specifically required for DAG representation. Orval and Axios jointly support typed API integration and transport abstraction.

Importantly, these technologies are not independent additions but form an integrated architectural stack. For example, ReactFlow depends on state synchronization through TanStack Query, while typed endpoint definitions generated through Orval propagate into component-level data retrieval. This interdependence is architecturally relevant because it supports consistency across visualization, interaction, and backend communication concerns.

Internationalization support and reusable shared utility abstractions similarly contribute to architectural maintainability beyond immediate functional requirements.

4.5.2 Error Handling and Logging Architecture

Error handling follows a layered architecture addressing transport-level errors, domain-level workflow failures, and user-facing feedback concerns separately.

At the transport layer, Axios interceptors and client abstractions normalize communication failures and propagate structured error states into query abstractions. At the server-state layer, query-level loading and error semantics support consistent recovery and retry behavior.

At the application layer, workflow-specific failures, including failed task states and execution errors, are surfaced through log access and state visualizations rather than treated as conventional request failures. This distinction is important because orchestration failures are domain events rather than communication errors.

Logging architecture similarly spans multiple layers. Backend and Airflow logs remain authoritative execution sources, while frontend concerns focus on retrieval, aggregation, and presentation. This preserves separation between operational logging and interface-level diagnostics.

4.6 Architectural Decisions and Trade-Offs

Several architectural decisions involved explicit trade-offs between competing quality attributes.

A central decision was the adoption of an API-driven frontend architecture with TanStack Query rather than heavier client-state solutions. This prioritized consistency and server-state synchronization over generalized state flexibility. While

this introduces dependency on query abstractions and polling strategies, it substantially simplifies orchestration-oriented state management.

A second decision concerned mediated backend interaction rather than unrestricted direct coupling to Airflow APIs. Although this adds architectural layers and some integration complexity, it improves abstraction stability, security, and alignment with MECOIS architectural boundaries.

A third trade-off involved polling-based monitoring rather than real-time push-based synchronization. Although push-based architectures could reduce latency, polling was selected due to lower infrastructure complexity, alignment with workflow monitoring requirements, and compatibility with existing backend capabilities. This decision is also consistent with project scope constraints identified in the thesis requirements and work outline.

Another trade-off concerns visualization abstraction. ReactFlow-based DAG representations introduce additional frontend complexity, yet they provide substantial gains in interpretability and workflow transparency, which are central to the thesis objectives.

Finally, the architecture reflects a broader trade-off between introducing reusable infrastructural foundations, including typed API generation, shared state management, localization, and reusable component abstractions, versus implementing a narrowly scoped feature solution. The chosen architecture intentionally favors extensibility and maintainability, positioning the prototype as an architectural foundation for future workflow-oriented frontend capabilities within MECOIS.

5 Design and Implementation

The architecture presented in the previous chapter established the structural foundations, integration boundaries, and principal design decisions underlying the proposed workflow integration frontend. Building upon this architectural foundation, this chapter focuses on the concrete realization of these structures through interface design, component implementation, and supporting infrastructural mechanisms. Whereas the architecture chapter emphasized system organization and architectural rationale, this chapter shifts attention toward how these decisions were translated into implementable frontend structures.

Particular emphasis is placed on the design objectives guiding the user interface and the implementation strategies adopted to operationalize architectural decisions. In accordance with the engineering focus of this thesis, the chapter combines design rationale with implementation-level discussion, highlighting not only what was implemented but also how specific design choices address the requirements established earlier.

5.1 Design Objectives and Implementation Approach

5.1.1 UI/UX Design Principles and Guidelines

The design of the workflow integration frontend was guided by the dual objective of supporting technically sophisticated orchestration interactions while maintaining usability for users with varying degrees of familiarity with Apache Airflow and pipeline-oriented systems. Unlike general-purpose dashboard interfaces, the interface developed in this work targets developer-facing and operator-facing interactions involving monitoring, triggering, and inspecting asynchronous workflows. This imposed specific usability demands related to transparency of system state, interaction efficiency, and support for cognitively complex information.

A primary design principle was progressive disclosure of complexity. Workflow orchestration systems inherently expose layered information, ranging from high-

level workflow states to detailed task execution logs. Presenting all available information simultaneously would create visual overload and reduce interpretability. The interface therefore favors layered disclosure patterns in which summary information is presented first, while detailed runtime data, execution metadata, and diagnostics remain accessible through structured expansion patterns, dedicated views, or secondary interaction mechanisms.

A second principle concerned visibility of system status. Since workflow orchestration involves asynchronous operations whose effects may unfold over extended periods, users require continuous awareness of execution states and state transitions. This principle influenced the extensive use of status indicators, execution state representations, progress-oriented visual feedback, and live-updating workflow views. State information is consistently represented through redundant visual channels, including textual labels, iconography, and color-coded semantics, explicitly avoiding reliance on color as the sole information carrier.

Another central principle was minimizing interaction friction for repetitive tasks. Common interactions such as copying identifiers, triggering workflows, inspecting logs, or filtering execution records were designed to minimize unnecessary navigation and repeated input effort. This objective motivated reusable interaction elements such as copy-field components, contextual tooltips, compact action controls, and filter persistence mechanisms.

Accessibility constituted an explicit design objective rather than a secondary consideration. Consistent with accessibility-oriented design principles introduced earlier in the thesis, interface decisions incorporated considerations including keyboard accessibility, semantic component structure, screen-reader compatible controls, focus visibility, and support for sufficient visual contrast. This was particularly relevant for status representations and graph-based workflow visualizations, where alternative informational cues beyond color were intentionally incorporated.

Consistency and reuse further informed the interface design language. Repeated interaction patterns across workflows, runs, task views, and variables management were deliberately standardized to reduce cognitive switching costs. Shared interaction models, consistent table behaviors, common status semantics, and reusable layout structures contribute not only to usability but also to maintainability.

The design process was additionally influenced by the specific context of technical developer tooling, drawing conceptually from interaction paradigms common in Continuous Integration and Continuous Delivery/Deployment (Continuous Integration and Continuous Delivery/Deployment (CI/CD)) dashboards, monitoring systems, and developer-oriented control interfaces. Rather than emphasizing consumer-oriented interface minimalism, the design favors information density

balanced through structure and progressive disclosure.

These principles collectively shaped not only visual interface decisions but also deeper implementation choices regarding component decomposition, state synchronization behavior, and error representation strategies discussed throughout this chapter.

5.1.2 General Implementation Strategy

The implementation strategy followed an iterative, component-driven approach aligned with the engineering orientation of the thesis and the incremental development process established during the project. Rather than pursuing a strictly sequential implementation of isolated features, the frontend was developed through successive increments in which reusable infrastructural foundations and workflow-specific features evolved together.

A deliberate implementation principle was to prioritize infrastructural foundations before feature breadth. Core concerns such as API integration, server-state synchronization, reusable components, and common utility abstractions were established early because they constrained and enabled later feature implementation. This approach reduced duplication and avoided introducing isolated feature implementations that would later require architectural refactoring.

Implementation proceeded largely through vertical slices that combined backend integration, data handling, interface behavior, and testing considerations for cohesive feature sets. For example, workflow triggering functionality was not treated solely as a user interface concern, but implemented alongside mutation handling, state invalidation logic, status feedback mechanisms, and associated reusable controls.

This approach also supported iterative refinement through frequent evaluation of implemented components against usability and maintainability considerations. Several interface elements underwent multiple refinement cycles, particularly in areas involving filtering interactions, status visualization, and workflow graph presentation.

At the technical level, the implementation strategy favored compositional rather than monolithic structures. Page-level views were assembled from reusable components, while cross-cutting concerns such as loading states, formatting utilities, status representations, and data table behaviors were extracted into shared abstractions. This strategy aligns directly with the maintainability objectives established in the requirements chapter.

A further implementation decision concerned strong reliance on typed contracts across component boundaries and API interactions. TypeScript typing, generated API models, and typed query interfaces were used not merely for implementation

convenience but as mechanisms to reduce integration inconsistencies and support reliable evolution of both frontend and backend interfaces.

The implementation process also followed an intentionally pragmatic stance regarding capabilities such as real-time synchronization or specialized orchestration interactions. Polling-based synchronization instead of event-driven streaming is one example of a broader implementation philosophy.

Overall, the implementation strategy can therefore be characterized as iterative, infrastructure-first, component-driven, and strongly aligned with architectural constraints established in the preceding chapter.

5.2 Cross-Cutting Frontend Infrastructure

5.2.1 Reusable Component Architecture

A major implementation objective was to avoid treating each workflow-related interface as an isolated page-specific solution. Instead, the frontend was structured around a reusable component architecture in which shared abstractions encapsulate recurring interaction patterns, presentation logic, and infrastructural concerns.

This component architecture follows a layered composition model. At the lowest level, utility-oriented primitives support common formatting and interaction concerns, including date and duration formatting, copy-to-clipboard interactions, status indicators, tooltip-based contextual hints, and generic loading or feedback components. These primitives are intentionally lightweight and broadly reusable.

Above this level, reusable domain-oriented components encapsulate recurring workflow interaction structures. Examples include shared table abstractions, execution state visualizations, reusable action controls, workflow overview cards, and graph-oriented visualization components. These components embed domain semantics while remaining sufficiently generic for reuse across multiple workflow views.

At the composition level, page-oriented components assemble these reusable building blocks into higher-level interfaces corresponding to workflow overviews, workflow run inspection, task instance exploration, and variables management. This layered approach reduces duplication while preserving separation between reusable concerns and page-specific orchestration.

The component model also reflects explicit decisions regarding component granularity and export structure. In line with project development conventions, reusable components are defined predominantly as individually scoped modules

with explicit interfaces, favoring named exports and compositional reuse over broad monolithic component hierarchies.

An important aspect of this architecture is that reuse is not limited to visual components. Behavioral reuse was treated as equally important. Patterns such as consistent action handling, common loading-state rendering, shared filtering interactions, and standardized status semantics are all part of the reusable architecture.

This architectural approach contributes to maintainability and testability, but it also directly supports interface consistency. Users encounter similar interaction patterns across distinct workflow contexts because those patterns are often implemented through shared component abstractions.

5.2.2 Data Fetching and State Synchronization with TanStack Query

Server-state management constitutes one of the most consequential infrastructural layers in the implementation. As discussed in the architecture chapter, workflow definitions, task states, and execution metadata are inherently remote and dynamic resources. Their treatment as server state rather than conventional application-local state shaped the implementation substantially.

TanStack Query was adopted not only as a data-fetching library but as the operational foundation for query orchestration, cache coordination, and asynchronous synchronization. Query abstractions were organized around resource-oriented query keys that reflect workflow identity, execution context, filtering parameters, and dependent resource scopes.

This structure supports predictable cache behavior while enabling fine-grained invalidation. For example, mutations such as triggering a workflow run or pausing a workflow invalidate only relevant query scopes rather than causing indiscriminate refetching.

Polling-based synchronization was implemented selectively according to resource volatility. Workflow definitions and structural metadata generally rely on longer-lived cache behavior, whereas run states and task execution information use periodic refetching to support near-live monitoring behavior. This differentiation was important both for responsiveness and backend load management.

Another significant implementation consideration was query composition. Complex views often depend on multiple coordinated resource queries, for example combining workflow metadata, execution details, and task states. Query composition patterns were therefore designed to preserve separation between dependent resources while supporting coherent loading and error behavior at the component

level.

Mutation handling further motivated the adoption of TanStack Query. Triggering, retrying, pausing, or deleting workflow-related resources are implemented through controlled mutation abstractions coupled with invalidation logic and user feedback mechanisms. This centralizes consistency handling and avoids embedding backend synchronization concerns into UI event handlers.

Taken together, TanStack Query became not simply an implementation convenience but a foundational synchronization layer supporting much of the frontend's behavior.

5.2.3 API Client Generation and Request Handling

API integration was designed around the objective of combining strong type safety with maintainable endpoint abstraction. Rather than manually implementing endpoint clients throughout the frontend, request handling is structured around generated API bindings supplemented by customized transport-level configuration.

Orval is employed to generate typed client interfaces from API specifications, producing endpoint bindings, response models, and query-compatible abstractions. This significantly reduces manual duplication of request definitions while strengthening alignment between backend contracts and frontend usage.

Generated interfaces are complemented by a customized Axios client layer that centralizes authentication handling, base URL configuration, request interceptors, and normalized error handling. This separates transport concerns from domain-level endpoint use.

A major advantage of this architecture is that endpoint semantics are not embedded directly in presentation components. Components consume generated hooks and domain abstractions rather than constructing request logic themselves. This reduces coupling, improves maintainability, and supports evolution of backend interfaces.

Request handling additionally incorporates normalization of common backend interaction concerns such as retry handling, standardized error propagation, and consistent authentication header application. These concerns are deliberately centralized rather than repeated across query implementations.

This layered approach, combining generation and customized transport abstraction, proved particularly beneficial in a project involving frequent iterative backend adjustments, since frontend changes resulting from API evolution could often be localized.

5.2.4 Localization and Internationalization Support

Although localization is not central to workflow orchestration functionality itself, internationalization support was deliberately integrated as part of the frontend infrastructure. This decision reflects a broader maintainability-oriented implementation philosophy in which foundational concerns are addressed proactively when architectural leverage is highest.

Localization support is based on centralized translation resources and key-driven string management integrated into reusable component structures. User-facing text is therefore externalized from component implementations rather than embedded as hard-coded strings.

This approach supports multilingual extensibility, but equally important, it improves consistency and maintainability even in single-language deployments by centralizing textual definitions and reducing uncontrolled duplication.

Integration of internationalization at the infrastructural level also required design considerations beyond text replacement. Interface layouts, reusable component labels, tooltip content, validation messages, and status descriptions all needed to support translatability without disrupting layout assumptions.

The decision to incorporate localization support during implementation rather than as later retrofitting significantly reduced structural complexity. Retrofitted internationalization often requires invasive refactoring, whereas early integration allowed localization concerns to become part of reusable component patterns from the outset.

More broadly, inclusion of internationalization support reflects the general implementation strategy underlying this work: features contributing to long-term extensibility and project-wide consistency were prioritized when they could be incorporated architecturally rather than bolted on later.

5.3 Workflow Management Interfaces

The cross-cutting infrastructure described in the previous section provides the technical foundation for the concrete workflow-oriented interfaces implemented in the frontend. Building upon these reusable infrastructural mechanisms, this section focuses on the design and implementation of the user-facing workflow management interfaces. Particular emphasis is placed on how these interfaces support workflow discovery, execution control, monitoring, and detailed runtime inspection while maintaining consistency with the design objectives introduced earlier.

Although these interfaces appear as separate pages within the frontend, they were

designed as a coherent interaction system rather than independent views. Navigation between overview-level workflow interactions, workflow-level inspection, and individual workflow run analysis follows a layered interaction model in which users progressively move from summary-oriented control to increasingly detailed operational views. This layered model was intentionally designed to support both exploratory use and focused diagnostic workflows.

The following subsections discuss the design rationale, accessibility considerations, and implementation structure of the core workflow management interfaces.

5.3.1 Workflows Page

User Experience and Interaction Design

The Workflows page serves as the primary entry point into orchestration interactions and was therefore designed to support rapid orientation, low-friction workflow access, and efficient execution control. Conceptually, the interface combines elements of dashboard overview, navigation hub, and lightweight control surface.

Figure 5.1 shows the compact overview representation in which workflows are presented in an accordion-oriented structure designed to balance information density with progressive disclosure.

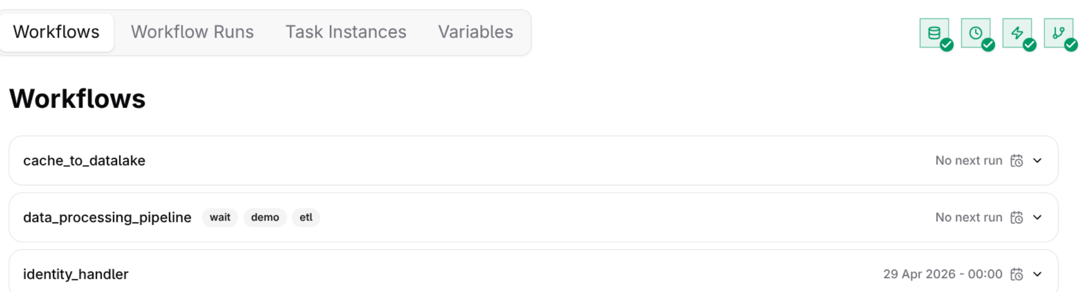


Figure 5.1: Compact Overview of the Workflows Page Showing Accordion-Based Workflow Listing and High-Level Scheduling Information

Rather than immediately exposing full workflow metadata for all available workflows, the interface presents a compact summary representation optimized for scanning and navigation. Workflow identifiers, scheduling information, and lightweight status signals provide sufficient information for overview-level decisions while avoiding cognitive overload.

Accordion-based expansion was selected as the primary interaction pattern because it allows users to move seamlessly between overview and detail inspection without disruptive navigation changes. Expanding a workflow reveals richer

metadata, execution controls, and direct entry points into deeper workflow inspection. This expanded interaction state is shown in Figure 5.2.

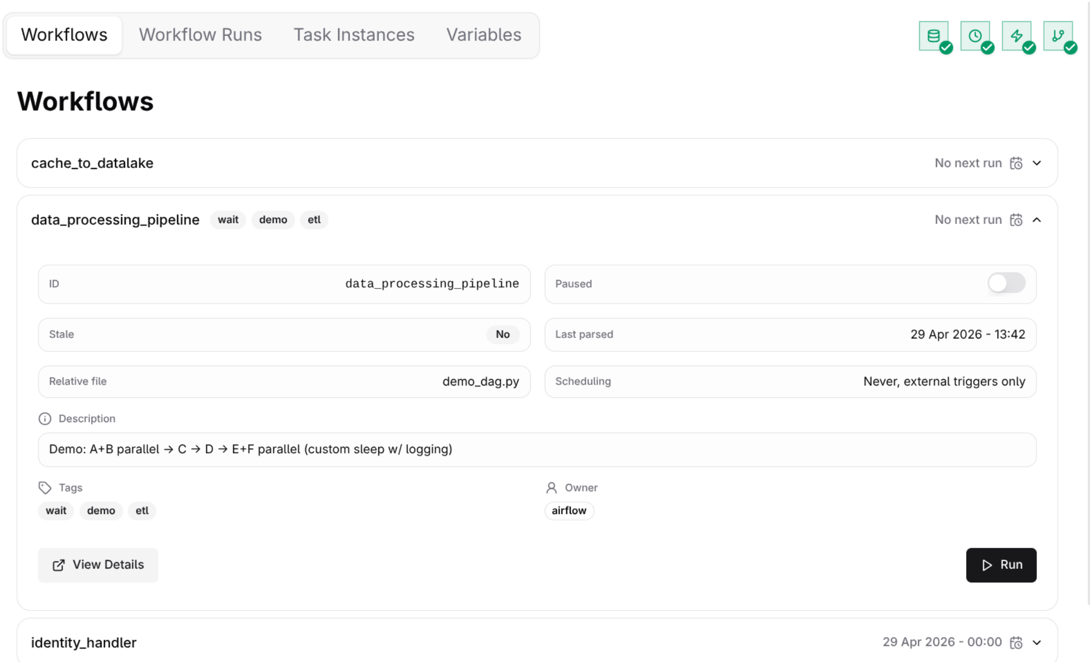


Figure 5.2: Expanded Workflow View Providing Embedded Workflow Metadata, Reusable Workflow Details, and Direct Execution Controls

The expanded design deliberately combines read-oriented and action-oriented interactions within a single contextual space. Users can inspect metadata such as scheduling information, ownership, tags, and descriptions while also triggering workflow execution or navigating into dedicated detail views. This reduces unnecessary context switching for common operational tasks.

The placement of the run action directly within the expanded view reflects an intentional interaction principle: high-frequency control operations should remain close to the contextual information informing those actions. Similarly, the explicit separation between quick execution and navigation into full details supports both routine operational usage and deeper diagnostic workflows.

Particular attention was also given to representing workflows as domain entities rather than merely Airflow objects. Terminology, information grouping, and interaction structure were shaped around user-oriented workflow concepts while preserving fidelity to underlying orchestration semantics.

Accessibility Considerations

Accessibility considerations for the Workflows page were informed both by general accessible interaction design principles and by requirements associated with the EAA, particularly regarding perceivability, operability, and robustness in interactive software systems.

Accordion interactions were implemented with semantic disclosure patterns and keyboard-operable controls to ensure workflows can be discovered, expanded, and triggered without reliance on pointer interaction. Focus order and tab navigation were designed to preserve logical reading and interaction flow across compact and expanded workflow states.

State-related information, including scheduling indicators, workflow metadata, and run controls, is represented through redundant channels rather than visual styling alone. This reflects deliberate adherence to accessibility principles requiring that color or visual emphasis never be the sole carrier of meaning.

The reusable workflow detail structures additionally support accessibility by employing semantically grouped information regions and consistent field-value layouts that improve interpretability for assistive technologies. Action controls such as run buttons and navigation actions include descriptive labeling and icon-plus-text combinations that improve both discoverability and accessibility.

EAA-aligned considerations also influenced interaction predictability and error prevention. Triggering a workflow represents a consequential operation, and the interaction design therefore emphasizes explicitness of control and avoids ambiguous action affordances.

Beyond formal compliance-oriented considerations, accessibility was treated more broadly as support for diverse user capabilities and expertise levels. Progressive disclosure itself can be understood as an accessibility-supporting mechanism because it reduces cognitive load and supports incremental information processing.

Code Structure and Reusability

A central implementation characteristic of the Workflows page is that its visible interactions are assembled from reusable domain-oriented components rather than implemented as page-specific logic.

Most notably, the expanded workflow metadata region shown in Figure 5.2 is implemented through a reusable Workflow Details component that is not confined to the Workflows page but reused throughout workflow-level interfaces. This component encapsulates metadata presentation, shared action controls, layout semantics, and interaction patterns, ensuring both consistency and maintainability.

This reuse is architecturally significant because workflow metadata presentation appears across overview-level and detailed views. By extracting it into a dedicated component rather than duplicating its implementation, the system preserves a single consistent representation of workflow-level information while reducing duplication and simplifying future evolution.

The page itself acts largely as a composition layer combining accordion structures, reusable detail components, shared state-aware action controls, and query-backed data retrieval. Interaction behavior such as expansion state, run triggering, and navigation actions is intentionally kept separate from presentation logic where possible.

This structure also supports testability. Because workflow detail rendering and associated interactions are encapsulated in reusable components, they can be validated independently from the page-level orchestration logic.

5.3.2 Workflow and Workflow Run Pages

User Experience and Interaction Design

Where the Workflows page supports overview-level orchestration interaction, the Workflow and Workflow Run pages support detailed inspection and execution analysis. Their design centers on combining structural pipeline visualization, metadata inspection, and run-oriented analysis within integrated but layered views.

The Workflow page combines three major interaction regions: DAG visualization, reusable workflow metadata and control structures, and historical workflow run inspection. This composition is shown in Figure 5.3.

The DAG visualization constitutes a central interaction element because it provides structural transparency into task dependencies that cannot be communicated effectively through tabular representations alone. Figure 5.4 highlights this visualization region.

Its placement at the top of the page reflects a deliberate prioritization of structural understanding before detailed metadata inspection. For workflow-oriented users, understanding execution structure often precedes investigation of individual run histories.

The reusable workflow details region remains intentionally aligned with its representation on the Workflows page, preserving consistency while enabling richer contextual interaction in the dedicated workflow view. This region is emphasized in Figure 5.5.

Historical workflow runs are integrated directly into the page rather than separ-

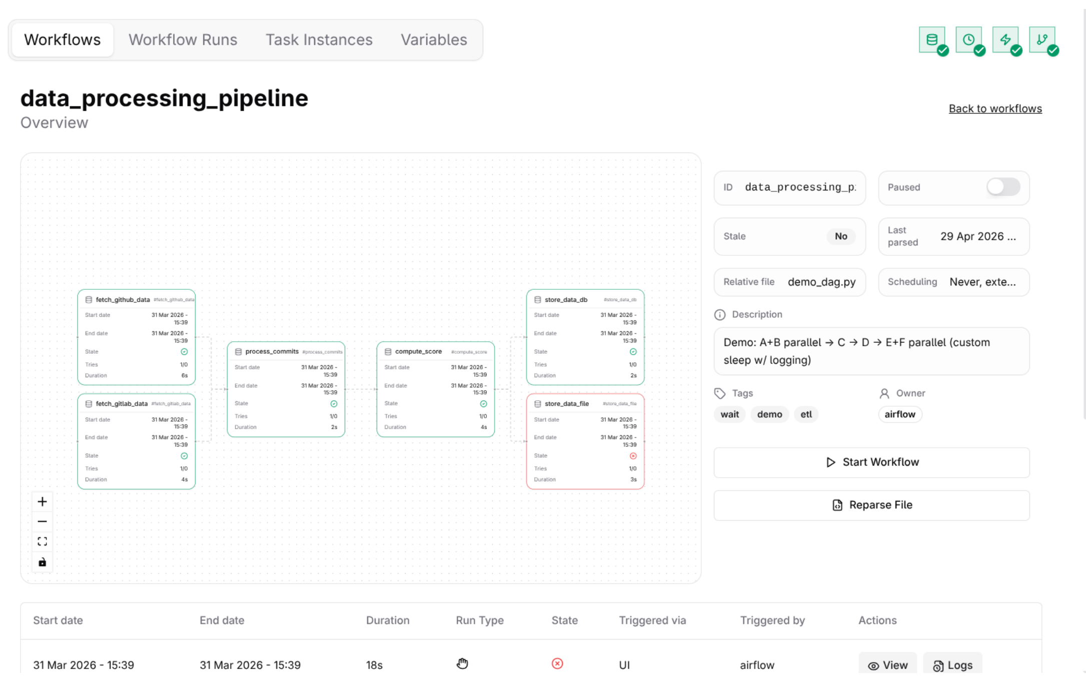


Figure 5.3: Workflow Detail Page Integrating DAG Visualization, Reusable Workflow Details, and Historical Workflow Run Overview

ated into disconnected navigation contexts, supporting a continuous exploratory workflow from structure inspection into execution history. This is illustrated in Figure 5.6.

The Workflow Run page extends this layered design further by combining run-specific metadata with run-state visualization and task-level execution structure. Figure 5.7 shows the combined interface.

Unlike the workflow-level DAG representation, the run visualization emphasizes runtime state rather than static dependency structure, supporting diagnosis of failures, parallel execution behavior, and execution progress. Complementing this, detailed run metadata is presented through structured execution information shown in Figure 5.8.

Together, these interfaces support a layered diagnostic workflow in which users move from workflow discovery, to structural understanding, to run-level execution analysis without discontinuous interaction shifts.

Accessibility Considerations

Accessibility concerns become particularly significant in these detailed workflow interfaces because they involve complex visual and state-rich information structures.

data_processing_pipeline
Overview

Figure 5.4: Interactive DAG Visualization Embedded in the Workflow Page for Structural Inspection of Task Dependencies

For DAG visualizations, accessibility considerations extended beyond conventional form and navigation concerns. Because graph-based representations can pose barriers for users relying on assistive technologies or alternative interaction modes, the visualization is complemented by alternative structured representations through surrounding metadata and tabular run information. This reflects an EAA-aligned principle of providing equivalent access paths to critical information.

Task-state representations similarly use redundant encoding through text, iconography, and color semantics to avoid inaccessible reliance on color-based state communication. This is particularly important for execution states such as failed, running, or successful tasks.

Keyboard navigation considerations informed zoom controls, graph interaction affordances, and navigation between visualization regions and metadata regions. Focus management was especially important because these interfaces contain dense interactive structures.

Semantic grouping of timing information, trigger metadata, and configuration

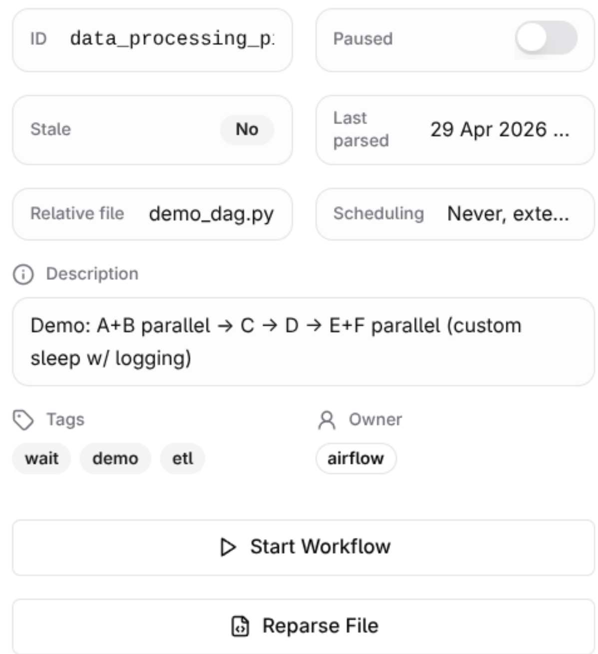


Figure 5.5: Reusable Workflow Details Component Embedded in the Dedicated Workflow View

information in the Workflow Run page improves both machine interpretability and cognitive accessibility. Structured information regions were intentionally designed to support predictable reading order and reduced scanning burden.

From a broader EAA perspective, these interfaces also address understandability through consistent interaction models carried over from the Workflows page. Consistency itself is a significant accessibility-supporting property, particularly in complex developer-facing systems.

Code Structure and Reusability

Reusability is particularly pronounced in the implementation of the Workflow and Workflow Run pages because much of their functionality emerges from recombining shared abstractions rather than introducing page-specific isolated logic.

Most prominently, the Workflow Details component discussed previously is reused directly in the dedicated Workflow page, demonstrating intentional architectural reuse across overview-level and detail-level contexts.

Start date	End date	Duration	Run Type	State	Triggered via	Triggered by	Actions
31 Mar 2026 - 15:39	31 Mar 2026 - 15:39	18s			UI	airflow	View Logs
31 Mar 2026 - 13:48	31 Mar 2026 - 13:48	17s			UI	airflow	View Logs
31 Mar 2026 - 13:18	31 Mar 2026 - 13:18	19s			UI	airflow	View Logs
31 Mar 2026 - 13:06	31 Mar 2026 - 13:19	12m 46s			UI	airflow	View Logs
31 Mar 2026 - 13:04	31 Mar 2026 - 13:06	2m 12s			UI	airflow	View Logs
31 Mar 2026 - 12:58	31 Mar 2026 - 13:05	6m 51s			UI	airflow	View Logs
31 Mar 2026 - 12:53	31 Mar 2026 - 12:54	17s			UI	airflow	View Logs

Figure 5.6: Embedded Workflow Run History Supporting Navigation from Workflow Structure to Execution Inspection

Similarly, the DAG visualization infrastructure, although discussed later in more depth, is designed as a reusable graph-oriented component layer rather than a workflow-page-specific implementation. Task nodes, edge definitions, and associated state transformations are composed through reusable abstractions that support both structural workflow views and run-oriented visualization contexts.

Execution metadata views likewise follow reusable structural patterns rather than bespoke layouts. Field-value information cards, status representations, and grouped metadata sections reuse shared component primitives and formatting utilities established earlier in the chapter.

At the page level, these interfaces primarily orchestrate composition of reusable components and query-driven data sources. This preserves separation between presentation composition, data transformation logic, and infrastructural concerns.

This compositional approach also supported iterative refinement during implementation. Because major interaction regions were structurally decoupled through reusable components, improvements to visualization, metadata presentation, or execution-state representation could be made with localized changes rather than broad page refactoring.

Finally, this reuse has an important architectural implication beyond maintainability: it reinforces conceptual consistency. Similar workflow concepts are represented through the same components across contexts, reducing both implementation divergence and user-facing inconsistency.

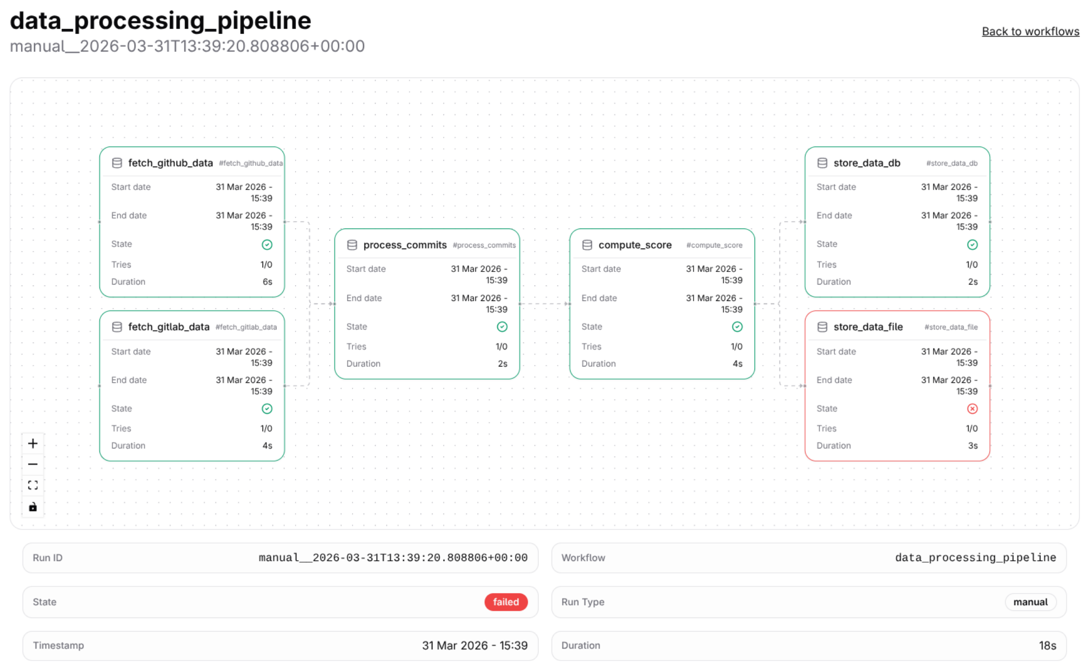


Figure 5.7: Workflow Run Detail View Combining Task-State Visualization and Execution Metadata Inspection

5.3.3 Task Instances and Workflow Runs Tables

User Experience and Interaction Design

The Workflow Runs and Task Instances tables extend the layered interaction model established in preceding interfaces by supporting detailed operational inspection through dense but structured tabular representations. While the workflow and workflow run pages emphasize contextual and structural understanding, these interfaces are optimized for search-oriented, filter-driven, and diagnostic interaction patterns.

The Workflow Runs table was designed around the principle that operational analysis frequently requires iterative narrowing rather than passive browsing. For this reason, filtering capabilities are treated as primary interaction mechanisms rather than secondary utilities. Figure 5.9 shows the overall structure of the Workflow Runs interface.

The filter model supports multiple dimensions of narrowing, including workflow selection, date-range restriction, state filtering, run-type filtering, and pattern-based run identification. Rather than relying exclusively on global search, explicit structured filters were preferred because they support more predictable interaction for technical monitoring tasks.

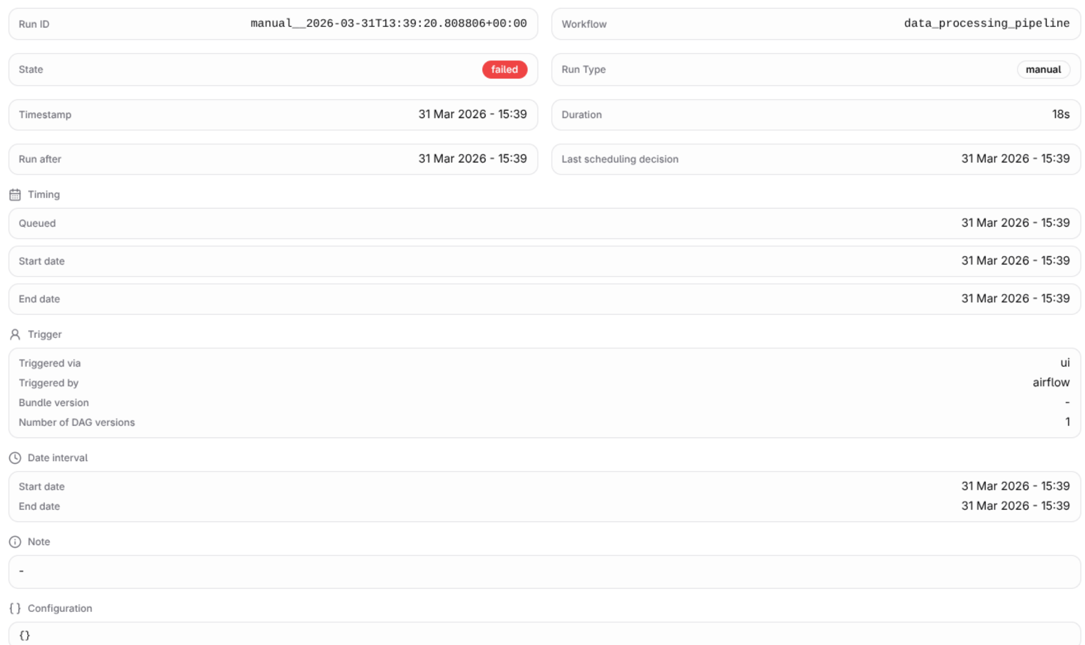


Figure 5.8: Structured Workflow Run Metadata View Showing Timing, Trigger, State, and Configuration Information

The state filtering mechanism shown in Figure 5.10 exemplifies this approach. Status-based filtering combines semantic labels with visual execution-state cues, supporting both efficient recognition and consistency with state representations used throughout the frontend.

Similarly, the integrated date-range filter shown in Figure 5.11 supports temporal narrowing for historical execution analysis, a recurring need in pipeline monitoring scenarios.

Column configurability and pagination were included to support differing user priorities and potentially large result sets. Rather than enforcing a rigid fixed representation, users can tailor information density to task context.

The Task Instances table adopts similar principles but emphasizes task-level operational diagnostics rather than run-level inspection. Figure 5.12 illustrates this interface.

Particular emphasis was placed on integrating action-oriented controls directly into tabular rows. Access to logs, task deletion actions, and task-state visibility are intentionally colocated with execution metadata to minimize interaction distance during diagnosis.

Together, these interfaces support an analytical interaction model complementary to the more structural interaction model of the workflow visualization pages.

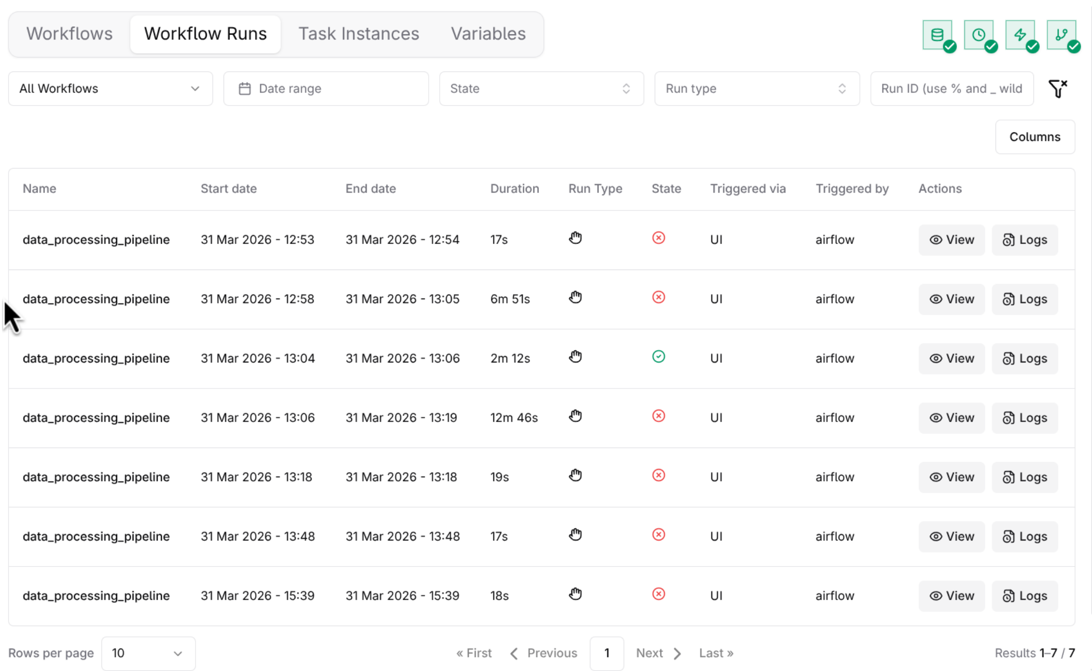


Figure 5.9: Workflow Runs Table Supporting Filter-Driven Inspection of Historical Execution Instances

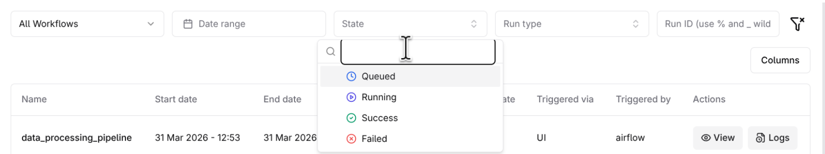


Figure 5.10: State-Based Filtering Interaction in the Workflow Runs Interface

Accessibility Considerations

Dense tabular interfaces pose distinctive accessibility challenges, particularly regarding navigation complexity, information overload, and operability across assistive interaction modes.

To address these concerns, the table implementations emphasize semantic structure, including accessible table relationships, predictable row-action positioning, and logical navigation order. This supports both screen-reader interpretation and efficient keyboard-based traversal.

Filtering controls were designed to satisfy EAA-aligned requirements related to operability and understandability. Filter controls are individually accessible, keyboard operable, and designed with explicit labeling and predictable interaction behavior. In particular, structured filters were favored over overly implicit interaction mechanisms because they reduce ambiguity.

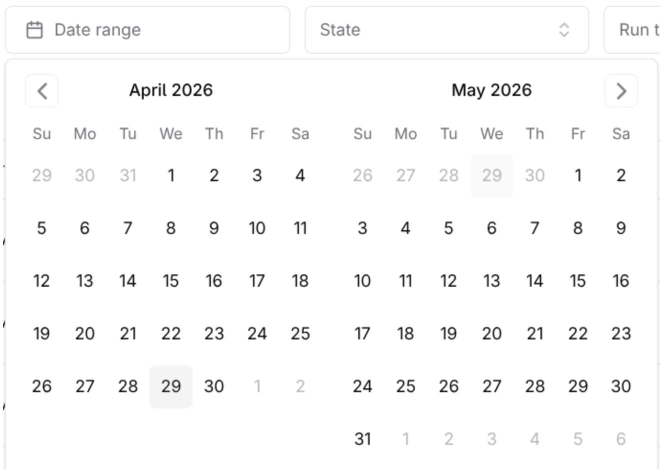


Figure 5.11: Date-Range Filtering for Temporal Restriction of Workflow Run Queries

Execution-state indicators within tables again use redundant encoding through text, iconography, and color semantics, preserving accessibility for users who cannot rely on visual color distinctions alone.

Pagination controls, column configuration interactions, and dropdown-based filters were designed with focus visibility and state awareness in mind, ensuring that interaction context remains clear when navigating complex data-heavy interfaces.

An important accessibility concern in diagnostic interfaces is cognitive accessibility. Dense monitoring tables can easily become overwhelming, and the combination of structured filtering, progressive narrowing, and customizable column density serves partly as a cognitive accessibility mechanism.

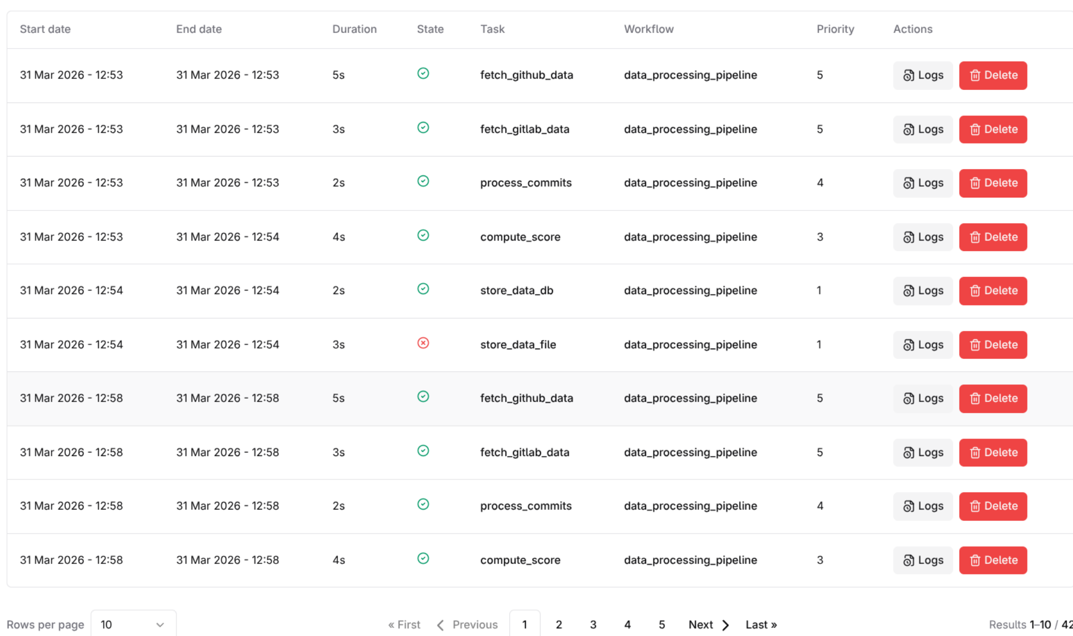
These considerations align not only with formal accessibility requirements but also with the broader usability objective of making operationally dense systems interpretable under varied user conditions.

Code Structure and Reusability

Reusability in these interfaces centers heavily around shared data-table abstractions and composable filter components.

Rather than implementing Workflow Runs and Task Instances as independent table implementations, both are built on shared reusable table infrastructure supporting column definitions, sorting behavior, pagination, row actions, and configurable filter composition.

This abstraction is particularly important because much of the behavioral complexity of these interfaces lies not in page-level logic but in reusable interaction



Start date	End date	Duration	State	Task	Workflow	Priority	Actions
31 Mar 2026 - 12:53	31 Mar 2026 - 12:53	5s		fetch_github_data	data_processing_pipeline	5	Logs Delete
31 Mar 2026 - 12:53	31 Mar 2026 - 12:53	3s		fetch_gitlab_data	data_processing_pipeline	5	Logs Delete
31 Mar 2026 - 12:53	31 Mar 2026 - 12:53	2s		process_commits	data_processing_pipeline	4	Logs Delete
31 Mar 2026 - 12:53	31 Mar 2026 - 12:54	4s		compute_score	data_processing_pipeline	3	Logs Delete
31 Mar 2026 - 12:54	31 Mar 2026 - 12:54	2s		store_data_db	data_processing_pipeline	1	Logs Delete
31 Mar 2026 - 12:54	31 Mar 2026 - 12:54	3s		store_data_file	data_processing_pipeline	1	Logs Delete
31 Mar 2026 - 12:58	31 Mar 2026 - 12:58	5s		fetch_github_data	data_processing_pipeline	5	Logs Delete
31 Mar 2026 - 12:58	31 Mar 2026 - 12:58	3s		fetch_gitlab_data	data_processing_pipeline	5	Logs Delete
31 Mar 2026 - 12:58	31 Mar 2026 - 12:58	2s		process_commits	data_processing_pipeline	4	Logs Delete
31 Mar 2026 - 12:58	31 Mar 2026 - 12:58	4s		compute_score	data_processing_pipeline	3	Logs Delete

Figure 5.12: Task Instances Table Supporting Task-Level Monitoring and Direct Log Access

patterns: filtering, row actions, column control, and server-driven pagination.

Filter controls themselves are implemented through reusable filter primitives rather than page-specific widgets. State filters, date-range selection, and search-driven filtering can therefore be reused across multiple tabular contexts.

The action cells embedded in rows similarly reuse shared controls and action patterns already used in other workflow interfaces, preserving both implementation consistency and conceptual continuity.

Integration with query-driven data loading further reinforces reuse. Filter state and query parameter transformation are intentionally decoupled from table rendering, allowing tables to remain primarily compositional consumers of shared infrastructural logic.

This approach proved particularly valuable as filtering requirements evolved during implementation, since enhancements could often be introduced at the reusable abstraction level rather than through duplicated per-page changes.

5.3.4 Variables Management Interface

User Experience and Interaction Design

The Variables interface differs substantially from the execution-oriented interfaces discussed previously because it supports configuration management rather than monitoring or orchestration control. Its design therefore prioritizes safe editing, structured validation, and controlled management interactions.

Figure 5.13 shows the primary variables management view.

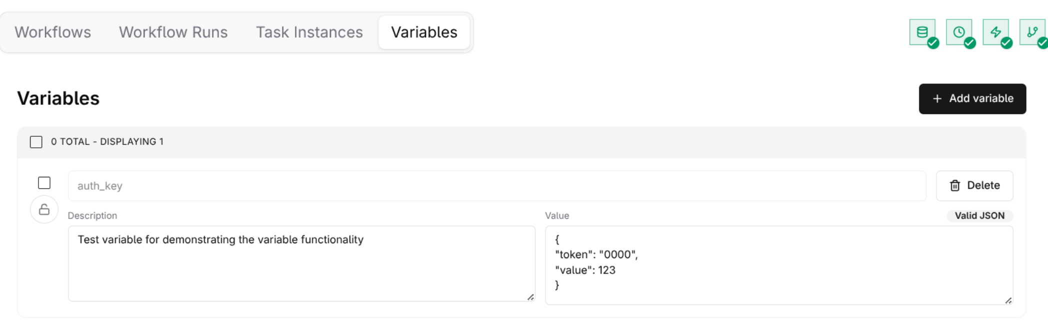


Figure 5.13: Variables Management Interface Supporting Structured Inspection and Editing of Airflow Variables

A central design objective was balancing editable flexibility with safeguards appropriate for potentially sensitive or operationally significant configuration data. For this reason, variables are presented through structured field groupings with explicit distinction between keys, descriptions, and values.

The add-variable interaction shown in Figure 5.14 was implemented through a modal pattern to focus user attention on the creation task while preserving contextual continuity.

Particular emphasis was placed on structured value handling, including support for JSON-oriented input and validation feedback. Since configuration errors may have downstream workflow consequences, validation feedback is designed to be immediate and explicit. Figure 5.15 illustrates invalid input handling.

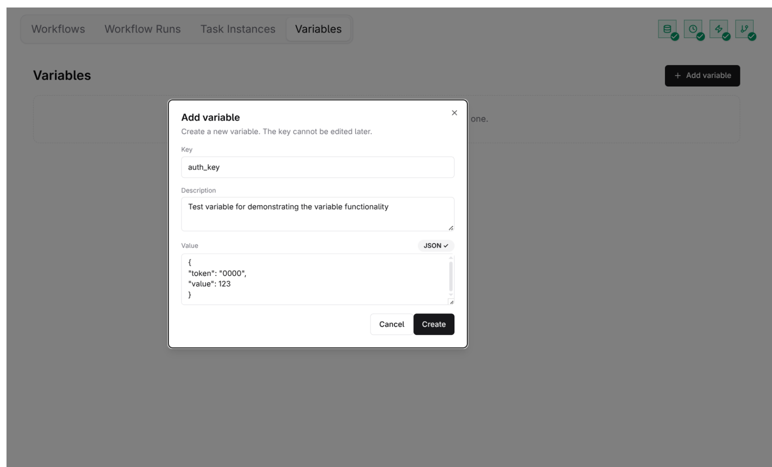


Figure 5.14: Variable Creation Dialog Supporting Structured Input and JSON-Aware Value Entry

Bulk interaction support, including selection-oriented actions and export capabilities shown in Figure 5.16, was introduced to support operational workflows involving multiple variables rather than isolated edits.

An important User Experience (UX) principle in this interface was explicitness over hidden automation. Configuration management interactions were intentionally designed to make system behavior predictable and user intent visible, particularly for creation, deletion, and validation-related actions.

Accessibility Considerations

Accessibility considerations for the Variables interface focus strongly on form accessibility, validation clarity, and modal interaction behavior.

Form controls follow conventional accessible interaction requirements, including associated labels, keyboard operability, visible focus indicators, and predictable field ordering. These properties are particularly important for configuration interfaces involving structured data entry.

Validation feedback was designed in alignment with EAA-related error identification and assistance principles. Invalid input is not indicated solely through color, but reinforced through explicit textual explanations and structured feedback messaging, as illustrated in Figure 5.15.

Modal interactions were implemented with attention to focus trapping, keyboard dismissal support where appropriate, and preservation of navigation context after dialog closure. These concerns are central for accessibility in modal-based interaction patterns.

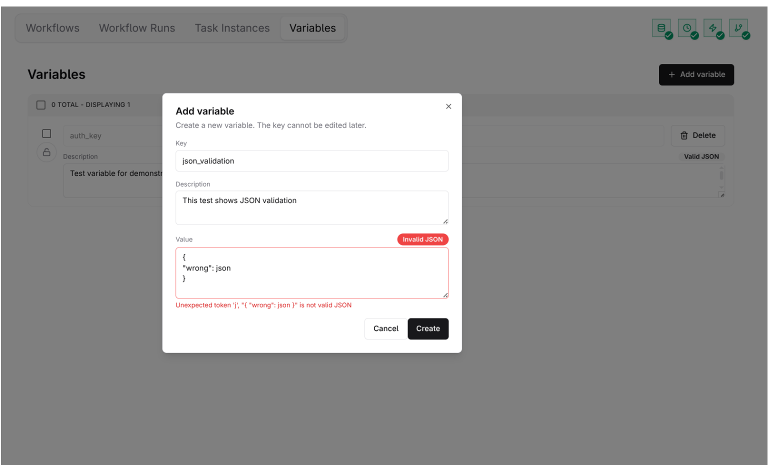


Figure 5.15: Inline Validation Feedback for Invalid JSON Input During Variable Creation

Bulk-selection interactions also required accessibility attention because multi-select behaviors can otherwise introduce ambiguity. Selection state is explicitly represented and associated actions remain contextually visible.

From a cognitive accessibility perspective, structured grouping of variable information and explicit action separation reduce ambiguity and help prevent configuration mistakes.

These considerations reinforce that even configuration-oriented developer interfaces benefit substantially from accessibility-oriented design treatment rather than assuming technical users can tolerate reduced accessibility support.

Code Structure and Reusability

Implementation of the Variables interface combines reusable form-oriented abstractions, validation logic, and domain-specific interaction components.

Form controls used in variable creation and editing rely heavily on shared input primitives and reusable validation-aware components rather than bespoke form implementations. This supports consistent interaction behavior and simplifies error handling.

JSON validation logic is intentionally encapsulated rather than embedded directly into modal presentation logic, preserving separation between validation concerns and interface rendering.

Selection handling and bulk actions reuse broader table-selection and action abstractions aligned with interaction patterns used elsewhere in the frontend, illustrating cross-feature reuse beyond purely visual components.

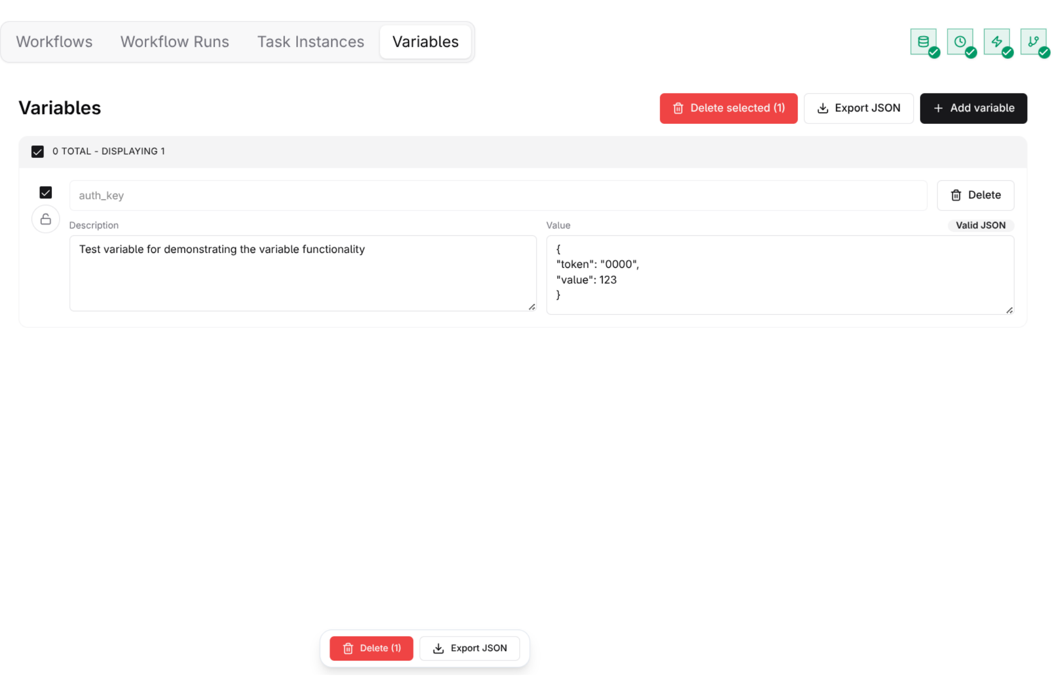


Figure 5.16: Selection-Based Bulk Actions Supporting Variable Export and Batch Operations

The modal-based creation flow similarly leverages reusable dialog infrastructure already employed for other interaction patterns, supporting consistency and reducing duplicated interaction logic.

This architecture also supports extensibility. Additional variable-related operations or richer validation behavior can be incorporated without fundamentally restructuring the interface.

5.3.5 Logs Interface

User Experience and Interaction Design

The Logs interface was designed as a focused diagnostic surface optimized for rapid interpretation of execution output rather than general document-style log browsing.

Figure 5.17 shows the implemented interface.

Rather than presenting raw unstructured log streams, the interface emphasizes structured readability through chronological segmentation, severity-oriented visual distinction, and source attribution. This reflects the objective of supporting diagnosis under time-sensitive operational conditions.

Severity labels provide immediate visual prioritization, while timestamp and

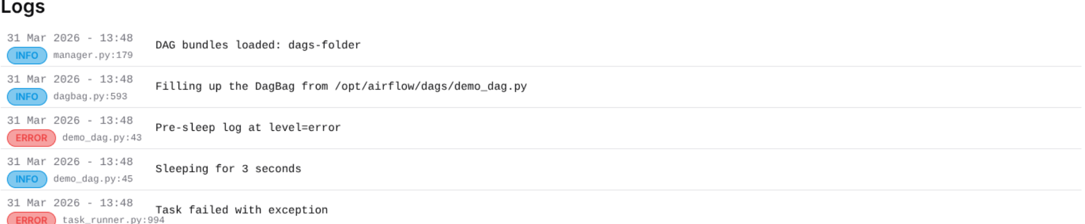


Figure 5.17: Structured Logs Interface with Severity-Based Highlighting and Chronological Presentation

source annotations preserve context needed for tracing execution behavior.

The interface intentionally avoids excessive visual ornamentation or terminal emulation metaphors, favoring clarity and scanability. This is particularly important because users often consult logs while simultaneously reasoning about workflow state in other interfaces.

A further design objective was continuity with the broader frontend interaction model. Access to logs originates directly from task and workflow run contexts, reinforcing a diagnostic progression from monitoring to detailed execution evidence.

Accessibility Considerations

Accessibility in log-oriented interfaces centers on readability, semantic clarity, and non-visual accessibility of severity information.

Severity indicators again avoid relying solely on color by combining visual highlighting with explicit textual severity labels. This is consistent with broader EAA-aligned accessibility treatment throughout the frontend.

Chronological structure and consistent entry formatting support both cognitive accessibility and assistive interpretation. Repetitive structural consistency is particularly valuable in content-dense diagnostic contexts.

Typography, spacing, and visual grouping were chosen to support sustained readability for extended log inspection sessions.

Because logs may contain technically dense content, the interface prioritizes perceivability and structured scanning over compact terminal-like density that could reduce accessibility.

These decisions reflect the broader accessibility principle that diagnostic interfaces should remain operable and interpretable under conditions of cognitive load, not merely under ideal interaction circumstances.

Code Structure and Reusability

Although visually simpler than other interfaces, the Logs interface also relies on reusable structural abstractions.

Log-entry rendering uses reusable severity-aware presentation patterns and formatting utilities rather than ad hoc string rendering. This supports consistent treatment of log metadata and severity states.

Severity labels, timestamp formatting, and source metadata reuse shared formatting and status-oriented component primitives used elsewhere in the frontend.

At the data level, log retrieval integrates through the same query-oriented infrastructural mechanisms established earlier, preserving consistency in loading, caching, and error handling behavior.

This reuse is particularly important because logs represent another specialized manifestation of broader status and state representation concerns already present across the frontend. Reusing these abstractions reinforces consistency while limiting duplicated logic.

5.4 Advanced Interaction and Execution Support

This section summarizes the advanced interaction mechanisms that complement the core workflow interfaces, focusing on graph-based visualization, asynchronous data handling, and consistent status communication.

5.4.1 DAG Visualization with ReactFlow

The DAG visualization is implemented using ReactFlow as a reusable graph rendering layer that translates Airflow task dependencies into an interactive node–edge representation. Tasks are mapped to nodes with enriched metadata such as execution state, duration, and retry count, while dependencies are rendered as directed edges. The visualization supports zooming, panning, and focused inspection, enabling users to explore both global workflow structure and local execution details.

A key design decision was to decouple layout computation from rendering. Layouts are derived from dependency graphs using deterministic positioning strategies, ensuring stable visual structures across reloads and preserving user mental models. Runtime state is overlaid onto nodes without modifying structural layout, allowing direct comparison between static topology and dynamic execution outcomes.

The visualization is implemented as a reusable component abstraction, supporting both workflow-level and run-level contexts with different data bindings. This reuse ensures consistency while allowing context-specific augmentation such as state-based coloring.

5.4.2 Handling Asynchronous Data and Polling

Given the inherently asynchronous nature of workflow execution, the frontend employs a polling-based synchronization strategy built on TanStack Query. Query intervals are configured according to resource volatility, with shorter intervals for actively executing workflows and longer intervals for static metadata.

Polling is combined with targeted cache invalidation following mutations such as triggering a workflow run. This ensures timely updates while minimizing unnecessary network requests. Loading, stale, and error states are consistently handled through shared UI patterns, ensuring predictable behavior across interfaces.

To avoid misleading interface states, optimistic updates are not used for critical execution-related actions. Instead, explicit feedback mechanisms such as notifications are used to communicate outcomes.

5.4.3 Error, Warning, and Status Representation

Consistent status representation is a cross-cutting concern across all interfaces. Execution states such as success, failure, and running are encoded through a combination of iconography, textual labels, and color semantics.

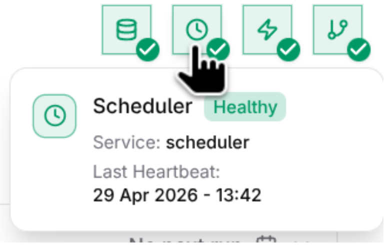


Figure 5.18: Reusable Status Indicator Combining Iconography, Color, and Textual Semantics

User actions are accompanied by transient feedback through non-blocking notification elements.

Micro-interactions further enhance usability, including tooltip-based affordances and copy-to-clipboard functionality.

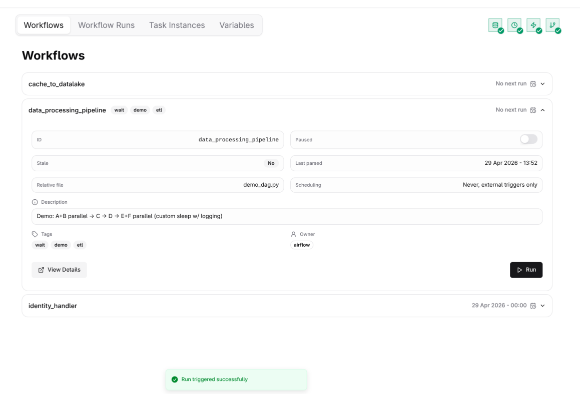


Figure 5.19: Snackbar Notification for Immediate Feedback on User Actions

5.5 Testing and Backend Support

This section outlines the testing strategy for the frontend, backend adaptations, and the use of representative workflows for validation.

5.5.1 Frontend Testing Strategy and Implementation

Frontend testing focuses on component-level correctness and interaction reliability. Reusable components such as tables, forms, and status indicators are tested in isolation under varying states. Integration tests cover critical workflows such as execution triggering, navigation, and filtering.

Mocked API responses are used to simulate backend behavior, enabling deterministic testing of asynchronous interactions. The testing approach prioritizes robustness of interaction contracts over exhaustive visual regression.

5.5.2 Backend Adjustments Supporting the Frontend

Several backend adjustments were introduced to align with frontend requirements, including standardized endpoint structures, enriched response payloads, and support for filtering parameters. These changes reduce frontend transformation complexity and improve system cohesion.

Authentication endpoints were aligned with token-based interaction patterns, simplifying client integration and ensuring consistent authorization handling.

5.5.3 Example Workflows and Demonstration Pipelines

Representative Airflow workflows were implemented to validate the frontend under realistic conditions. These include multi-branch pipelines, parallel task exe-

Figure 5.20: Icon with Tooltip Providing Contextual Information

Figure 5.21: Copy-to-Clipboard Interaction for Efficient Identifier Handling

cution, and controlled failure scenarios.

Such workflows were essential for validating visualization behavior, filtering mechanisms, and diagnostic interfaces, particularly for error handling and log inspection.

5.6 Implementation Decisions and Trade-Offs

The implementation reflects several deliberate trade-offs.

The emphasis on reusable components increased initial development effort but significantly improved consistency and maintainability.

Generated API clients improved type safety and reduced integration errors, at the cost of dependency on API specification quality.

Finally, graph-based visualization provides strong structural insight but requires

additional accessibility considerations and complementary representations.

Overall, the implementation prioritizes pragmatic, maintainable solutions aligned with the scope and objectives of the thesis.

6 Evaluation

6.1 Evaluation Goals and Research Questions

The evaluation aims to systematically assess the effectiveness, usability, and practical utility of the developed frontend in comparison to existing solutions, particularly the native Apache Airflow interface. Given the design objectives outlined in Chapter 5, the evaluation focuses on determining whether the proposed system improves interaction quality, reduces cognitive load, and enhances workflow comprehension and debugging efficiency.

The evaluation is guided by the following research questions:

- To what extent does the proposed frontend improve usability compared to the native Airflow interface?
- Does the interface reduce cognitive load and improve learnability for users with varying levels of Airflow experience?
- How effectively does the interface support common workflow management and debugging tasks?
- What qualitative perceptions and behavioral patterns emerge during interaction with the system?

These questions reflect both quantitative performance considerations and qualitative user experience dimensions, aligning with the dual focus on usability and practical applicability.

6.2 Evaluation Methodology

6.2.1 Study Design

A mixed-methods user study was conducted, combining task-based evaluation with semi-structured interviews and observational analysis. Participants interacted with both the native Airflow interface and the developed frontend in a controlled setting. To mitigate ordering effects, the interface exposure order was

alternated across participants.

The study employed a think-aloud protocol, enabling participants to verbalize their reasoning and perceptions during task execution. This approach provided insight into cognitive processes, usability challenges, and interaction strategies.

Each session included task execution, followed by a reflective interview capturing subjective impressions and comparative judgments.

6.2.2 Participant Selection and Recruitment

A total of $n = 18$ participants were recruited, representing a range of technical backgrounds and experience levels. The participant pool primarily consisted of individuals with software development experience, including students and professionals in computer science and related fields.

Figure 6.1 illustrates the age distribution of participants, while Figure 6.2 shows the relationship between general development experience and Airflow familiarity.

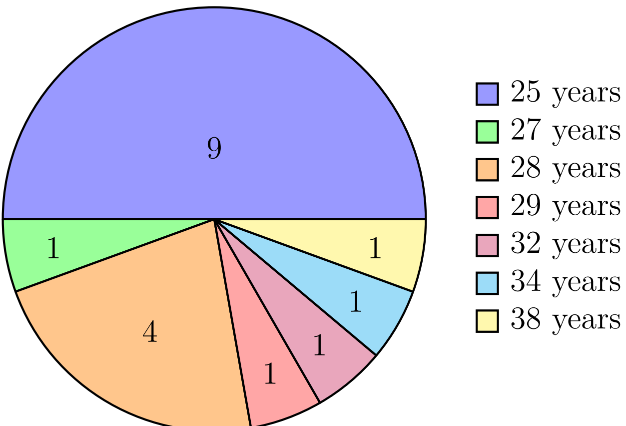


Figure 6.1: Age distribution of the participants in the user study. The majority of participants were 25 years old, while the remaining participants were distributed across higher age groups.

This distribution ensures that the evaluation captures both novice and experienced perspectives, particularly relevant for assessing learnability and accessibility.

6.2.3 Task-Based Evaluation Scenarios

Participants were asked to complete a series of representative tasks reflecting common workflow management scenarios, including:

- Inspecting workflow structures and dependencies

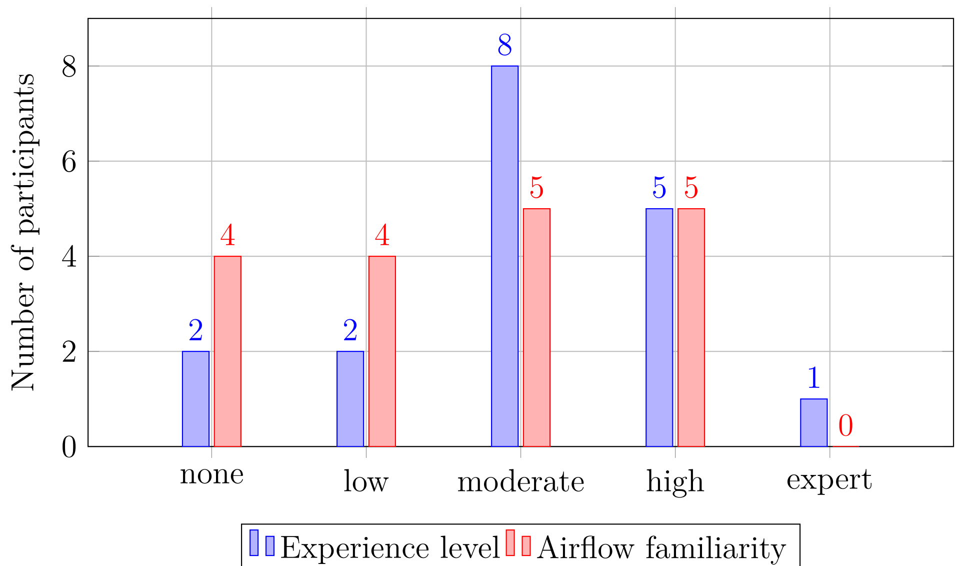


Figure 6.2: Comparison of participants' general software development experience and Apache Airflow familiarity across defined proficiency levels. While development experience is concentrated in the intermediate and advanced categories, familiarity with Airflow is more evenly distributed, with a noticeable proportion of participants reporting no or low prior exposure.

- Identifying failed tasks and diagnosing issues
- Accessing and interpreting logs
- Triggering workflow executions
- Modifying configuration parameters

These tasks were selected to cover both monitoring and configuration-oriented interactions. Observations from individual participants indicate that the custom interface enabled more efficient task completion and reduced reliance on exploratory navigation, particularly for debugging-related tasks.

6.2.4 Data Collection and Analysis Approach

Data collection combined quantitative and qualitative methods. Quantitative data included task completion observations and structured ratings of usability dimensions such as navigation ease, cognitive load, and learnability.

Qualitative data was collected through think-aloud protocols, interview responses, and observational notes. These data sources were analyzed using thematic analysis, identifying recurring patterns and emergent themes across participants.

Figure 6.3 provides an overview of interview durations.

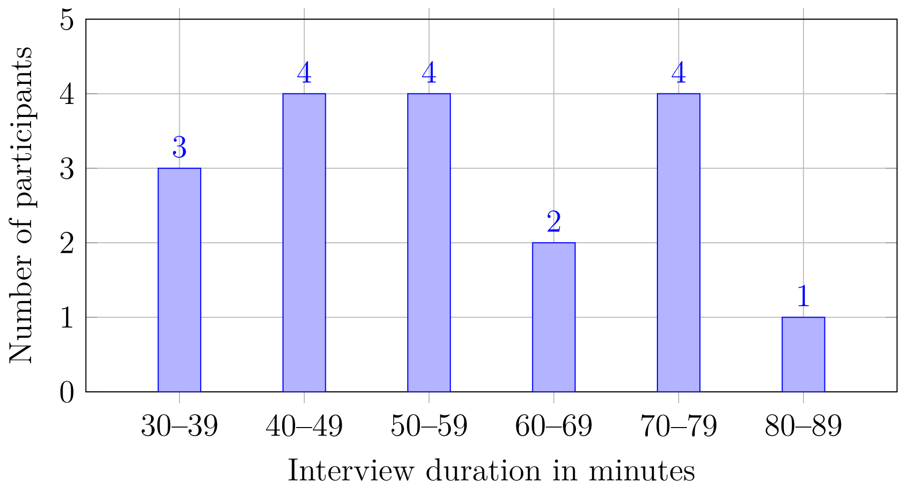


Figure 6.3: Distribution of interview durations grouped into ten-minute intervals. Most interviews lasted between 40 and 79 minutes, with only one interview reaching the 80-minute interval.

6.3 Comparative Evaluation Setup

6.3.1 Comparison with Airflow's Native Interface

The evaluation employs a direct comparative design between the native Airflow interface and the developed frontend. Participants interacted with both systems using identical task scenarios, enabling direct comparison of usability and performance characteristics.

The native interface serves as a baseline representing established industry practice, while the developed frontend introduces improvements in visualization, interaction structure, and information accessibility.

6.3.2 Quantitative and Qualitative Evaluation Metrics

Evaluation metrics were divided into quantitative and qualitative categories.

Quantitative measures included:

- Task completion success
- Interaction efficiency indicators
- Structured ratings for usability dimensions

Qualitative measures included:

- Participant perceptions and preferences
- Observed interaction patterns

- Emergent thematic insights

Figure 6.4 summarizes aggregated perception scores across key usability dimensions.

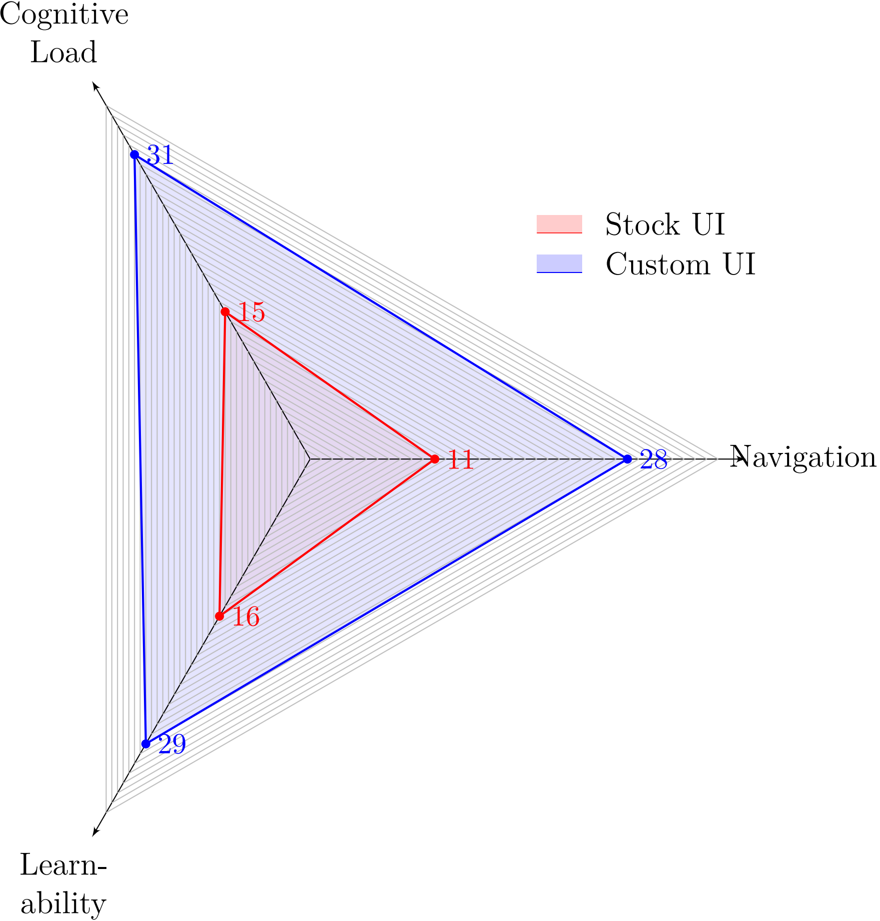


Figure 6.4: Participant perception of the stock and custom interfaces across navigation ease, cognitive load, and learnability. Aggregated scores show that the custom interface is consistently perceived as superior across all evaluated dimensions.

6.4 Evaluation Results

6.4.1 Usability and Effectiveness Results

The evaluation results indicate a consistent advantage of the custom interface across all measured usability dimensions. Participants reported improved navigation ease, reduced cognitive load, and higher perceived learnability.

Figure 6.5 shows that the majority of participants preferred the custom interface for daily use.

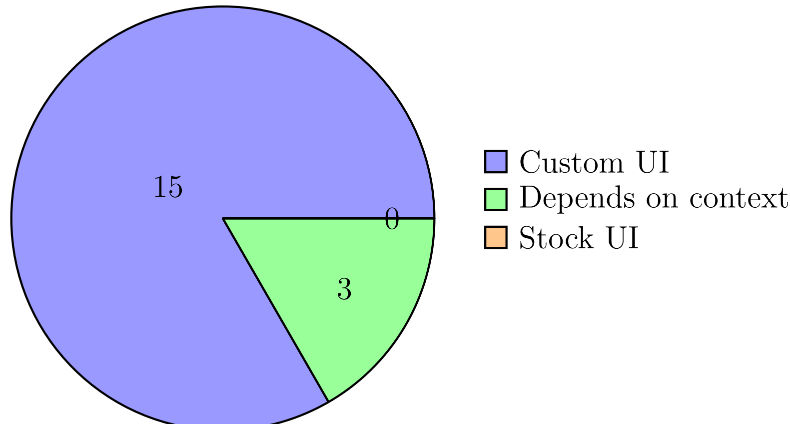


Figure 6.5: Preferred user interface for daily use among participants. The majority of participants expressed a clear preference for the custom interface, while a small subset indicated that their preference depends on the specific context. No participant preferred the stock interface.

These findings suggest that the design objectives related to usability and efficiency were largely achieved.

6.4.2 Participant Feedback and Observed Themes

Thematic analysis revealed several recurring themes across participant interviews. Learnability and efficiency were identified in all interviews, while visibility and trust also emerged as dominant factors.

Participants frequently highlighted improved mental models of workflows, enhanced debugging capabilities, and reduced context switching as key benefits.

Observed interaction styles further support these findings. As shown in Figure 6.7, confident and exploratory behaviors dominated, indicating low interaction friction.

6.4.3 Interpretation of Findings

The results indicate that the frontend successfully addresses key limitations of the native Airflow interface. Improvements in visualization, structured navigation, and integrated diagnostics contribute to enhanced usability and efficiency.

The prominence of learnability and visibility suggests that the interface supports both new and experienced users effectively. The absence of negative sentiment, as shown in Figure 6.8, further reinforces the overall positive reception.

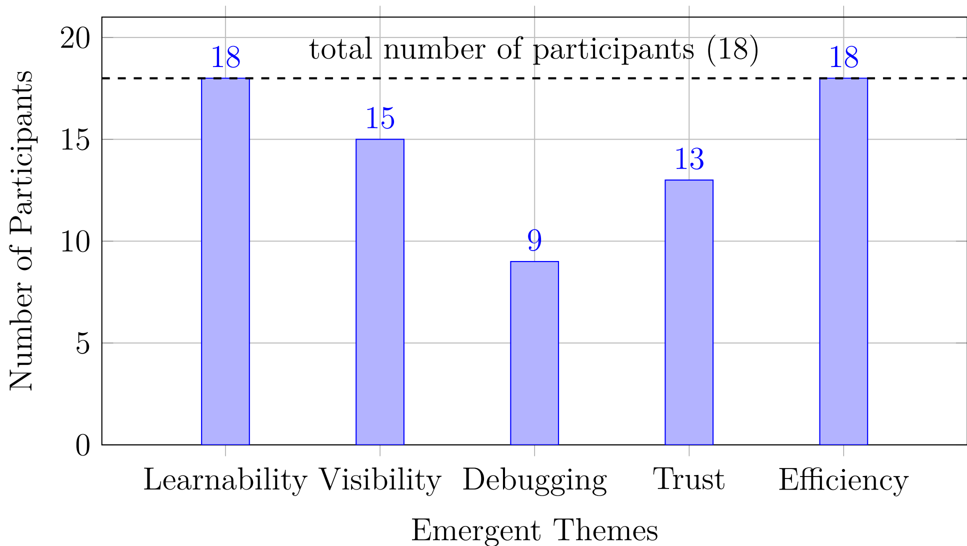


Figure 6.6: Frequency of emergent themes across all participant interviews. Learnability and efficiency were identified in all interviews, while visibility and trust were also prominent. Debugging emerged less consistently but remained a significant theme. The dashed line indicates the total number of participants ($n = 18$).

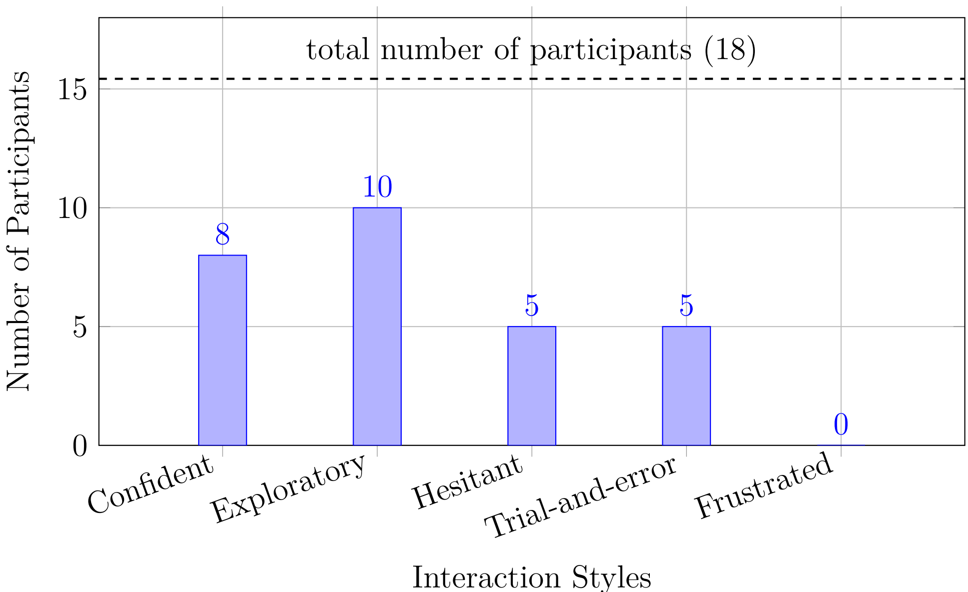


Figure 6.7: Distribution of observed interaction styles during the user study. Exploratory and confident interaction patterns were most common, while hesitant and trial-and-error behaviors occurred less frequently. No participant exhibited sustained frustration during task execution. The dashed line indicates the total number of participants ($n = 18$).

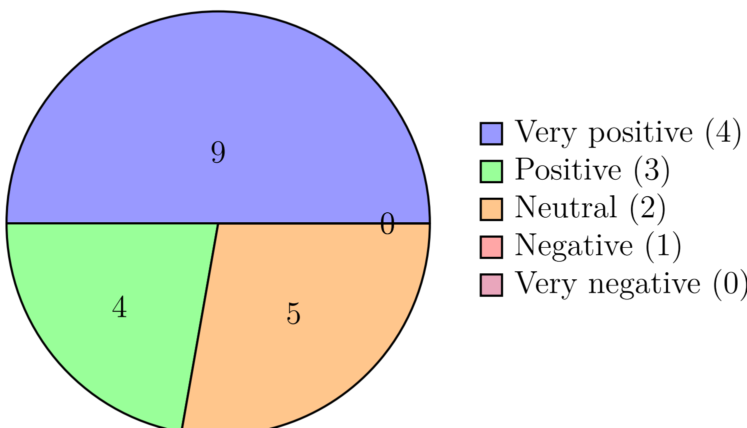


Figure 6.8: Distribution of participant sentiment toward the custom interface. The majority of participants reported very positive or positive impressions, while the remaining participants expressed neutral sentiment. No negative or very negative perceptions were observed.

Overall, the findings validate the design approach and demonstrate that targeted frontend improvements can significantly enhance interaction quality in workflow orchestration systems.

6.5 Threats to Validity and Limitations

Despite the positive results, several limitations must be acknowledged.

First, the participant sample, while diverse in experience, is limited in size ($n = 18$) and may not fully represent all user groups, particularly domain experts with extensive production experience.

Second, the controlled study environment may not fully capture real-world usage conditions, including long-term interaction patterns and operational stress scenarios.

Third, the evaluation focuses primarily on usability and interaction quality rather than system performance or scalability.

Finally, the comparison is limited to the native Airflow interface and does not include other workflow orchestration tools, which may offer different interaction paradigms.

These limitations suggest that while the results are indicative, further evaluation in real-world deployment contexts would strengthen the generalizability of the findings.

Additional qualitative details and individual participant insights are provided

in Appendix B, supporting transparency and reproducibility of the evaluation process.

7 Conclusion

This thesis addresses the integration of workflow orchestration functionality into the MECOIS frontend with the goal of providing a cohesive and user-centered interaction model for managing data processing pipelines. Starting from the observation that Apache Airflow provides a powerful backend scheduler but an insufficiently project-specific user interface, the work focused on the design, implementation, and evaluation of a dedicated frontend solution embedded directly into the existing MECOIS platform. The central objective was therefore not just to connect a frontend to an external API, but to create an interaction layer through which workflow-related operations can be performed in a way that is more consistent with the surrounding system, more accessible to its users, and better aligned with the concrete requirements of the project.

The resulting contribution is an engineering artifact that demonstrates how orchestration capabilities offered by a general-purpose scheduler can be adapted to a domain-specific frontend environment. The implemented solution provides integrated support for central workflow interactions, such as triggering workflows, inspecting workflow and task states, accessing logs and error information, and understanding task dependencies through graphical visualization. In this way, the work closes part of the gap between backend orchestration capability and frontend usability that motivated the thesis. Rather than requiring users to leave the platform and switch to Airflow's native interface, the developed frontend consolidates relevant orchestration activities inside MECOIS and thereby contributes to a more coherent workflow-oriented user experience.

From a research-oriented perspective, the thesis shows that the integration of workflow orchestration into an existing application frontend is not only a question of technical connectivity, but also of interface structure, information presentation, and operational fit. The work indicates that the usefulness of such an integration depends strongly on whether the frontend presents execution states clearly, supports efficient transitions between overview and detail, and makes relevant control and inspection actions directly available in the context in which users already work. In this sense, the thesis contributes a concrete case showing how scheduler functionality can be transformed into a more usable and project-adapted frontend

interaction model.

From an engineering-oriented perspective, the thesis demonstrates the feasibility of implementing such a solution as a working prototype inside an existing software environment. The frontend was not treated as an isolated mock-up, but as an integrated component whose design had to account for existing architecture, frontend conventions, backend interfaces, accessibility expectations, and testing considerations. This practical orientation is one of the central outcomes of the work. The thesis does not end with a conceptual discussion of desirable interface properties, but with a prototype artifact that can be used, assessed, and further developed within the MECOIS context.

The evaluation of the developed frontend further supports this contribution. The comparison with Airflow's native interface and the collection of qualitative user feedback provided evidence that the integrated solution offers practical benefits in terms of workflow interaction and operational clarity, while also revealing concrete areas in which the initial implementation could be improved. This is an important result in its own right. It shows that the value of the artifact lies not only in the features it implements, but also in the way the implementation made strengths and weaknesses visible through practical use. The evaluation therefore served both as an assessment of the artifact and as a mechanism for refining it.

Several lessons emerged from the design and implementation process. First, interfaces for workflow orchestration must place strong emphasis on the clear communication of execution state, failure conditions, and available actions. Second, graphical representations of workflow structure are useful, but only when embedded into an interaction model that supports orientation and focused inspection. Third, seemingly secondary concerns such as accessibility, status indication, and visual consistency are in practice central to the usability of technical interfaces. Finally, the work confirmed that integrating orchestration functionality into the existing platform context creates practical value for MECOIS by reducing context switching and providing a reusable basis for workflow-related frontend development within the system.

At the same time, the findings of this thesis should be interpreted in line with its intended scope. The work was not designed to produce universally generalizable guidelines about workflow interfaces, but rather to design, implement, and assess a concrete solution in a specific project environment. Its main contribution is therefore practical and design-oriented: it provides a validated prototype, a structured implementation approach, and a set of lessons that can inform further development in MECOIS and similar systems.

7.1 Limitations

The thesis has several limitations that follow directly from its scope as an engineering thesis. First, the developed solution is a prototype and should be understood as such. Although it was designed and implemented with practical use in mind, it does not represent a finalized production subsystem that exhaustively covers all possible orchestration-related needs of the platform. Its purpose is to demonstrate feasibility, support realistic use, and provide a sound basis for evaluation and further refinement.

Second, the scope of the implementation was intentionally bounded. The thesis focused on the frontend-facing integration layer between users, MECOIS, and the Airflow-based scheduling infrastructure. It did not aim to replace Apache Airflow as backend orchestrator, redesign the entire MECOIS platform, or shift the center of gravity of the work toward large-scale backend restructuring. Likewise, some capabilities that were discussed during the project remained outside the implemented core because they would have extended the thesis beyond its primary objective. These include, for example, broader support for Airflow entities and views beyond the main workflow-focused interaction flow, features such as favorites, pools and providers views, XCom browsing, code-file viewing, or expanded human-in-the-loop and notification-related functionality, which were considered during the project but not treated as part of the essential prototype scope.

Third, the evaluation is limited by its qualitative and context-specific character. The findings are grounded in the MECOIS environment and in feedback from participants familiar with the project or closely related development contexts. The evaluation therefore provides strong insight into practical usefulness and concrete refinement needs, but it does not justify claims of universal generalizability across all workflow tools, user groups, or application domains. The conclusions of the thesis should accordingly be understood as empirically informed within the project context rather than as broadly general scientific claims.

7.2 Future work

The most immediate avenue for future work lies in the continued refinement of the developed frontend within MECOIS. The user feedback collected during the project already identified several concrete opportunities for improvement, including stronger and more indicative status communication, further refinement of the DAG visualization, improved focus on expanded or currently relevant interface areas, and clearer handling of certain workflow-related actions. These findings provide a natural starting point for the next iteration of the frontend and show that the current prototype can serve as a stable basis for targeted usability improvements.

A second direction concerns the extension of workflow-related functionality within the same integrated frontend model. While the thesis deliberately concentrated on the most relevant orchestration interactions, future development could broaden the range of supported workflow-management capabilities and extend the frontend toward additional operational views or Airflow-related entities where these are useful for MECOIS in practice.

A third direction concerns internationalization. Since the project already established the foundations and infrastructure for localization, future work should extend this support across the remaining parts of the MECOIS frontend so that the mechanisms introduced in this thesis can be adopted consistently throughout the broader application.

Finally, future evaluation should continue to focus on the needs of MECOIS developers and closely related technical users in realistic usage scenarios. In particular, it would be valuable to observe the frontend over a longer period of actual project use and to assess how the implemented interaction patterns perform once they become part of day-to-day workflow management. This would complement the current qualitative evaluation by showing how the solution behaves in continued practical use and how the identified improvement opportunities should be prioritized in future iterations.

Overall, this thesis provides a practical foundation for embedding workflow orchestration functionality into the MECOIS frontend. By designing, implementing, and evaluating an integrated prototype, it demonstrates that project-specific orchestration interfaces can improve the usability and coherence of workflow interaction in an existing software platform. At the same time, it opens a clear path for continued refinement and further frontend-supported orchestration work within MECOIS.

Appendices

A Tool Disclaimer

During the preparation of this work, the author used ChatGPT-5.5 as well as Overleaf's AI Assistant (2026) in order to improve readability and flow of language. After using these tools, the manuscript was carefully reviewed and the content was edited as needed. No tools or services were used for content generation. The content of this thesis was entirely created by the author, who has ensured that all material presented reflects their original work and research. The author takes full responsibility for the content of the thesis.

B Interview Summaries

B.1 Participant 1

1. Participant Metadata

Participant ID	01	**Age**	25
Date	14.02.2026	**Duration**	57 minutes
Prof. Background	Full Stack Developer / MSc Computer Science Student	**Experience**	3 years

Experience Level
☐ None ☐ Beginner ☒ Intermediate ☐ Advanced

Airflow Familiarity
☐ None ☐ Beginner ☒ Intermediate ☐ Advanced

2. Study Setup

UI Order Used
☒ Stock → Custom
☐ Custom → Stock

☒ Think-Aloud Used

Tasks Completed
☒ Inspect failed tasks
☒ Find logs
☒ Trigger workflow
☒ Modify parameter
☒ Understand DAG structure
☒ Identify dependencies

3. Interview Summary

The participant completed all tasks without assistance and emphasized the custom interface reduced cognitive effort compared to the reference interface. In particular, visual workflow states and searchable tables improved orientation. The participant praised the debugging support and highlighted that direct access to logs and workflow structure made fault tracing substantially easier. Some suggestions focused on stronger visual differentiation for states and table discoverability.

Emergent Theme Codes
☒ Learnability
☒ Visibility
☒ Debugging
☐ Trust
☒ Efficiency

4. Participant Perceptions

Navigation Ease

Stock ☒ Low ☐ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Perceived Cognitive Load

Stock ☒ Low ☐ Med ☐ High
Custom ☐ Low ☒ Med ☐ High

Learnability

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Preferred UI for Daily Work

☐ Stock
☒ Custom
☐ Depends

Key Benefits Mentioned

☒ Better DAG overview
☒ Faster debugging
☒ Reduced context switching
☒ Better mental model

5. Observational Notes

Behavioral Observations

Hesitation points: Minor hesitation before locating parameter edit.

Interaction Style

☒ Confident
☒ Exploratory
☐ Hesitant
☐ Trial-and-error
☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues

– Workflow state colors were not sufficiently indicative.
– Running and queued colors were too similar.
– Column visibility control was not immediately recognizable.
– Table search and sorting options were not easily discoverable.
– Total entry counts in tables were missing.

Priority of Most Relevant Issue

☐ Low ☒ Medium ☐ High

Participant Suggestions

– Strengthen semantic color coding for workflow states.
– Make columns button more prominent.
– Add sortable tables and stronger search capabilities.
– Show total entry counts in data tables.
– Support direct workflow execution from overview pages.
– Add favorite workflows for frequently used pipelines.

Overall Sentiment Toward Custom UI

Very Negative — Negative — Neutral — Positive — **● Very Positive**

B.2 Participant 2

1. Participant Metadata

Participant ID	02	**Age**	25
Date	15.02.2026	**Duration**	43 minutes
Prof. Background	iOS App Developer / MSc Computer Science Student	**Experience**	2 years

Experience Level
☐ None ☐ Beginner ☒ Intermediate ☐ Advanced

Airflow Familiarity
☐ None ☒ Beginner ☐ Intermediate ☐ Advanced

2. Study Setup

UI Order Used
☐ Stock → Custom
☒ Custom → Stock

☒ Think-Aloud Used

Tasks Completed
☒ Inspect failed tasks
☒ Find logs
☒ Trigger workflow
☒ Modify parameter
☒ Understand DAG structure
☒ Identify dependencies

3. Interview Summary

The participant completed all tasks successfully and reacted particularly positively to the DAG visualization and fullscreen interaction. Learnability was reported as high, though some uncertainty arose regarding advanced controls such as column visibility. The participant emphasized visual clarity and workflow navigation as major improvements, while debugging aspects were discussed less prominently.

Emergent Theme Codes
☒ Learnability
☒ Visibility
☐ Debugging
☐ Trust
☒ Efficiency

4. Participant Perceptions

Navigation Ease

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Perceived Cognitive Load

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☒ Med ☐ High

Learnability

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Preferred UI for Daily Work

☐ Stock
☒ Custom
☐ Depends

Key Benefits Mentioned

☒ Better DAG overview
☐ Faster debugging
☒ Reduced context switching
☒ Better mental model

5. Observational Notes

Behavioral Observations

Observed behavior: Explored visualization controls before proceeding to later tasks. Brief pause when interpreting column controls.

Interaction Style

☐ Confident
☒ Exploratory
☐ Hesitant
☒ Trial-and-error
☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues

- Embedded DAG view felt spatially constrained.
- Maximum zoom-out was too limited.
- Workflow details icon did not sufficiently reflect selected state.
- Expanded sections lacked focus emphasis.
- Dark mode language icon had visibility issues.
- Description field appeared editable although read-only.

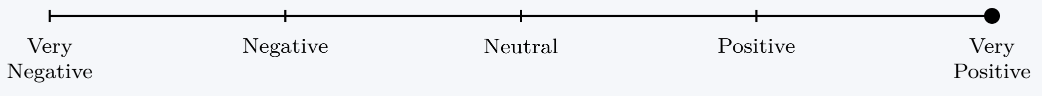
Priority of Most Relevant Issue

☐ Low ☒ Medium ☐ High

Participant Suggestions

- Add fullscreen DAG visualization mode.
- Improve zoom limits and graph navigation.
- Better link workflow detail icons to state context.
- Increase focus highlighting for expanded sections.
- Improve dark mode icon contrast.
- Add confirmation dialogs for destructive bulk actions.

Overall Sentiment Toward Custom UI



B.3 Participant 3

1. Participant Metadata

Participant ID	03	**Age**	27
Date	16.02.2026	**Duration**	71 minutes
Prof. Background	Backend and Pipeline Developer / MSc Computer Science Student	**Experience**	2 years

Experience Level
☐ None ☐ Beginner ☒ Intermediate ☐ Advanced

Airflow Familiarity
☐ None ☐ Beginner ☐ Intermediate ☒ Advanced

2. Study Setup

UI Order Used
☒ Stock → Custom
☐ Custom → Stock

☒ Think-Aloud Used

Tasks Completed
☒ Inspect failed tasks
☒ Find logs
☒ Trigger workflow
☒ Modify parameter
☒ Understand DAG structure
☒ Identify dependencies

3. Interview Summary

The participant approached the evaluation from a strong backend perspective and completed all tasks efficiently. Advanced functionality such as aggregated logs, variables, and workflow structure inspection were discussed extensively. The participant particularly emphasized debugging support and trust in the system through better transparency of task state and dependencies.

Emergent Theme Codes
☒ Learnability
☒ Visibility
☒ Debugging
☒ Trust
☒ Efficiency

4. Participant Perceptions

Navigation Ease

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Perceived Cognitive Load

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Learnability

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Preferred UI for Daily Work

☐ Stock
☒ Custom
☐ Depends

Key Benefits Mentioned

☒ Better DAG overview
☒ Faster debugging
☒ Reduced context switching
☒ Better mental model

5. Observational Notes

Behavioral Observations

Observed behavior: Moved through tasks systematically with few interruptions. Occasionally verbalized comparisons to known tooling.

Interaction Style

☒ Confident
☒ Exploratory
☐ Hesitant
☐ Trial-and-error
☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues

- Advanced Airflow-oriented functionality was partially missing.
- JSON parameter inputs lacked syntax guidance.
- Paused button behavior appeared inconsistent.
- Empty start-date handling was unclear.
- Filtering capabilities could be stronger for expert workflows.

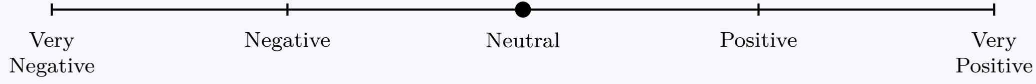
Priority of Most Relevant Issue

☐ Low ☒ Medium ☐ High

Participant Suggestions

- Support variable import functionality.
- Add Pools and Providers view.
- Allow browsing XComs.
- Separate singular and aggregated log views.
- Provide access to underlying workflow code files.
- Add example syntax hints for JSON input.
- Improve advanced filtering in data tables.

Overall Sentiment Toward Custom UI



B.4 Participant 4

1. Participant Metadata

Participant ID	04	**Age**	25
Date	17.02.2026	**Duration**	52 minutes
Prof. Background	System Administrator	**Experience**	5 years

Experience Level

☐ None ☐ Beginner ☐ Intermediate ☒ Advanced

Airflow Familiarity

☐ None ☐ Beginner ☒ Intermediate ☐ Advanced

2. Study Setup

UI Order Used

☐ Stock → Custom

☒ Custom → Stock

☒ Think-Aloud Used

Tasks Completed

☒ Inspect failed tasks

☒ Find logs

☒ Trigger workflow

☒ Modify parameter

☒ Understand DAG structure

☒ Identify dependencies

3. Interview Summary

All tasks were completed successfully, with the participant emphasizing improved orientation and state visibility in the custom interface. The participant highlighted efficiency gains through easier access to operational information and fewer navigation steps. Some suggestions focused on table interactions and graph navigation rather than core workflow execution.

Emergent Theme Codes

☒ Learnability

☒ Visibility

☐ Debugging

☒ Trust

☒ Efficiency

4. Participant Perceptions

Navigation Ease

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☒ Med ☐ High

Perceived Cognitive Load

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Learnability

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☒ Med ☐ High

Preferred UI for Daily Work

☐ Stock
☒ Custom
☐ Depends

Key Benefits Mentioned

☒ Better DAG overview
☐ Faster debugging
☒ Reduced context switching
☒ Better mental model

5. Observational Notes

Behavioral Observations

Observed behavior: Careful but steady interaction. Slight trial-and-error behavior while using filtering controls.

Interaction Style

☐ Confident
☒ Exploratory
☐ Hesitant
☒ Trial-and-error
☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues

– State colors could better distinguish runtime states.
– Horizontal table navigation was cumbersome.
– Workflow detail icon-state linkage was weak.
– Logs navigation required too many interaction steps.
– Large result sets lacked overview information.

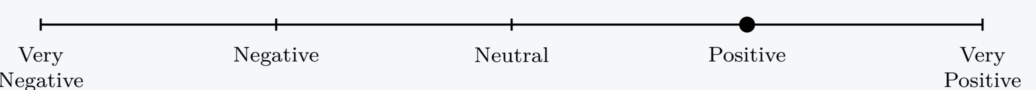
Priority of Most Relevant Issue

☐ Low ☒ Medium ☐ High

Participant Suggestions

– Improve distinction between queued and running states.
– Add horizontal scrolling for wider tables.
– Link workflow detail indicators more tightly to current state.
– Simplify navigation between task and aggregated logs.
– Display total table entry counts.
– Add providers overview and infrastructure-oriented table sorting.

Overall Sentiment Toward Custom UI



B.5 Participant 5

1. Participant Metadata

Participant ID	05	**Age**	29
Date	18.02.2026	**Duration**	78 minutes
Prof. Background	Backend and Infrastructure Developer / MSc Computer Science Student	**Experience**	8 years

Experience Level

☐ None ☐ Beginner ☐ Intermediate ☒ Advanced

Airflow Familiarity

☐ None ☐ Beginner ☐ Intermediate ☒ Advanced

2. Study Setup

UI Order Used

☒ Stock → Custom

☐ Custom → Stock

☒ Think-Aloud Used

Tasks Completed

☒ Inspect failed tasks

☒ Find logs

☒ Trigger workflow

☒ Modify parameter

☒ Understand DAG structure

☒ Identify dependencies

3. Interview Summary

The participant strongly compared both interfaces from an infrastructure perspective and favored the custom solution throughout. Debugging support, workflow transparency, and trust in system state were recurring themes. The participant explicitly framed several improvements as reducing operational uncertainty and increasing execution confidence.

Emergent Theme Codes

☒ Learnability

☒ Visibility

☒ Debugging

☒ Trust

☒ Efficiency

4. Participant Perceptions

Navigation Ease

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Perceived Cognitive Load

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Learnability

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Preferred UI for Daily Work

☐ Stock
☒ Custom
☐ Depends

Key Benefits Mentioned

☒ Better DAG overview
☒ Faster debugging
☒ Reduced context switching
☒ Better mental model

5. Observational Notes

Behavioral Observations

Observed behavior: Very confident navigation and frequent proactive exploration beyond assigned tasks.

Interaction Style

☒ Confident
☒ Exploratory
☐ Hesitant
☐ Trial-and-error
☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues

- Several expert-oriented orchestration views were missing.
- Workflow control actions could be accessed more directly.
- Paused button malfunction reduced trust.
- Destructive actions lacked additional safeguards.
- Debugging information was somewhat fragmented.

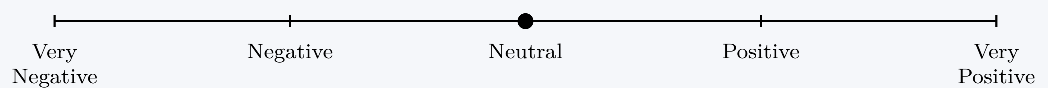
Priority of Most Relevant Issue

☐ Low ☒ Medium ☐ High

Participant Suggestions

- Add Pools and Providers view.
- Support XCom inspection.
- Add variable import support.
- Enable direct workflow execution.
- Separate singular and aggregated logs.
- Add code file view for workflows.
- Support favorite workflows.
- Add confirmation screens for destructive actions.

Overall Sentiment Toward Custom UI



B.6 Participant 6

1. Participant Metadata

Participant ID	06	**Age**	25
Date	19.02.2026	**Duration**	39 minutes
Prof. Background	MSc Computer Science Student	**Experience**	0 years

Experience Level
☒ None ☐ Beginner ☐ Intermediate ☐ Advanced

Airflow Familiarity
☒ None ☐ Beginner ☐ Intermediate ☐ Advanced

2. Study Setup

UI Order Used
☐ Stock → Custom
☒ Custom → Stock

☒ Think-Aloud Used

Tasks Completed
☒ Inspect failed tasks
☒ Find logs
☒ Trigger workflow
☒ Modify parameter
☐ Understand DAG structure
☒ Identify dependencies

3. Interview Summary

The participant completed nearly all tasks successfully, with minor difficulty identifying dependencies initially. Learnability and interface visibility were strongly emphasized, especially regarding visual cues and clearer structure compared to the baseline interface. Feedback focused more on discoverability and ease-of-use than technical debugging functionality.

Emergent Theme Codes
☒ Learnability
☒ Visibility
☐ Debugging
☐ Trust
☒ Efficiency

4. Participant Perceptions

Navigation Ease

	Low	Med	High
Stock	☒	☐	☐
Custom	☐	☒	☐

Perceived Cognitive Load

	Low	Med	High
Stock	☒	☐	☐
Custom	☐	☒	☐

Learnability

	Low	Med	High
Stock	☒	☐	☐
Custom	☐	☒	☐

Preferred UI for Daily Work

- ☐ Stock
- ☐ Custom
- ☒ Depends

Key Benefits Mentioned

- ☒ Better DAG overview
- ☐ Faster debugging
- ☐ Reduced context switching
- ☒ Better mental model

5. Observational Notes

Behavioral Observations

Hesitation points: Repeatedly reread task instructions before triggering workflow actions.

Interaction Style

- ☐ Confident
- ☐ Exploratory
- ☒ Hesitant
- ☒ Trial-and-error
- ☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues

- Several controls lacked discoverability for novice users.
- Expanded sections did not sufficiently focus attention.
- Empty start-date behavior was confusing.
- JSON fields lacked guidance.
- Description fields appeared editable although read-only.

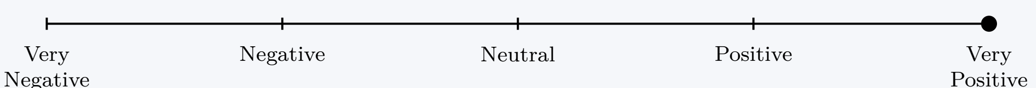
Priority of Most Relevant Issue

☒ Low ☐ Medium ☐ High

Participant Suggestions

- Use stronger indicative colors throughout the interface.
- Improve focus treatment for expanded sections.
- Make columns controls and search more discoverable.
- Provide example syntax hints for structured inputs.
- Clarify read-only descriptions visually.
- Add confirmation dialogs for risky actions.

Overall Sentiment Toward Custom UI



B.7 Participant 7

1. Participant Metadata

Participant ID	07	**Age**	28
Date	20.02.2026	**Duration**	66 minutes
Prof. Background	Backend and AI Developer / MSc Medical Engineering Student	**Experience**	2 years

Experience Level
☐ None ☐ Beginner ☒ Intermediate ☐ Advanced

Airflow Familiarity
☐ None ☐ Beginner ☐ Intermediate ☒ Advanced

2. Study Setup

UI Order Used
☒ Stock → Custom
☐ Custom → Stock

☒ Think-Aloud Used

Tasks Completed
☒ Inspect failed tasks
☒ Find logs
☒ Trigger workflow
☒ Modify parameter
☒ Understand DAG structure
☒ Identify dependencies

3. Interview Summary

The participant completed all tasks confidently and focused particularly on debugging-oriented interactions. XCom-like data visibility, logs, and dependency understanding were considered highly valuable. The participant repeatedly linked transparency to trust in the orchestration process and viewed the interface as improving efficiency for error diagnosis.

Emergent Theme Codes
☒ Learnability
☒ Visibility
☒ Debugging
☒ Trust
☒ Efficiency

4. Participant Perceptions

Navigation Ease

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Perceived Cognitive Load

Stock ☒ Low ☐ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Learnability

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Preferred UI for Daily Work

☐ Stock
☒ Custom
☐ Depends

Key Benefits Mentioned

☒ Better DAG overview
☒ Faster debugging
☒ Reduced context switching
☐ Better mental model

5. Observational Notes

Behavioral Observations

Observed behavior: Exploratory interaction pattern, particularly in graph and log views. Tested multiple navigation paths.

Interaction Style

☐ Confident
☒ Exploratory
☐ Hesitant
☒ Trial-and-error
☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues

- Advanced workflow introspection features were only partially represented.
- DAG zoom-out limits constrained graph exploration.
- Queued and running states could be distinguished more clearly.
- Debugging information was distributed across multiple views.
- Code-level workflow transparency was limited.

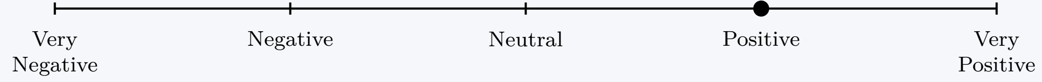
Priority of Most Relevant Issue

☐ Low ☒ Medium ☐ High

Participant Suggestions

- Add import variables support.
- Support browsing XComs.
- Add Pools and Providers view.
- Improve fullscreen and zoom behavior in DAG visualization.
- Strengthen state color differentiation.
- Add access to workflow code files.
- Improve aggregated log exploration.

Overall Sentiment Toward Custom UI



B.8 Participant 8

1. Participant Metadata

Participant ID	08	**Age**	25
Date	21.02.2026	**Duration**	47 minutes
Prof. Background	Full Stack Developer / MSc Computer Science Student	**Experience**	1 years

Experience Level

☐ None ☒ Beginner ☐ Intermediate ☐ Advanced

Airflow Familiarity

☐ None ☒ Beginner ☐ Intermediate ☐ Advanced

2. Study Setup

UI Order Used

☐ Stock → Custom

☒ Custom → Stock

☒ Think-Aloud Used

Tasks Completed

☒ Inspect failed tasks

☒ Find logs

☒ Trigger workflow

☒ Modify parameter

☒ Understand DAG structure

☒ Identify dependencies

3. Interview Summary

The participant completed all tasks successfully and reacted positively to the simplified navigation structure and data tables. Learnability and efficiency emerged most strongly, while debugging was discussed more moderately. The participant highlighted reduced friction when locating workflows and triggering runs.

Emergent Theme Codes

☒ Learnability

☐ Visibility

☐ Debugging

☒ Trust

☒ Efficiency

4. Participant Perceptions

Navigation Ease

Stock	☐ Low	☒ Med	☐ High
Custom	☐ Low	☒ Med	☐ High

Perceived Cognitive Load

Stock	☐ Low	☒ Med	☐ High
Custom	☐ Low	☐ Med	☒ High

Learnability

Stock	☐ Low	☒ Med	☐ High
Custom	☐ Low	☐ Med	☒ High

Preferred UI for Daily Work

☐ Stock
☒ Custom
☐ Depends

Key Benefits Mentioned

☒ Better DAG overview
☐ Faster debugging
☒ Reduced context switching
☒ Better mental model

5. Observational Notes

Behavioral Observations

Observed behavior: Confident task execution with occasional experimentation in table controls.

Interaction Style

☒ Confident
☒ Exploratory
☐ Hesitant
☐ Trial-and-error
☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues

- Table interactions lacked some discoverability.
- Columns control could be more prominent.
- Horizontal navigation for wide data was cumbersome.
- Workflow overview could expose recurring workflows better.
- Direct execution actions could be surfaced earlier.

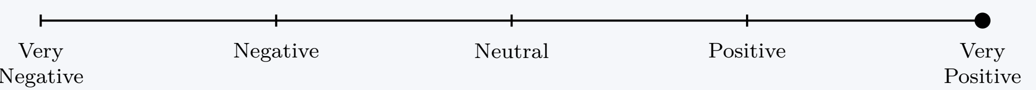
Priority of Most Relevant Issue

☐ Low ☒ Medium ☐ High

Participant Suggestions

- Add stronger sorting and search affordances.
- Show total entry counts.
- Improve horizontal scrolling behavior.
- Support favorite workflows.
- Allow direct workflow execution from overview screens.
- Add fullscreen DAG mode.

Overall Sentiment Toward Custom UI



B.9 Participant 9

1. Participant Metadata

Participant ID	09	**Age**	28
Date	22.02.2026	**Duration**	36 minutes
Prof. Background	Process Optimization Contributor / Law Student	**Experience**	1 years

Experience Level
☐ None ☒ Beginner ☐ Intermediate ☐ Advanced

Airflow Familiarity
☒ None ☐ Beginner ☐ Intermediate ☐ Advanced

2. Study Setup

UI Order Used
☒ Stock → Custom
☐ Custom → Stock

☒ Think-Aloud Used

Tasks Completed
☒ Inspect failed tasks
☒ Find logs
☒ Trigger workflow
☐ Modify parameter
☒ Understand DAG structure
☒ Identify dependencies

3. Interview Summary

The participant did not fully complete the parameter modification task but completed all other tasks successfully. Feedback focused strongly on learnability and trust, especially regarding confirmation mechanisms and clearer action feedback. The participant emphasized usability and process clarity more than debugging depth.

Emergent Theme Codes
☒ Learnability
☐ Visibility
☐ Debugging
☒ Trust
☒ Efficiency

4. Participant Perceptions

Navigation Ease
Stock ☒ Low ☐ Med ☐ High
Custom ☐ Low ☒ Med ☐ High

Perceived Cognitive Load
Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☒ Med ☐ High

Learnability
Stock ☒ Low ☐ Med ☐ High
Custom ☐ Low ☒ Med ☐ High

Preferred UI for Daily Work
☐ Stock
☐ Custom
☒ Depends

Key Benefits Mentioned
☒ Better DAG overview
☐ Faster debugging
☐ Reduced context switching
☒ Better mental model

5. Observational Notes

Behavioral Observations
Hesitation points: Some uncertainty during parameter modification. Resolved task through exploratory trial steps.

Interaction Style
☐ Confident
☐ Exploratory
☒ Hesitant
☒ Trial-and-error
☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues
- Some controls appeared ambiguous to non-technical users.
- Expanded sections lacked sufficient emphasis.
- Descriptions appeared editable when they were not.
- Search and column discoverability could improve.

Priority of Most Relevant Issue
☒ Low ☐ Medium ☐ High

Participant Suggestions
- Improve status colors and visual cues.
- Add confirmation screens for mass deletions.
- Increase emphasis on expanded sections.
- Clarify read-only descriptions visually.
- Improve discoverability of search and columns controls.
- Allow direct workflow execution with clearer affordances.

Overall Sentiment Toward Custom UI

Very Negative — Negative — Neutral — Positive — Very Positive (marked: Very Positive)

B.10 Participant 10

1. Participant Metadata

Participant ID	10	**Age**	28
Date	24.02.2026	**Duration**	74 minutes
Prof. Background	Computer Science Graduate (Graphics / Security)	**Experience**	5 years

Experience Level
☐ None ☐ Beginner ☐ Intermediate ☒ Advanced

Airflow Familiarity
☐ None ☐ Beginner ☒ Intermediate ☐ Advanced

2. Study Setup

UI Order Used
☐ Stock → Custom
☒ Custom → Stock

☒ Think-Aloud Used

Tasks Completed
☒ Inspect failed tasks
☒ Find logs
☒ Trigger workflow
☒ Modify parameter
☒ Understand DAG structure
☒ Identify dependencies

3. Interview Summary

All tasks were completed successfully. The participant particularly valued graph comprehension and workflow-state visibility, emphasizing improvements in understanding dependencies and execution progression. Debugging and efficiency also emerged as important themes, especially concerning log access.

Emergent Theme Codes
☒ Learnability
☒ Visibility
☒ Debugging
☒ Trust
☒ Efficiency

4. Participant Perceptions

Navigation Ease

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Perceived Cognitive Load

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Learnability

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☒ Med ☐ High

Preferred UI for Daily Work

☐ Stock
☒ Custom
☐ Depends

Key Benefits Mentioned

☒ Better DAG overview
☒ Faster debugging
☒ Reduced context switching
☒ Better mental model

5. Observational Notes

Behavioral Observations

Observed behavior: Fast and structured progression through tasks, often anticipating next steps.

Interaction Style

☒ Confident
☒ Exploratory
☐ Hesitant
☐ Trial-and-error
☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues

- DAG visualization navigation still had zoom constraints.
- Workflow detail icon-state linkage was unclear.
- Dark mode icon visibility needed refinement.
- Logs could be better separated by aggregation level.
- Wide tables required improved navigation.

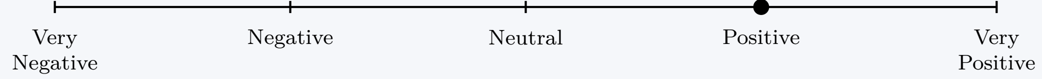
Priority of Most Relevant Issue

☐ Low ☒ Medium ☐ High

Participant Suggestions

- Add fullscreen DAG exploration.
- Improve zoom-out limits.
- Link workflow detail indicators more clearly to state.
- Improve dark mode icon visibility.
- Separate singular and aggregated logs.
- Add access to workflow code files.
- Improve horizontal scrolling.

Overall Sentiment Toward Custom UI



B.11 Participant 11

1. Participant Metadata

Participant ID	11	**Age**	38
Date	25.02.2026	**Duration**	62 minutes
Prof. Background	Architectural Draftsman	**Experience**	15 years

Experience Level

Airflow Familiarity

☐ None ☒ Beginner ☐ Intermediate ☐ Advanced

2. Study Setup

UI Order Used

☒ Stock → Custom

☐ Custom → Stock

☒ Think-Aloud Used

Tasks Completed

☒ Inspect failed tasks

☒ Find logs

☒ Trigger workflow

☒ Modify parameter

☒ Understand DAG structure

☒ Identify dependencies

3. Interview Summary

The participant completed all tasks and frequently assessed the interface from a visualization and drafting perspective. Visibility and efficiency were dominant themes, with praise for layout structure and fullscreen graph exploration. Suggestions centered more on interaction refinements than on fundamental shortcomings.

Emergent Theme Codes

☒ Learnability

☒ Visibility

☐ Debugging

☒ Trust

☒ Efficiency

4. Participant Perceptions

Navigation Ease
Stock ☒ Low ☐ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Perceived Cognitive Load
Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Learnability
Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☒ Med ☐ High

Preferred UI for Daily Work
☐ Stock
☒ Custom
☐ Depends

Key Benefits Mentioned
☒ Better DAG overview
☐ Faster debugging
☒ Reduced context switching
☒ Better mental model

5. Observational Notes

Behavioral Observations
Observed behavior: Explored zooming and graph interaction extensively before moving on.

Interaction Style
☐ Confident
☒ Exploratory
☐ Hesitant
☐ Trial-and-error
☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues
- Graph navigation could support broader exploration.
- Expanded sections could better focus attention.
- Some destructive actions lacked sufficient safeguards.
- Table usability could improve.

Priority of Most Relevant Issue
☐ Low ☒ Medium ☐ High

Participant Suggestions
- Improve color encoding and graph navigation.
- Add fullscreen graph interaction.
- Improve sortable tables and horizontal scrolling.
- Strengthen focus indication in expanded sections.
- Add confirmation prompts for risky actions.
- Support favorite workflows.

Overall Sentiment Toward Custom UI

Very Negative — Negative — Neutral — Positive — **Very Positive** (selected)

B.12 Participant 12

1. Participant Metadata

Participant ID	12	**Age**	32
Date	26.02.2026	**Duration**	80 minutes
Prof. Background	Backend Developer	**Experience**	7 years

Experience Level
☐ None ☐ Beginner ☐ Intermediate ☒ Advanced

Airflow Familiarity
☐ None ☐ Beginner ☐ Intermediate ☒ Advanced

2. Study Setup

UI Order Used
☐ Stock → Custom
☒ Custom → Stock

☒ Think-Aloud Used

Tasks Completed
☒ Inspect failed tasks
☒ Find logs
☒ Trigger workflow
☒ Modify parameter
☒ Understand DAG structure
☒ Identify dependencies

3. Interview Summary

The participant completed all tasks successfully and concentrated strongly on advanced operational concerns. Debugging, trust, and efficiency emerged prominently, particularly around logs, providers, and workflow controls. The participant characterized the custom interface as substantially better suited for production-oriented monitoring.

Emergent Theme Codes
☒ Learnability
☒ Visibility
☒ Debugging
☒ Trust
☒ Efficiency

4. Participant Perceptions

Navigation Ease

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☒ Med ☐ High

Perceived Cognitive Load

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Learnability

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Preferred UI for Daily Work

☐ Stock
☒ Custom
☐ Depends

Key Benefits Mentioned

☒ Better DAG overview
☒ Faster debugging
☒ Reduced context switching
☒ Better mental model

5. Observational Notes

Behavioral Observations

Observed behavior: Very confident interactions, particularly in debugging-oriented tasks.

Interaction Style

☒ Confident
☒ Exploratory
☐ Hesitant
☐ Trial-and-error
☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues

- Several expert debugging capabilities were missing.
- Paused-button issue reduced confidence.
- Empty start-date handling was unclear.
- Direct operational controls could be expanded.

Priority of Most Relevant Issue

☐ Low ☒ Medium ☐ High

Participant Suggestions

- Add Pools and Providers view.
- Support XCom inspection.
- Add variable import support.
- Provide code file viewing.
- Support aggregated log access.
- Enable direct workflow execution.
- Support favorite workflows.
- Improve empty start-date handling.

Overall Sentiment Toward Custom UI

Very Negative — Negative — **● Neutral** — Positive — Very Positive

B.13 Participant 13

1. Participant Metadata

Participant ID	13	**Age**	25
Date	27.02.2026	**Duration**	55 minutes
Prof. Background	Full Stack Developer / MSc Computer Science Student	**Experience**	3 years

Experience Level

☐ None ☐ Beginner ☒ Intermediate ☐ Advanced

Airflow Familiarity

☐ None ☐ Beginner ☒ Intermediate ☐ Advanced

2. Study Setup

UI Order Used

☒ Stock → Custom

☐ Custom → Stock

☒ Think-Aloud Used

Tasks Completed

☒ Inspect failed tasks

☒ Find logs

☒ Trigger workflow

☒ Modify parameter

☒ Understand DAG structure

☒ Identify dependencies

3. Interview Summary

The participant completed all tasks successfully and emphasized reduced context switching as a major benefit. Learnability and efficiency emerged strongly, while debugging was discussed moderately. Several remarks focused on improving discoverability through stronger visual signaling.

Emergent Theme Codes

☒ Learnability

☒ Visibility

☐ Debugging

☒ Trust

☒ Efficiency

4. Participant Perceptions

Navigation Ease

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Perceived Cognitive Load

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☒ Med ☐ High

Learnability

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Preferred UI for Daily Work

☐ Stock
☒ Custom
☐ Depends

Key Benefits Mentioned

☒ Better DAG overview
☐ Faster debugging
☒ Reduced context switching
☒ Better mental model

5. Observational Notes

Behavioral Observations

Observed behavior: Minor experimentation around search and sorting features, otherwise direct task completion.

Interaction Style

☐ Confident
☒ Exploratory
☐ Hesitant
☒ Trial-and-error
☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues

- Table discoverability could be improved.
- Colors could more strongly differentiate states.
- Columns controls were too subtle.
- Destructive actions lacked explicit confirmation.

Priority of Most Relevant Issue

☐ Low ☒ Medium ☐ High

Participant Suggestions

- Improve status color semantics.
- Add stronger search and sorting support.
- Show total entry counts.
- Make columns controls more visible.
- Add direct workflow execution.
- Add confirmation dialogs for mass deletions.

Overall Sentiment Toward Custom UI

Very Negative — Negative — Neutral — Positive — Very Positive (●)

B.14 Participant 14

1. Participant Metadata

Participant ID	14	**Age**	28
Date	28.02.2026	**Duration**	49 minutes
Prof. Background	Computer Science Graduate (IT Security)	**Experience**	2 years

Experience Level
☐ None ☐ Beginner ☒ Intermediate ☐ Advanced

Airflow Familiarity
☐ None ☐ Beginner ☒ Intermediate ☐ Advanced

2. Study Setup

UI Order Used
☐ Stock → Custom
☒ Custom → Stock

☒ Think-Aloud Used

Tasks Completed
☒ Inspect failed tasks
☒ Find logs
☒ Trigger workflow
☒ Modify parameter
☒ Understand DAG structure
☒ Identify dependencies

3. Interview Summary

The participant completed all tasks and highlighted debugging support and workflow transparency as major strengths. Trust in system state representation emerged repeatedly, especially compared with the baseline interface. Several suggestions targeted technical refinements rather than conceptual weaknesses.

Emergent Theme Codes
☒ Learnability
☒ Visibility
☒ Debugging
☒ Trust
☒ Efficiency

4. Participant Perceptions

Navigation Ease

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☒ Med ☐ High

Perceived Cognitive Load

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Learnability

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☒ Med ☐ High

Preferred UI for Daily Work

☐ Stock
☒ Custom
☐ Depends

Key Benefits Mentioned

☒ Better DAG overview
☒ Faster debugging
☒ Reduced context switching
☐ Better mental model

5. Observational Notes

Behavioral Observations

Observed behavior: Strongly exploratory interaction style, frequently inspecting additional details beyond task scope.

Interaction Style

☐ Confident
☒ Exploratory
☐ Hesitant
☐ Trial-and-error
☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues

- Some advanced operational information was missing.
- Paused-button issue affected trust.
- State indication in workflow details could improve.
- Table navigation was cumbersome in wider views.

Priority of Most Relevant Issue

☐ Low ☒ Medium ☐ High

Participant Suggestions

- Add code file access.
- Separate singular and aggregated logs.
- Add providers overview and variable import.
- Improve state color differentiation.
- Resolve paused-button inconsistency.
- Improve horizontal table scrolling.

Overall Sentiment Toward Custom UI

Very Negative — Negative — Neutral — **● Positive** — Very Positive

B.15 Participant 15

1. Participant Metadata

Participant ID	15	**Age**	25
Date	01.03.2026	**Duration**	41 minutes
Prof. Background	Architecture and Civil Engineering Student	**Experience**	2 years

Experience Level

☐ None ☐ Beginner ☒ Intermediate ☐ Advanced

Airflow Familiarity

☒ None ☐ Beginner ☐ Intermediate ☐ Advanced

2. Study Setup

UI Order Used

☒ Stock → Custom
☐ Custom → Stock

☒ Think-Aloud Used

Tasks Completed

☒ Inspect failed tasks
☒ Find logs
☒ Trigger workflow
☒ Modify parameter
☐ Understand DAG structure
☒ Identify dependencies

3. Interview Summary

The participant completed nearly all tasks successfully, with minor difficulty identifying dependencies. Learnability and visibility were dominant themes, and the participant favored the graphical workflow representation for reducing complexity. Efficiency benefits were acknowledged, while debugging emerged less strongly.

Emergent Theme Codes

☒ Learnability
☒ Visibility
☐ Debugging
☐ Trust
☒ Efficiency

4. Participant Perceptions

Navigation Ease
Stock ☒ Low ☐ Med ☐ High
Custom ☐ Low ☒ Med ☐ High

Perceived Cognitive Load
Stock ☒ Low ☐ Med ☐ High
Custom ☐ Low ☒ Med ☐ High

Learnability
Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☒ Med ☐ High

Preferred UI for Daily Work
☐ Stock
☐ Custom
☒ Depends

Key Benefits Mentioned
☒ Better DAG overview
☐ Faster debugging
☐ Reduced context switching
☒ Better mental model

5. Observational Notes

Behavioral Observations
Hesitation points: Paused before dependency interpretation and required some trial-and-error exploration.

Interaction Style
☐ Confident
☐ Exploratory
☒ Hesitant
☒ Trial-and-error
☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues
- DAG exploration could be more flexible.
- Search and sorting interactions could be stronger.
- Expanded section focus was limited.
- Some actions lacked additional safeguards.

Priority of Most Relevant Issue
☐ Low ☒ Medium ☐ High

Participant Suggestions
- Improve visual color guidance.
- Extend zooming and fullscreen graph interaction.
- Improve search and sorting behavior.
- Display total entry counts.
- Support favorite workflows.
- Add confirmation screens for risky actions.

Overall Sentiment Toward Custom UI

Very Negative — Negative — Neutral — Positive — Very Positive (●)

B.16 Participant 16

1. Participant Metadata

Participant ID	16	**Age**	34
Date	03.03.2026	**Duration**	76 minutes
Prof. Background	Backend Developer (Simulation) / MSc Computer Science Student	**Experience**	7 years

Experience Level
☐ None ☐ Beginner ☐ Intermediate ☒ Advanced

Airflow Familiarity
☐ None ☐ Beginner ☐ Intermediate ☒ Advanced

2. Study Setup

UI Order Used
☐ Stock → Custom
☒ Custom → Stock

☒ Think-Aloud Used

Tasks Completed
☒ Inspect failed tasks
☒ Find logs
☒ Trigger workflow
☒ Modify parameter
☒ Understand DAG structure
☒ Identify dependencies

3. Interview Summary

The participant completed all tasks successfully and strongly focused on advanced debugging and operational support. The discussion repeatedly linked richer workflow introspection to increased trust and execution efficiency. Overall assessment was highly positive with suggestions centered on extending expert-oriented functionality.

Emergent Theme Codes
☒ Learnability
☒ Visibility
☒ Debugging
☒ Trust
☒ Efficiency

4. Participant Perceptions

Navigation Ease

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Perceived Cognitive Load

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Learnability

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Preferred UI for Daily Work

☐ Stock
☒ Custom
☐ Depends

Key Benefits Mentioned

☒ Better DAG overview
☒ Faster debugging
☒ Reduced context switching
☒ Better mental model

5. Observational Notes

Behavioral Observations

Observed behavior: Confident and efficient execution, often verbalizing expected system behavior.

Interaction Style

☒ Confident
☒ Exploratory
☐ Hesitant
☐ Trial-and-error
☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues

- Advanced debugging-oriented information surfaces could be richer.
- JSON input lacked syntax guidance.
- Empty start-date behavior was unclear.
- Several Airflow expert views were absent.

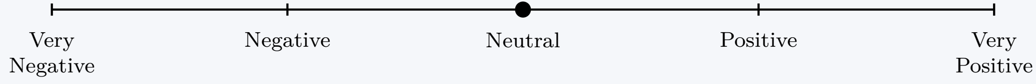
Priority of Most Relevant Issue

☐ Low ☒ Medium ☐ High

Participant Suggestions

- Add variable import.
- Provide JSON syntax examples.
- Support XCom inspection.
- Add Pools and Providers view.
- Add workflow code file access.
- Improve aggregated log support.
- Support direct workflow execution.
- Improve debugging-oriented information surfaces.

Overall Sentiment Toward Custom UI



B.17 Participant 17

1. Participant Metadata

Participant ID	17	**Age**	25
Date	04.03.2026	**Duration**	58 minutes
Prof. Background	Computer Science Graduate (Graphics / Image Processing)	**Experience**	2 years

Experience Level
☐ None ☐ Beginner ☒ Intermediate ☐ Advanced

Airflow Familiarity
☐ None ☒ Beginner ☐ Intermediate ☐ Advanced

2. Study Setup

UI Order Used
☒ Stock → Custom
☐ Custom → Stock

☒ Think-Aloud Used

Tasks Completed
☒ Inspect failed tasks
☒ Find logs
☒ Trigger workflow
☒ Modify parameter
☒ Understand DAG structure
☒ Identify dependencies

3. Interview Summary

All tasks were completed successfully. The participant reacted particularly positively to the graph representation and task-state visualization, emphasizing learnability and visibility gains. Some debugging support was appreciated, though efficiency improvements were discussed more strongly.

Emergent Theme Codes
☒ Learnability
☒ Visibility
☒ Debugging
☐ Trust
☒ Efficiency

4. Participant Perceptions

Navigation Ease

Stock ☒ Low ☐ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Perceived Cognitive Load

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Learnability

Stock ☐ Low ☒ Med ☐ High
Custom ☐ Low ☐ Med ☒ High

Preferred UI for Daily Work

☐ Stock
☒ Custom
☐ Depends

Key Benefits Mentioned

☒ Better DAG overview
☒ Faster debugging
☒ Reduced context switching
☒ Better mental model

5. Observational Notes

Behavioral Observations

Observed behavior: Exploratory use of graph features and alternate task navigation paths.

Interaction Style

☐ Confident
☒ Exploratory
☐ Hesitant
☐ Trial-and-error
☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues

- Graph navigation still had zoom constraints.
- Workflow detail state indication could improve.
- Dark mode icon contrast required refinement.
- Expanded sections lacked some visual focus.

Priority of Most Relevant Issue

☐ Low ☒ Medium ☐ High

Participant Suggestions

- Improve fullscreen DAG visualization.
- Extend zoom-out limits.
- Improve workflow detail icon-state linkage.
- Improve dark mode icon visibility.
- Add stronger focus cues for expanded sections.
- Improve sortable tables and horizontal scrolling.

Overall Sentiment Toward Custom UI

Very Negative — Negative — Neutral — Positive — ● Very Positive

B.18 Participant 18

1. Participant Metadata

Participant ID	18	**Age**	25
Date	05.03.2026	**Duration**	35 minutes
Prof. Background	Teacher in Training	**Experience**	0 years

Experience Level
☒ None ☐ Beginner ☐ Intermediate ☐ Advanced

Airflow Familiarity
☒ None ☐ Beginner ☐ Intermediate ☐ Advanced

2. Study Setup

UI Order Used
☐ Stock → Custom
☒ Custom → Stock

☒ Think-Aloud Used

Tasks Completed
☒ Inspect failed tasks
☒ Find logs
☒ Trigger workflow
☒ Modify parameter
☒ Understand DAG structure
☒ Identify dependencies

3. Interview Summary

The participant completed all tasks successfully and emphasized ease of understanding over technical aspects. Learnability, trust, and efficiency emerged clearly, particularly concerning safer interactions and reduced confusion. Feedback largely focused on accessibility and confidence-building aspects of the interface.

Emergent Theme Codes
☒ Learnability
☐ Visibility
☐ Debugging
☒ Trust
☒ Efficiency

4. Participant Perceptions

Navigation Ease

Stock	☒ Low	☐ Med	☐ High
Custom	☐ Low	☒ Med	☐ High

Perceived Cognitive Load

Stock	☐ Low	☒ Med	☐ High
Custom	☐ Low	☒ Med	☐ High

Learnability

Stock	☐ Low	☒ Med	☐ High
Custom	☐ Low	☐ Med	☒ High

Preferred UI for Daily Work

- ☐ Stock
- ☒ Custom
- ☐ Depends

Key Benefits Mentioned

- ☒ Better DAG overview
- ☐ Faster debugging
- ☐ Reduced context switching
- ☒ Better mental model

5. Observational Notes

Behavioral Observations

Hesitation points: Initially cautious interaction, but confidence increased noticeably after first tasks.

Interaction Style

- ☐ Confident
- ☐ Exploratory
- ☒ Hesitant
- ☐ Trial-and-error
- ☐ Frustrated

6. Pain Points & Improvement Opportunities

Reported Issues

- Some controls were initially hard to interpret.
- Read-only descriptions appeared editable.
- Search discoverability could improve.
- Risky actions lacked explicit confirmation.

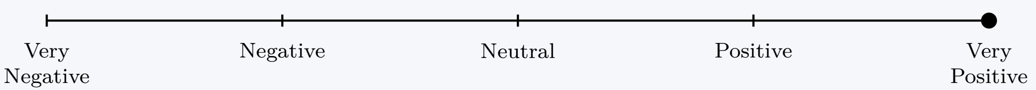
Priority of Most Relevant Issue

☒ Low ☐ Medium ☐ High

Participant Suggestions

- Improve indicative color usage.
- Clarify read-only descriptions visually.
- Add stronger focus cues for expanded sections.
- Provide example syntax hints.
- Improve search discoverability.
- Add confirmation screens for destructive actions.
- Support favorite workflows.

Overall Sentiment Toward Custom UI



References

[1] N. S. Miriyala, ‘Study of workflow orchestration engines: Open-source & cloud-native solutions,’ *Stochastic Modelling and Computational Sciences*, vol. 5, no. 1, pp. 1–16, Mar. 2025, Published in 2025; full text accessed via ResearchGate. Accessed: 14th Apr. 2026. [Online]. Available: https://www.researchgate.net/publication/390988200_STUDY_OF_WORKFLOW_ORCHESTRATION_ENGINES_OPEN-SOURCE_CLOUD-NATIVE_SOLUTIONS.

[2] Apache Software Foundation. ‘Architecture overview — apache airflow documentation,’ Apache Software Foundation, Accessed: 14th Apr. 2026. [Online]. Available: https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/overview.html.

[3] Apache Software Foundation. ‘Dags — apache airflow documentation,’ Apache Software Foundation, Accessed: 14th Apr. 2026. [Online]. Available: https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dags.html.

[4] Prefect Technologies, Inc. ‘Flows — prefect documentation,’ Prefect, Accessed: 14th Apr. 2026. [Online]. Available: https://docs.prefect.io/v3/concepts/flows.

[5] Dagster Labs. ‘Overview — dagster docs,’ Dagster Labs, Accessed: 14th Apr. 2026. [Online]. Available: https://docs.dagster.io/.

[6] Luigi Contributors. ‘Tasks — luigi documentation,’ Luigi, Accessed: 14th Apr. 2026. [Online]. Available: https://luigi.readthedocs.io/en/latest/tasks.html.

[7] J. Guerrero-García, ‘Evolutionary design of user interfaces for workflow information systems,’ *Science of Computer Programming*, vol. 86, pp. 89–102, 2014. DOI: 10.1016/j.scico.2013.07.003.

[8] S. Yongchareon, C. Liu, X. Zhao, J. Yu, K. Ngamakeur and J. Xu, ‘Deriving user interface flow models for artifact-centric business processes,’ *Computers in Industry*, vol. 96, pp. 66–85, 2018. DOI: 10.1016/j.compind.2017.11.001.

[9] R. Rabiei and S. Almasi, ‘Requirements and challenges of hospital dashboards: A systematic literature review,’ *BMC Medical Informatics and Decision Making*, vol. 22, no. 287, 2022. DOI: 10.1186/s12911-022-02037-8.

[10] J. Siette et al., 'Usability and acceptability of clinical dashboards in aged care: Systematic review,' *JMIR Aging*, vol. 6, e42274, 2023. DOI: 10.2196/42274.

[11] J. Ke, P. Liao, J. Li and X. Luo, 'Effect of information load and cognitive style on cognitive load of visualized dashboards for construction-related activities,' *Automation in Construction*, vol. 154, p. 105 029, 2023. DOI: 10.1016/j.autcon.2023.105029.

[12] M. Smereka, G. Kołaczek, J. Sobecki and A. Wasilewski, 'Adaptive user interface for workflow-ERP system,' *Procedia Computer Science*, vol. 225, pp. 2381–2391, 2023. DOI: 10.1016/j.procs.2023.10.229.

[13] P. A. Akiki, 'CHAIN: Developing model-driven contextual help for adaptive user interfaces,' *Journal of Systems and Software*, vol. 135, pp. 165–190, 2018. DOI: 10.1016/j.jss.2017.10.017.

[14] L. H. Lee, K. Y. Lam and P. Hui, 'Exploring user engagement by diagnosing visual guides in onboarding screens with linear regression and XGBoost,' *Displays*, vol. 87, p. 102 975, 2025. DOI: 10.1016/j.displa.2025.102975.

[15] J. D. Still, 'Web page visual hierarchy: Examining faraday's guidelines for entry points,' *Computers in Human Behavior*, vol. 84, pp. 352–359, 2018. DOI: 10.1016/j.chb.2018.03.014.

[16] E. Andersen and A. Maier, 'The attentional guidance of individual colours in increasingly complex displays,' *Applied Ergonomics*, vol. 81, p. 102 885, 2019. DOI: 10.1016/j.apergo.2019.102885.

[17] C. Ma, H. Wang, J. Wu and C. Xue, 'Applying gestalt similarity to improve visual perception of interface color quantity: An EEG study,' *International Journal of Industrial Ergonomics*, vol. 100, p. 103 521, 2024. DOI: 10.1016/j.ergon.2023.103521.

[18] T. Jin, W. Wang, J. He, Z. Wu and H. Gu, 'Influence mechanism of icon semantics on visual search performance: Evidence from an eye-tracking study,' *International Journal of Industrial Ergonomics*, vol. 93, p. 103 402, 2023. DOI: 10.1016/j.ergon.2022.103402.

[19] D. Dowding, J. A. Merrill, Y. Barrón, N. Onorato, K. Jonas and D. Russell, 'Usability evaluation of a dashboard for home care nurses,' *Computers, Informatics, Nursing*, vol. 37, no. 1, pp. 11–19, 2019. DOI: 10.1097/CIN.0000000000000484.

[20] S. Almasi, K. Bahaadinbeigy, H. Ahmadi, S. Sohrabei and R. Rabiei, 'Usability evaluation of dashboards: A systematic literature review of tools,' *BioMed Research International*, vol. 2023, p. 9 990 933, 2023. DOI: 10.1155/2023/9990933.

[21] F. S. Rossi, M. C. B. Adams, G. A. Aarons, M. P. McGovern et al., 'From glitter to gold: Recommendations for effective dashboards from design through sustainment,' *Implementation Science*, vol. 20, no. 16, 2025. DOI: 10.1186/s13012-025-01430-x.

[22] S. de Lusignan et al., 'Atrial fibrillation dashboard evaluation using the think aloud protocol,' *BMJ Health & Care Informatics*, vol. 27, no. 3, e100191, 2020. DOI: 10.1136/bmjhci-2020-100191.

[23] European Union. 'Accessibility of products and services,' EUR-Lex, Accessed: 14th Apr. 2026. [Online]. Available: https://eur-lex.europa.eu/EN/legal-content/summary/accessibility-of-products-and-services.html.

[24] European Union. 'Directive (eu) 2019/882 of the european parliament and of the council of 17 april 2019 on the accessibility requirements for products and services,' EUR-Lex, Accessed: 14th Apr. 2026. [Online]. Available: https://eur-lex.europa.eu/eli/dir/2019/882/oj.

[25] Bundesministerium der Justiz and Bundesamt für Justiz. 'Barrierefreiheitsstärkungsgesetz (bfsg),' Gesetze im Internet, Accessed: 14th Apr. 2026. [Online]. Available: https://www.gesetze-im-internet.de/bfsg/BJNR297010021.html.

[26] Bundesministerium der Justiz and Bundesamt für Justiz. 'Verordnung zum barrierefreiheitsstärkungsgesetz (bfsgv),' Gesetze im Internet, Accessed: 14th Apr. 2026. [Online]. Available: https://www.gesetze-im-internet.de/bfsgv/BJNR092800022.html.

[27] World Wide Web Consortium. 'Wcag 2 overview,' W3C Web Accessibility Initiative, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/standards-guidelines/wcag/.

[28] World Wide Web Consortium. 'Web content accessibility guidelines (wcag) 2.2 is a w3c recommendation,' W3C, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/news/2023/web-content-accessibility-guidelines-wcag-2-2-is-a-w3c-recommendation/.

[29] World Wide Web Consortium. 'Web content accessibility guidelines (wcag) 2.2 approved as iso/iec international standard,' W3C, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/news/2025/web-content-accessibility-guidelines-wcag-2-2-approved-as-iso-iec-international-standard/.

[30] World Wide Web Consortium. 'Introduction to understanding wcag 2.2,' W3C Web Accessibility Initiative, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/WCAG22/Understanding/intro.

[31] World Wide Web Consortium. 'Web content accessibility guidelines (wcag) 2.2,' W3C, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/TR/wcag/.

[32] World Wide Web Consortium. 'Web content accessibility guidelines (wcag) 2 level aa conformance,' W3C Web Accessibility Initiative, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/WCAG2AA-Conformance.

[33] World Wide Web Consortium. 'Understanding guideline 2.1: Keyboard accessible,' W3C Web Accessibility Initiative, Accessed: 14th Apr. 2026. [On-

line]. Available: https://www.w3.org/WAI/WCAG21/Understanding/keyboard-accessible.
[34] World Wide Web Consortium. 'Using aria,' W3C, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/TR/using-aria/.
[35] World Wide Web Consortium. 'Understanding success criterion 2.4.3: Focus order,' W3C Web Accessibility Initiative, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/WCAG22/Understanding/focus-order.html.
[36] World Wide Web Consortium. 'Understanding success criterion 2.4.7: Focus visible,' W3C Web Accessibility Initiative, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/WCAG22/Understanding/focus-visible.
[37] World Wide Web Consortium. 'Dialog (modal) pattern,' W3C ARIA Authoring Practices Guide, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/ARIA/apg/patterns/dialog-modal/.
[38] Government Digital Service. 'Button,' GOV.UK Design System, Accessed: 14th Apr. 2026. [Online]. Available: https://design-system.service.gov.uk/components/button/.
[39] World Wide Web Consortium. 'Read me first,' W3C ARIA Authoring Practices Guide, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/ARIA/apg/practices/read-me-first/.
[40] World Wide Web Consortium. 'Labeling controls,' W3C Web Accessibility Initiative, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/tutorials/forms/labels/.
[41] World Wide Web Consortium. 'Form instructions,' W3C Web Accessibility Initiative, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/tutorials/forms/instructions/.
[42] World Wide Web Consortium. 'Caption & summary,' W3C Web Accessibility Initiative, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/tutorials/tables/caption-summary/.
[43] World Wide Web Consortium. 'Table pattern,' W3C ARIA Authoring Practices Guide, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/ARIA/apg/patterns/table/.
[44] World Wide Web Consortium. 'Grid (interactive tabular data and layout containers) pattern,' W3C ARIA Authoring Practices Guide, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/ARIA/apg/patterns/grid/.
[45] World Wide Web Consortium. 'Understanding success criterion 1.4.1: Use of color,' W3C Web Accessibility Initiative, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/WCAG22/Understanding/use-of-color.

[46] Government Digital Service. 'Navigate a service,' GOV.UK Design System, Accessed: 14th Apr. 2026. [Online]. Available: https://design-system.service.gov.uk/patterns/navigate-a-service/.

[47] Government Digital Service. 'Pagination,' GOV.UK Design System, Accessed: 14th Apr. 2026. [Online]. Available: https://design-system.service.gov.uk/components/pagination/.

[48] World Wide Web Consortium. 'Tabs pattern,' W3C ARIA Authoring Practices Guide, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/ARIA/apg/patterns/tabs/.

[49] Government Digital Service. 'Tabs,' GOV.UK Design System, Accessed: 14th Apr. 2026. [Online]. Available: https://design-system.service.gov.uk/components/tabs/.

[50] World Wide Web Consortium. 'Breadcrumb pattern,' W3C ARIA Authoring Practices Guide, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/ARIA/apg/patterns/breadcrumb/.

[51] World Wide Web Consortium. 'Understanding success criterion 4.1.3: Status messages,' W3C Web Accessibility Initiative, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/WCAG21/Understanding/status-messages.html.

[52] World Wide Web Consortium. 'Technique aria22: Using role=status to present status messages,' W3C Web Accessibility Initiative, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/WCAG22/Techniques/aria/ARIA22.html.

[53] World Wide Web Consortium. 'Technique aria25: Using an aria live region to convey the status of a progress bar,' W3C Web Accessibility Initiative, Accessed: 14th Apr. 2026. [Online]. Available: https://www.w3.org/WAI/WCAG22/Techniques/aria/ARIA25.

[54] Meta Platforms, Inc. 'Sharing state between components,' React Documentation, Accessed: 14th Apr. 2026. [Online]. Available: https://react.dev/learn/sharing-state-between-components.

[55] Meta Platforms, Inc. 'Choosing the state structure,' React Documentation, Accessed: 14th Apr. 2026. [Online]. Available: https://react.dev/learn/choosing-the-state-structure.

[56] Microsoft. 'Tsconfig option: Strict,' TypeScript Documentation, Accessed: 14th Apr. 2026. [Online]. Available: https://www.typescriptlang.org/tsconfig/strict.html.

[57] Microsoft. 'Typescript handbook: Narrowing,' TypeScript Documentation, Accessed: 14th Apr. 2026. [Online]. Available: https://www.typescriptlang.org/docs/handbook/2/narrowing.html.

[58] TanStack. 'Important defaults,' TanStack Query Documentation, Accessed: 14th Apr. 2026. [Online]. Available: https://tanstack.com/query/latest/docs/framework/react/guides/important-defaults.

[59] TanStack. 'Query invalidation,' TanStack Query Documentation, Accessed: 14th Apr. 2026. [Online]. Available: https://tanstack.com/query/latest/docs/framework/react/guides/query-invalidation.

[60] xyflow. 'Accessibility — react flow,' React Flow Documentation, Accessed: 14th Apr. 2026. [Online]. Available: https://reactflow.dev/learn/advanced-use/accessibility.

[61] xyflow. 'Performance — react flow,' React Flow Documentation, Accessed: 14th Apr. 2026. [Online]. Available: https://reactflow.dev/learn/advanced-use/performance.

[62] Apache Software Foundation. 'Airflow public api — apache airflow documentation,' Apache Software Foundation, Accessed: 14th Apr. 2026. [Online]. Available: https://airflow.apache.org/docs/apache-airflow/stable/stable-rest-api-ref.html.