Zurich University
of Applied Sciences

School of
Engineering
InIT Institute of Applied
Information Technology

MSE Master's Thesis

Data-Centric Distributed Scheduling of ETL Jobs

Author Julie Ann George

Date 31.01.2024

Zurich University
of Applied Sciences

School of
Engineering
InIT Institute of Applied
Information Technology

DECLARATION OF ORIGINALITY

Master's Thesis at the School of Engineering

By submitting this Master's thesis, the undersigned student confirms that this thesis is his/her own work and was written without the help of a third party. The student declares that all sources in the text (including Internet pages) and appendices have been correctly disclosed. This means that there has been no plagiarism, i.e. no sections of the Master's thesis have been partially or wholly taken from other texts and represented as the student's own work or included without being correctly referenced. Any misconduct will be dealt with according to paragraphs 39 and 40 of the General Academic Regulations for Bachelor's and Master's Degree courses at the Zurich University of Applied Sciences (Rahmenprüfungsordnung ZHAW (RPO)) and subject to the provisions for disciplinary action stipulated in the University regulations.

City/Date: Signature:

Neftenbach, 29.12.2023

---------- ----------

The original signed and dated document (no copies) must be included in the appendix of the ZHAW version of all Master's theses submitted.

Abstract

In today's data-driven world, especially within the domain of data engineering, the necessity for refined operational methodologies is paramount. These methodologies must not only address computational requirements but also consider the potential consequences stemming from the presence or absence of data knowledge.

Efficient data integration is of utmost importance in the landscape of data engineering and analytics. Traditionally, the ETL (Extract, Transform, Load) model has served as the cornerstone for integrating data from various sources, transforming it, and preparing it for analytical applications. However, conventional scheduling practices are often command-centric or task-centric and lack a deep understanding of the intrinsic attributes of the data and their consequential impact on efficiency and resource allocation.

A more dynamic approach involves the adoption of data-centric scheduling, rooted in the principle of deriving insights directly from the data itself. This approach emphasizes a thoughtful consideration of the data's inherent characteristics, drawing insights from source data, ongoing queries, and even manually annotated metadata.

The objective of this research is to explore the potential of data-centric scheduling and develop a more effective system for managing data processing jobs, particularly ETL tasks, through the implementation of data-centric scheduling. Such an approach holds the promise of improving job performance and sustainability.

To gauge the current state of scheduling solutions and aid in selecting a suitable candidate for developing the prototype, we will conduct a comparative study of both licensed and open-source schedulers. Our research methodology involves creating a controlled test environment with representative ETL jobs subjected to various failure conditions. The chosen ETL scheduler will be evaluated for compliance with requirements and enhanced with data-centric features.

The culmination of this research will be the extension of a prototype system, followed by an evaluation with use cases that highlight the advantages of data-centric scheduling. The overarching goal of this thesis is to provide a roadmap toward more efficient practices in ETL operations, contributing to the ongoing discourse in sustainable data engineering.

Table of Contents

1. Introduction

ETL (extract, transform, load) involves the process of extracting data from relevant sources, modifying it to meet specific requirements, and then transferring it to a statistical or analytical tool for further examination. Scheduling, on the other hand, entails setting up timed events to occur at specific moments. For instance, this could include a routine refresh of a dashboard on a daily basis, which loads newly acquired data.

ETL scheduling refers to the automation and scheduling of the tasks required for extracting data from diverse sources, transforming it into a format suitable for analysis, and loading it into a data storage system. For companies to ensure that they are making the most effective and efficient use of data, they should schedule and automate their ETL processes (11)

An ETL job scheduler is vital to enhancing productivity and reducing errors for organizations and individuals by streamlining workflows and automating repetitive tasks. Some of the primary functions performed by a job scheduler include the following:

- Job Definition: With the help of a job scheduler, users can specify the tasks that need to be performed, along with their respective inputs and outputs.

- Job Scheduling: Scheduling enables users to determine the timing of job execution. The specifics of scheduling may vary based on the type of job. Certain jobs may be scheduled to respond to specific triggers or events, while others may be set to run at predetermined times.

- Job Execution: Once a job is defined and scheduled, the job scheduler ensures its successful execution. Additionally, the job scheduler allows users to establish conditions or alerts to handle any anomalies during job execution. This may involve sending notifications to relevant stakeholders or adjusting permissions accordingly. (11)

ETL performance is crucial for multiple reasons. It affects the timeliness and accuracy of information available for decision-making in data warehouses and data marts. Poor performance can lead to data quality issues, resulting in incorrect or outdated data. Slow performance can disrupt organizational productivity, causing workflow disruptions and task delays. (2)

1.1 Problem Statement

Current job scheduling systems, responsible for the regular data updates crucial to business practices, are predominantly command or task-centric. These systems do not account for the

nature of data, and while advancements from cron jobs to more sophisticated scheduling middleware have occurred, limitations persist. These systems are still frequently plagued by issues surrounding stability, outage recovery, and reliable distributed operation (26).

The advent of data-centric ETL scheduling promises a solution. By incorporating data properties, this approach to scheduling holds the potential for more efficient resource allocation and usage (27). As an illustration, utilizing metadata or insights derived from the data, we can adapt the scheduling process by selectively excluding specific scheduled executions. For instance, if there have been no relevant updates to the database for a given timespan, we can omit the scheduled task run, effectively conserving computational resources.

However, despite its promise, data-centric scheduling is still an under-explored area. The question remains whether an existing system can be optimized to incorporate data-centric scheduling or if a novel, custom-developed scheduler is necessary for effective implementation.

This thesis seeks to address the existing gap in research by investigating the development and evaluation of a data-centric distributed scheduling model for ETL jobs. The aim is to enhance the efficiency of ETL processes, consequently contributing towards a more sustainable data engineering practice.

1.2 Objective

The central objective of this Master's Thesis is to undertake an investigation into the development and evaluation of a data-centric distributed scheduling model for ETL jobs, with a view towards overcoming the existing limitations prevalent in traditional job schedulers. This objective is drawn from the burgeoning need within data engineering to optimize the integration process, with current job scheduling tools demonstrating a lack of data awareness in their operations, thus limiting their efficacy (27).

Initial steps towards achieving this objective will involve undertaking a comprehensive comparative analysis of several open-source and licensed schedulers leading to the creation of a shortlist of the top-performing systems. Simultaneously, there will be a critical analysis of the benefits and drawbacks associated with utilizing an existing system versus the potential rewards of a custom-developed, highly-targeted scheduler. This discussion will help determine whether an innovative, built-for-purpose scheduler may hold the key to enhanced reliability and efficiency.

Once we have identified the top contenders, we will test them on a controlled testbed environment created and populated with representative ETL jobs subjected to various failure

scenarios. The value of this approach is to provide an authentic replication of the complex, dynamic, and, at times, volatile environments in which these job schedulers operate.

Finally, in collaboration with industry partner Datalizard, a prototype system will be extended or developed that should optimally meet the defined functional and non-functional requirements set by the industry partner. A thorough validation and evaluation of this system will follow, with the aim of establishing its benefits, particularly in less than ideal operating conditions. The study will contribute to the limited yet growing body of literature on data-centric scheduling in ETL job contexts, thus providing valuable insights for academia and industry.

1.3 Outline

This paper aims to address the problem of conventional ETL scheduling by proposing a new system that supports data-centric scheduling. In the first section, we introduce ETL scheduling and provide a historical background of scheduling in general. Then we review the state of existing solutions for ETL scheduling available in the market followed by a short literature analysis of research papers focused on ETL scheduling and explore current trends through relevant technical articles.

In the section on "Comparative Analysis of Existing Solutions", we outline a methodology for conducting a comparative study of the systems and proceed with an analysis of the existing systems. which highlights their strengths and weaknesses. The top 2 relevant systems are shortlisted for evaluation based on several measurement parameters. To facilitate the technical evaluation of the systems we set up a suitable test bed. Although the initial analysis examines both open-source and licensed schedulers, only the open-source schedulers will be considered for an in-depth analysis. This analysis serves as a foundation for decision-making and provides valuable information for selecting the most suitable ETL scheduler for prototyping data-centric scheduling. Finally, the section concludes with a discussion of the results, offering insights derived from the evaluation.

The remainder of this paper is structured as follows. In section 4, the thesis paper begins by discussing the motivation behind data-centric and distributed scheduling and explores the advantages it offers. By addressing the motivation and benefits of data-centric scheduling, this section sets the stage for the subsequent implementation and validation phases of the research. This section also outlines the high-level functional and non-functional requirements set for the scheduling system by our industry partner to guide the selection and validation of the ETL scheduler.

The next section focuses on the chosen system's ability to fulfill the defined requirements specified by the industry partner validating the functional requirements. Strengths and weaknesses of the system are identified and discussed, providing insights into its positive aspects and areas for improvement It includes systematic verification of each requirement, ensuring alignment with expectations. It zooms into key aspects such as parallelism, failover mechanisms, and integration with API and UI. Additionally, it explores the implementation of job versioning, triggering mechanisms, optimal installation methods, and effective data transmission. The section combines theoretical analysis with practical testing to demonstrate the system's efficiency and adherence to specified functional requirements.

Section 6 "Exploration of data-centric scheduling approaches" starts by examining the challenges in managing multi-workflow pipelines. The focus then shifts to the benefits of using data insights and metadata in decision-making within ETL scheduling. The paper discusses the addition of data-driven features to the ETL scheduler and the technical details involved, including building a custom metadata plugin. Practical use cases are presented to validate the effectiveness of these enhancements. Finally, it highlights smarter scheduling decisions made through actual queried data, again supported by real-world examples. This section highlights the practical benefits of data-centric approaches in complex ETL environments.

Finally, we acknowledge and discuss the limitations encountered during the research and extension process of the proposed ETL scheduler. We conclude with a concise summary of the key findings, contributions, and outcomes of the research and outline potential directions for future work.

2. Background and Related Work

2.1 Evolution of ETL scheduling

ETL scheduling has evolved over time to meet the growing needs of data integration and management. The history of ETL scheduling can be traced back to the early days of computing. In this section, we attempt to acquire a brief overview of its progression (16, 17).

Early manual scheduling: In the early days of data integration, ETL processes were manually scripted. Developers wrote custom scripts or programs to extract data from various sources, apply transformations, and load it into the target system. These processes required manual intervention and lacked automation.

Introduction of cron: The introduction of job scheduling tools, such as
of cron, a time-based job scheduler in Unix-like operating systems, revolutionized ETL scheduling. It allowed users to schedule and automate the execution of ETL tasks at specific intervals, days, or times. Cron provided a more efficient and reliable way to manage recurring ETL jobs.

Growth of dedicated ETL tools: As data volumes and complexity increased, dedicated ETL tools emerged. These tools provided a visual interface and a range of functionalities to design, schedule, and manage ETL workflows. Examples of popular ETL tools include Informatica PowerCenter, IBM InfoSphere DataStage, and Microsoft SQL Server Integration Services (SSIS). These tools offered advanced capabilities like data mapping, transformation libraries, and job dependency management.

Enterprise scheduling solutions: With the expansion of enterprise data ecosystems, the need for centralized scheduling and coordination of ETL processes became crucial. Enterprise job scheduling solutions, such as Control-M, Tivoli Workload Scheduler, and Autosys, were developed to handle complex dependencies, job orchestration, and workload automation across multiple systems. These tools allowed for the management of complex dependencies, job orchestration across multiple systems, and workload automation.

Cloud-Based and Serverless Scheduling: The rise of cloud computing introduced new possibilities for ETL scheduling. Cloud-based solutions like AWS Data Pipeline, Google Cloud Dataflow, and Azure Data Factory offer scalable, serverless, and managed services for ETL scheduling. These platforms provided capabilities for integrating and orchestrating data workflows across diverse cloud services and on-premises systems.

Containerized Scheduling: Containerization technologies, notably Kubernetes, brought containerized scheduling to ETL processes. Container orchestration platforms enable the deployment, scaling, and management of ETL workflows as containerized applications. This approach provides flexibility, portability, and efficient resource utilization.

Overall, the history of ETL scheduling reflects the evolution of technology and the increasing demand for efficient and automated data integration processes. From manual scripting to sophisticated scheduling tools and cloud-based solutions, ETL scheduling has advanced to accommodate the complexities of modern data ecosystems.

2.2 State of existing solutions for ETL scheduling

There is a myriad of existing solutions for ETL scheduling that we can consider for automating and managing our data integration workflows. When selecting an ETL job scheduler for your organization, it's important to determine why you need an ETL job scheduler and identify your specific requirements. Next, evaluate if the tool offers the necessary features to meet those requirements. Choose a vendor with a good reputation in the industry and positive reviews online. Compare different tools based on functionality, usability, features, and price. Opt for a vendor with a flexible and transparent pricing structure, but avoid compromising on quality for a cheaper quote. It's essential to find a balance between meeting your organization's needs and maintaining a reasonable cost.

There are many open-source and licensed scheduler options that we can consider.
Informatica is a well-known ETL solution that caters to large organizations and has strong scheduling capabilities, scalability, and a variety of features. Some popular and widespread ETL scheduling systems include Microsoft SQL Server Integration Services (SSIS), which works well with Microsoft technologies. Autosys requires licensing and is good if you have complex interdependence in your processing workflow. Quartz is good if you need something more than CRON or Task Manager. Apache Airflow is an open-source platform with a vibrant community and is a good way to manage both streaming and batch ETL stages (22). Control-M is widely used for scheduling purposes across the IT industry. ActiveBatch is a multifaceted job scheduler that enables the utilization of various scheduling options. The platform empowers users to schedule jobs according to resources, events, and time, providing flexibility and control. Apache Nifi, primarily known for its data flow management capabilities, also provides scheduling features. Users can schedule the execution of data flows, making it suitable for ETL scheduling tasks (19).

Other than the learning curve, license cost, and ease of use, job opportunity is one of the criteria that define popularity.

These are just a few examples of ETL scheduling solutions available on the market. The choice of the right solution depends on your specific requirements, such as the complexity of your data integration workflows, integration needs with other systems, scalability requirements, and budget considerations. In the next sections, we will delve a little deeper into the open-source and licensed scheduling tools.

2.3 Methodology and Literature Analysis

Through a short literature survey on topics surrounding ETL scheduling, we aim to understand the current state of the technologies as well as the research progression and challenges related to the field. All the publications that have been chosen for the survey appear indexed either in the DBLP computer science bibliography or in Crossref.

These papers were retrieved through a keyword search and then we filtered the papers based on our topic of interest. The search terms we used included “ETL scheduling”, “data-driven ETL” and “ETL scheduler”.

The papers that were reviewed mostly talked about scheduling algorithms, optimizations related to ETL pipelines, and improvement of ETL performance. To get a well-rounded grasp of the topic, we also scoured several blog articles that talked about ETL scheduling and best practices related to it.

2.4 Related Work

ETL task scheduling plays a crucial role in ensuring data integrity, timeliness, efficiency, and quality throughout the data integration process. It ensures that data is processed and updated regularly, maintaining consistency across different systems. ETL tasks can be resource-intensive and time-consuming, particularly when working with substantial amounts of data.

In this section, we look into some of the research related to workflow and pipeline scheduling in ETL & stream processing domains to shed light on the work that has been undertaken to solve the complex problem of ETL scheduling.

By juxtaposing these works against each other, we can draw parallels and contrasts in their approaches, methodologies, and focus areas, enriching our understanding of the diverse strategies employed in ETL task scheduling and optimization.

Optimization of Resource Utilization

Zhenxue et al. (2020) (1) proposed a multi-objective scheduling optimization (MOETSA) algorithm to shorten the average execution time of ETL tasks in data integration and utilize the resources of the cluster reasonably. Their algorithm captured performance information for each node, evaluated task execution time using a random forest model, and estimated load balance states considering node performance differences. The study focused on CPU and memory utilization as performance metrics. The effectiveness of the algorithm was demonstrated, and this research aligns with the need for advanced algorithms in optimizing ETL tasks, specifically in distributed systems.

Yang and Xu (5) in their work, address the challenges of efficient scheduling and optimization in the ETL process within a service-oriented architecture in a big data environment. The goal was to resolve issues such as slow data integration, inefficient task scheduling, high bandwidth consumption, and low execution efficiency. The proposed solution consists of a distributed scheduling and execution framework along with a method for scheduling and executing the ETL process. The framework utilizes performance differences between nodes and a load-balancing index to dynamically allocate the ETL process to achieve load balancing. Additionally, a locality-aware strategy is applied to optimize the scheduling of the ETL service, taking into account data volume and network distance to improve efficiency.

This study's approach to handling big data environments also complements the work of Seenivasan (2) and further emphasizes the importance of efficient resource allocation.

Both of the studies (1) & (5) focus on optimizing resource utilization in ETL processes. Zhenxue et al. use a multi-objective scheduling optimization algorithm considering CPU and memory utilization, while Yang and Xu address scheduling in big data environments, emphasizing load balancing and performance differences between nodes. These studies can be contrasted in their approach to resource optimization, with Zhenxue et al. focusing more on the algorithmic efficiency of individual tasks and Yang and Xu on the overall system efficiency in a distributed environment.

Scheduling and Pipeline Optimization

Seenivasan (2) delves into the basics of optimizing ETL pipelines, which include optimizing the source data, parallel processing, caching, and incremental load (a technique that can improve ETL jobs' performance by only processing new or changed data rather than processing the entire dataset each time the ETL job is run). It emphasizes monitoring for performance management and strategies like indexing and common lookup files for performance optimization. Using common lookup files, disabling and enabling indexes during ETL jobs, and scheduling the jobs after peak hours are other ways mentioned that optimize scheduling performance.

Appropriate scheduling strategies are required for orchestrating the smooth flow of data toward the target data stores. The study by Karagiannis et al. (3) explores the impact of four scheduling policies on ETL performance, focusing on memory consumption and execution time. These policies range from fair scheduling to those emphasizing input queue management and high data processing rates, with another combining these elements for flow parallelization. Their findings reveal that policies geared towards minimizing cost and memory yield significant improvements in execution time and memory usage, respectively. The study also recommends workflow segmentation and mixed policy strategies for enhanced resource allocation and time efficiency.

Seenivasan's work on optimizing ETL pipelines through incremental load, parallel processing, and caching can be compared with Karagiannis et al.'s study on the impact of various scheduling policies on ETL performance. Both studies aim at improving the efficiency of ETL processes, but from different angles – Seenivasan through pipeline optimization techniques and Karagiannis et al. through the implementation of different scheduling policies.

Task Clustering in Data Warehousing

Bruno et al., in their paper (6) explore the use of task clustering to simplify the development of data warehousing processes. These processes involve integrating data from various sources, and task clustering allows the grouping of related tasks to form higher-level constructs. The authors propose a task-clustering technique that organizes finer-grained tasks into collections, allowing for different levels of abstraction and accommodating various project stakeholders. The categorization of clusters enables the identification and classification of patterns that can be reused across different data integration processes, facilitating communication and data interchange within workflows.

This study, focusing on process simplification and efficiency, aligns with the overarching theme of optimizing ETL processes seen in other research.

Real-Time Data Processing

There are also several works related to scheduling in data warehouses. Song et al. address the challenges of executing ETL tasks in real-time data warehouses. Their research proposes an Integration Based Scheduling Approach (IBSA) to handle the triggering, scheduling, and balancing of updates and queries. Experiments demonstrate that IBSA optimizes task order, efficiently utilizes system resources, and improves query response time and the real-time capabilities of ETL processes.

The paper "Metadata-Driven Industrial-Grade ETL System " (9) discusses the limitations of traditional ETL technology in handling real-time data from diverse sources. The authors

propose an automated ETL system that improves versatility and efficiency in data processing. To achieve this, they optimized the ETL workflow using a custom metadata management framework. Experimental results using actual railway KPI data demonstrate that the proposed metadata-supported automated ETL system effectively processes data using open-source distributed big data technologies. This research complements the other studies by focusing on metadata utilization for process optimization.

Song et al.'s research on real-time data warehouses using IBSA can be juxtaposed against the study on metadata-driven ETL systems for real-time data processing. Both focus on enhancing the real-time processing capabilities of ETL systems but approach it differently – Song et al. through task scheduling and the latter through metadata optimization. Song et al.'s focus on real-time data warehousing also complements the scheduling optimization goals seen in the works of Zhenxue et al. and others.

Algorithmic Approaches to Scheduling

In the paper titled “Data-Driven Intelligent Scheduling For Long Running Workloads In Large-Scale Datacenters”(12) it is stated that the current average data center utilization in the industry is alarmingly low, ranging from 6% to 12%. This low utilization hurts both operational and capital costs. To address this issue, there is an urgent need for new and effective scheduling and resource allocation approaches in large-scale enterprise data centers. Predictive scheduling techniques, such as reservation, auto-scaling, migration, and rescheduling, are crucial for improving efficiency in long-running workloads. Therefore, it is important to explore intelligent scheduling techniques that leverage predictive knowledge. In the paper, they further elaborate on the intelligent cloud data center scheduler developed, which incorporates resource-to-performance modeling, predictive optimal reservation estimation, QoS-aware predictive scheduling, and strict adherence to service level agreements (SLAs) to maximize resource efficiency across multiple dimensions.

The approaches discussed in this paper discuss a broader range of predictive scheduling techniques for resource allocation in data centers and contribute to the understanding of how algorithmic approaches can enhance scheduling efficiency in different contexts.

Adaptability in ETL Processes

Wojciechowski (4) introduces the E-ETL framework, designed for the automatic repair of ETL workflows in response to structural changes in External Data Sources (EDSs). This innovative framework can detect changes in EDSs, repair affected workflow segments using customizable algorithms, and integrate with various ETL engines. The paper extends previous work by providing detailed descriptions of the change detection mechanism and evolution rules, emphasizing the automatic repair of ETL workflows based on user-defined rules. This

research emphasizes adaptability in ETL processes, a crucial aspect alongside the efficient scheduling approaches discussed in other studies.

Qu. and Deßloch (15) explore an incremental ETL pipeline for data warehouse maintenance on-demand, employing parallel processing for simultaneous maintenance jobs. However, to support incremental joins and slowly changing dimension tables, different pipeline operators may concurrently access or update the tables, leading to inconsistencies. The paper introduces two types of consistency zones, addressing this issue. Through examples and appropriate implementations, the paper explains the potential consistency anomalies and offers solutions for incremental joins and slowly changing dimensions within the incremental ETL pipeline engine. This work thereby adds to the collective understanding of ETL process optimization, particularly in incremental processing.

Wojciechowski's E-ETL framework for automatic repair of ETL workflows in response to changes in External Data Sources (EDSs) shares a common theme of adaptability with Qu. and Deßloch's work on incremental ETL pipelines. Both studies emphasize the need for ETL processes to be dynamic and responsive to changes, whether they are in the source data or the data warehouse structure.

In conclusion, the exploration of various approaches to ETL task scheduling and optimization in this section underscores the complexity and multifaceted nature of data integration processes. This comprehensive overview not only enhances our current knowledge but also lays a solid foundation for future advancements in data-centric scheduling, setting the stage for more efficient, robust, and adaptable data integration systems.

2.5 Current Trends

To get some further insights into ETL Scheduling and factors to consider while building an ETL Scheduler, we delve into some interesting insights from relevant technical blogs.

ETL scheduling metrics

A recent article (10) gives us some interesting insights on ETL scheduling metrics. To measure and report the impact and value of your ETL scheduling strategy, we need to assess various metrics. These metrics include job duration, frequency, success rate, latency, and backlog. By monitoring these metrics, you can identify and address bottlenecks, optimize resource utilization, and ensure timely data delivery, thereby demonstrating smooth ETL processes that meet SLAs and deliver quality data to stakeholders. To collect, store, analyze, and visualize these ETL metrics, you can utilize a range of tools. ETL schedulers like Airflow, Luigi, or Cron can handle the scheduling aspect. ETL monitors such as DataDog, Splunk, or Grafana can provide real-time monitoring. ETL dashboards like Tableau, Power BI, or Superset enable

visual representation of metrics, while ETL documentation tools like Dataedo, Data Catalog, or Apache Atlas help in documenting the processes.

To maximize ETL scheduling efficiency, it's crucial to utilize a combination of tools for task orchestration, real-time monitoring, visual analytics, and process documentation. This integrated approach enhances operational transparency and optimizes performance

Data-driven scheduling

Data-driven scheduling is another feature that optimizes ETL pipelines. A new feature introduced in Apache Airflow 2.4 (13) called data-driven scheduling enhances Airflow's capability to handle data-driven dependencies. Data-driven scheduling allows DAG authors to inform Airflow when upstream tasks are completed, triggering the execution of downstream tasks. This feature simplifies the design process for common data engineering patterns and is expected to be of great benefit.

Utilizing a scheduling dashboard

A good ETL scheduler would be made even more effective with the use of a good scheduling dashboard (14) that highlights where to make changes in the schedule. A good ETL scheduling dashboard should have the following key features:

- Workflow visualization: Clear and intuitive visualization of ETL workflows, showing status, progress, and performance of tasks, jobs, and pipelines.
- Scheduling flexibility: Ability to define frequency, timing, priority, and triggers, and modify schedules easily.
- Alerting and notification: Customizable alerts for issues or failures in ETL processes, with options for frequency, channel, recipients, and severity.
- Reporting & analysis: Comprehensive reports on execution time, duration, throughput, success/error rates, and resource consumption.

Gartner Magic Quadrant for Data Integration Tools

The Gartner Magic Quadrant for Data Integration Tools provides a visual snapshot of the market's competitive landscape, showcasing how various competitors measure up against Gartner's criteria for completeness of vision and ability to execute. It serves as a strategic resource in selecting the right tool for an organization's data integration needs.
It is a research tool developed by Gartner Inc., providing a graphical competitive positioning of technology providers in markets where growth is high and provider differentiation is distinct. Vendors are categorized as Leaders, Challengers, Visionaries, or Niche Players based on their ability to execute their strategies and their understanding of market trends. *(Source: Gartner Inc.)*

The 2022 vs. 2023 comparison depicts market dynamics and shifts in vendor strategies. Leaders like Informatica, Oracle, & IBM consistently demonstrate industry dominance with robustness & visionary approaches. Meanwhile, Microsoft & SAP maintain strong positions through continuous innovation & customer satisfaction. Challengers, including AWS & TIBCO Software, showcase growth potential signaling strong market competition. Visionaries like Palantir and SnapLogic push the envelope with unique and progressive solutions, potentially disrupting the status quo. In the Niche Players quadrant, Safe Software & other specialized vendors cater to particular market segments. The evolution from 2022 to 2023 underscores the

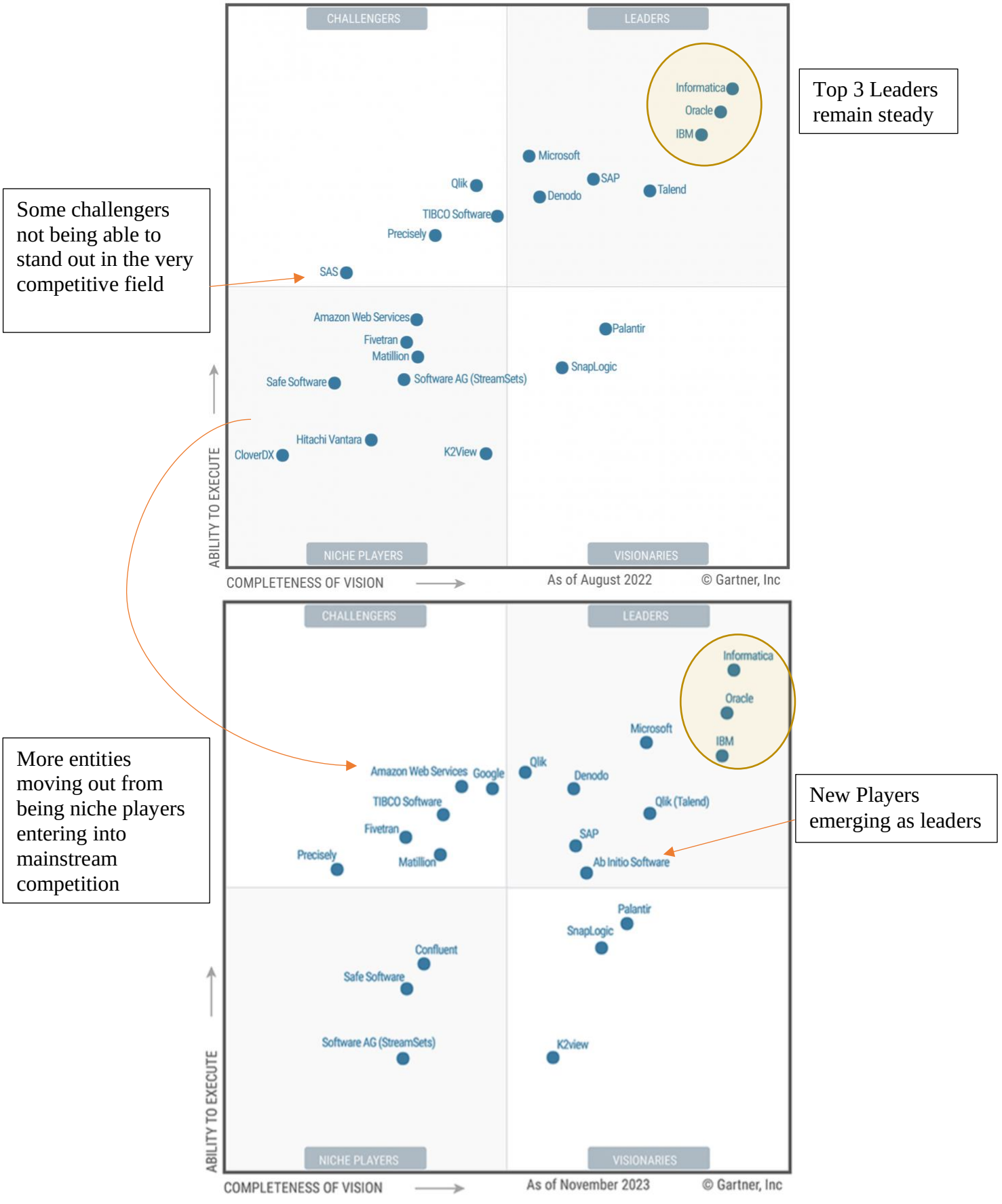

Figure 2.1: Gartner Magic Quadrant for Data Integration Tools 2022 & 2023.
[Source: Figure modified from Gartner, Inc. (https://www.gartner.com)]

3. Comparative Analysis of Existing Solutions

ETL solutions differ in aspects such as their implementation complexity, processing capabilities, fault tolerance, scalability, and cost efficiency. Some solutions excel at handling high data volumes and complex transformations, while others are recognized for their user-friendly interface and ease of integration with existing systems. Features like real-time data processing, concurrent job execution, and event-driven scheduling further differentiate these solutions. A comparative analysis helps identify their unique strengths, weaknesses, and suitability for different business needs.

The choice of an ETL solution is contingent on an organization's specific requirements. Some organizations may prioritize cost efficiency and ease of use, while others might prioritize advanced functionalities and scalability. Comparative analysis helps businesses navigate these choices, facilitating a more informed decision-making process that aligns with their strategic goals and resource constraints.

3.1 Licensed Schedulers (Theoretical Analysis)

Licensed ETL schedulers typically provide comprehensive support, documentation, and training resources, ensuring organizations receive reliable technical assistance. They often offer more advanced features, scalability options, and integrations with a wide range of data sources and platforms. Licensed ETL schedulers prioritize security, providing robust authentication, encryption, and access control mechanisms. Furthermore, licensed solutions often have a dedicated development team that continually improves and updates the software based on customer feedback and industry trends.

Overall, licensed ETL schedulers provide a more comprehensive and supported solution, offering organizations peace of mind and enhanced functionality for their data integration needs.

Our research predominantly explored open-source schedulers. This is due to the licensing costs associated with testing a full-fledged version of the licensed products and the accessibility challenges presented by the trial versions of their commercial counterparts. To understand the market better, let’s also take a look into some of the most popular licensed solutions existing for ETL scheduling,

1. Informatica PowerCenter:

- Breadth of Features: Informatica PowerCenter offers comprehensive data integration capabilities, including broad data source connectivity, workflow orchestration, advanced data transformation, error handling, and scalability features.
- Depth of Features: It provides robust task scheduling, extensive monitoring and alerting options, security features, error handling, and recovery mechanisms. It also supports performance optimization techniques.
- Specific Needs and Requirements: Informatica PowerCenter integrates well with various data ecosystems, provides scalability, and offers different licensing options. It has a strong support community and documentation.
- Cons:
    - Cost: Informatica PowerCenter is a commercial product, and its licensing and maintenance costs can be relatively high for small or budget-constrained organizations.
    - Learning Curve: PowerCenter has a complex architecture and can have a steep learning curve for users unfamiliar with the tool, requiring additional training and resources.

2. SQL Server Integration Services (SSIS):

- Breadth of Features: SSIS provides connectivity to Microsoft SQL Server and other data sources. It offers workflow orchestration and basic data transformation capabilities.
- Depth of Features: SSIS provides flexible task scheduling, monitoring, and basic error handling and recovery mechanisms.
- Specific Needs and Requirements: SSIS is well-suited for organizations heavily invested in the Microsoft ecosystem, particularly those using SQL Server. It comes with SQL Server licensing and support.
- Cons:
    - Limited Platform Support: SSIS is primarily designed for Microsoft SQL Server environments, limiting its compatibility with other database systems and platforms.
    - Scalability Challenges: Scaling SSIS can be challenging, especially for large-scale data processing requirements, as it relies on the underlying SQL Server infrastructure.

3. IBM InfoSphere DataStage:

- Breadth of Features: InfoSphere DataStage supports various data sources, workflow orchestration, and data transformation capabilities.
- Depth of Features: It offers flexible task scheduling, monitoring and alerting options, security features, error handling, recovery mechanisms, and performance optimization techniques.
- Specific Needs and Requirements: InfoSphere DataStage is a good choice for organizations using IBM technologies, as it integrates well with the IBM ecosystem. It provides scalability and offers different licensing options.
- Cons:
    - Complexity: DataStage has a complex and feature-rich environment, which may require specialized skills and knowledge to effectively leverage its capabilities.
    - Cost: Licensing fees for InfoSphere DataStage can be relatively high, making it less suitable for organizations with tight budgets.

4. Oracle Data Integrator (ODI):

- Breadth of Features: ODI supports diverse data sources, workflow orchestration, and advanced data transformation.
- Depth of Features: ODI provides comprehensive task scheduling, monitoring, security features, error handling, and recovery mechanisms. It also supports performance optimization techniques.
- Specific Needs and Requirements: ODI is an ideal choice for organizations heavily reliant on Oracle technologies. It offers scalability, multiple licensing options, and has a supportive user community.
- Cons:
    - Licensing Costs: Oracle's licensing model can be expensive, and the cost of ODI may not be feasible for small to mid-sized organizations.
    - Performance Limitations: While ODI offers robust data integration capabilities, some users have reported performance limitations when dealing with large volumes of data.

5. Google Cloud Scheduler:

- Breadth of Features: Google Cloud Scheduler is a cloud-native scheduler, primarily designed for managing tasks and workflows within the Google Cloud Platform ecosystem.
- Depth of Features: It offers flexible scheduling options, monitoring, and alerting capabilities.
- Specific Needs and Requirements: Google Cloud Scheduler is suitable for organizations utilizing the Google Cloud Platform. It provides seamless integration with other Google Cloud services and offers scalability.
- Cons:
    - Vendor Lock-In: Google Cloud Scheduler is tightly integrated with the Google Cloud Platform, which may limit portability and make it challenging to migrate to other cloud providers.
    - Limited Customization: While Google Cloud Scheduler offers reliable scheduling capabilities within its ecosystem, customization options may be limited compared to other standalone schedulers.

3.2 Open Source Schedulers

Open-source ETL scheduling solutions are key to many businesses' data integration and transformation strategies. They come with distinct advantages and disadvantages

Advantages:

- Cost-Effectiveness: Open-source ETL tools are usually free or significantly cheaper than their proprietary counterparts, reducing the overall cost of data management.
- Flexibility: These tools often allow more customization and modification, enabling businesses to tailor the tool to their unique needs.
- Community Support: Open-source tools typically have active user communities. This community support can provide help through forums and often contributes to the development and improvement of the tool.
- Transparency: Open-source software provides visibility into the code, which can enhance understanding and trust in the tool's capabilities and potential bugs.

Disadvantages:

- Technical Complexity: Open-source ETL tools can require substantial technical know-how. Their setup, customization, and maintenance might necessitate experienced developers.
- Limited Customer Support: While community support can be beneficial, the absence of dedicated, professional customer support can make problem resolution slower and more challenging.
- Uncertain Longevity: The sustainability of open-source projects depends on the ongoing commitment of their communities. There's always a risk that development may slow or cease, which could leave users without updates or security patches.
- Scalability Issues: Some open-source ETL tools may struggle with high data volumes or complex transformations, potentially affecting performance and reliability.

Selecting an open-source ETL scheduling solution requires a careful balance between the needs of the organization, available resources, and these inherent advantages and disadvantages. Here's a comparitive analysis of some popular open-source ETL schedulers:

1. Apache Airflow:

- Breadth of Features: Apache Airflow offers a wide range of features, including workflow orchestration, task scheduling, dependency management, monitoring, and extensibility through custom operators and hooks.
- Depth of Features: Airflow provides a highly scalable and distributed architecture, robust error handling and retries, support for different types of data sources and sinks, and a rich ecosystem of plugins and integrations.
- Specific Needs and Requirements: Apache Airflow is well-suited for organizations that require flexible and scalable workflow management, especially in cloud-based environments. It has an active community and strong documentation.
- Cons:
    - Learning Curve: Apache Airflow (code-first approach) has a steep learning curve, especially for users new to the tool or the concepts of Directed Acyclic Graphs (DAGs) and workflows. The platform might not be intuitive to new users.
    - Infrastructure Requirements: Airflow requires additional infrastructure setup and maintenance, including a database, message broker, and worker nodes, which can be complex and time-consuming.

2. Apache NiFi:

- Breadth of Features: Apache NiFi offers a comprehensive set of features for data ingestion, routing, transformation, and output. It supports real-time data streaming, data governance, security, and data provenance.
- Depth of Features: NiFi provides a visual interface for designing data flows, extensive connectivity to various data sources and systems, data transformation capabilities, and robust monitoring and management features.
- Specific Needs and Requirements: Apache NiFi is ideal for organizations dealing with complex data routing and transformation scenarios, especially in the context of data streaming and IoT data processing.
- Cons:
    - Complexity: Apache NiFi has a complex interface and configuration options, requiring a certain level of expertise to effectively utilize its capabilities.
    - Resource Intensive: NiFi's resource requirements can be significant, especially when dealing with large volumes of data, which may necessitate additional hardware or scaling considerations.

3. Talend Data Integration:

- Breadth of Features: Talend Data Integration offers a comprehensive set of features for data integration, including data profiling, data quality, data transformation, connectivity to various data sources, and scheduling options.
- Depth of Features: Talend provides a visual development environment, a wide range of pre-built connectors, data mapping and transformation capabilities, data lineage, and monitoring and management features.
- Specific Needs and Requirements: Talend Data Integration is suitable for organizations that require a robust and user-friendly platform for end-to-end data integration, including data profiling, data quality management, and data transformation.
- Cons:
    - Limited Support: The community edition of Talend Data Integration may have limited official support compared to the commercial version, which can be a drawback for organizations requiring assistance or troubleshooting.
    - Complexity for Advanced Features: Certain advanced features and functionalities may only be available in the commercial edition of Talend, requiring additional investment.
    - The open-source community version is not as feature-rich as the enterprise version.

4. QUARTZ:

- Breadth of Features: QUARTZ is a feature-rich, open-source job scheduling library that provides flexible and powerful scheduling options.
- Depth of Features: QUARTZ offers support for complex scheduling requirements, including cron-like expressions, time zones, and advanced scheduling features like misfire handling, job listeners, and triggers.
- Specific Needs and Requirements: QUARTZ is suitable for organizations that require a lightweight and versatile job scheduling solution for automating recurring tasks and workflows.
- Cons (35):
    - Limited Functionality: Built-in capability for multiple execution nodes via pooling or clustering is not provided. No monitoring or alert. Insufficient mechanisms for dealing with errors/failures and recovery
    - Lack of Visual Interface: QUARTZ is primarily a Java library and may not have a user-friendly visual interface for designing and managing workflows. It has no administration UI that allows all job scheduling and configuration to be done

5. Luigi:

- Breadth of Features: Luigi is an open-source workflow scheduler that focuses on building data pipelines. It provides features for defining dependencies, task scheduling, monitoring, and error handling.
- Depth of Features: Luigi offers a simple and intuitive Python-based workflow definition syntax, support for both sequential and parallel execution, and extensibility through custom tasks and targets.
- Specific Needs and Requirements: Luigi is suitable for organizations that prioritize simplicity and prefer a Python-centric approach for building and managing data pipelines.
- Cons (30):
    - Lack of Real-time Data Processing: Luigi is primarily designed for batch processing and may not be well-suited for real-time data integration or streaming scenarios.
    - “The assumption is that each task is a sizable chunk of work. While you can probably schedule a few thousand jobs, it’s not meant to scale beyond tens of thousands” (30).
    - Luigi does not support the distribution of execution. This becomes significant when you operate workers executing thousands of tasks daily, as the worker nodes begin to experience overloading.

- Luigi does not come with built-in triggering, and you still need to rely on something like Crontab to trigger workflows periodically (30).
- Python Dependency: Luigi relies on Python for defining workflows, which may be a limitation for organizations that prefer other programming languages or have limited Python expertise.

6. Pentaho Data Integration (Kettle):

- Breadth of Features: Pentaho Data Integration, also known as Kettle, is an open-source ETL tool that offers comprehensive features for data integration, including data extraction, transformation, loading, and scheduling.
- Depth of Features: Kettle provides a visual development environment, a wide range of connectors and transformations, metadata management, data profiling, and job orchestration capabilities.
- Specific Needs and Requirements: Pentaho Data Integration is suitable for organizations that require an open-source ETL tool with extensive functionality, flexible data integration options, and strong community support.
- Cons:
  - Limited Performance Optimization: Pentaho Data Integration may have performance limitations when handling large volumes of data or complex transformation scenarios, requiring additional optimization efforts.
  - Inadequate community support: If an element isn't working, we have to wait until the next version is released.

7. Apache Dolphin Scheduler

- Breadth of Features (32, 33): It is a distributed and extensible workflow scheduler platform with powerful DAG visual interfaces. Dolphin Scheduler helps to solve complex job dependencies in the data pipeline.
- Depth of Features: Reduces the need for code by using a visual DAG structure. Users can now drag-and-drop to create complex data workflows quickly, thus drastically reducing errors. DS provides decentralized multi-worker and multi-master for high reliability, overload processing, and self-support. It also has good error handling and suspension features. High expandability is ensured with scheduling capabilities scaling linearly with cluster size.
- Specific Needs and Requirements: It is suitable for enterprise-level scenarios. Its visualized process definitions empower non-coders to create complex workflows, enhancing user-friendliness. Numerous multinational corporations, such as Lenovo,

Dell, and IBM China, among others, employ Dolphin Scheduler. It caters to a plethora of scenarios, supporting various task types like Spark, Hive, Python, Flink, and MapReduce, along with promoting efficient multitenancy.

- Cons:
    - Maturity: It is still a fairly new project, and it might not have had the same level of maturity as other, more established workflow management systems which could potentially lead to less stability, fewer features, and less community support.
    - Documentation: The documentation for newer projects can often be less complete, less detailed, or less clear than for older, more established projects. This could potentially make it more difficult for new users to get started with the system or for more experienced users to resolve issues.

3.3 Methodology used to shortlist ETL schedulers

We aim to evaluate the top 2 open source ETL schedulers in depth to understand their potential faults and shortcomings. This evaluation will serve as a basis to consider whether an existing scheduling solution will help fulfill our requirements.

One of the important features that we used to shortlist the ETL schedulers was their ability to execute the tasks in a distributed manner. Other than distributed job scheduling support, we also tried to gauge from various sources if the scheduler supports restarting activities after a down phase and also whether it supports failover clustering. The ETL scheduler we aim for must be able to detect a service failure, restart the failed service, or reroute the tasks to the next node.

In the previous sections, we have researched the market to identify the possible ETL schedulers that might be a good fit. In this evaluation, we have limited our options to testing out only open-source schedulers due to licensing costs and the ability to extend the chosen product according to our needs.

The open-source schedulers given in the previous section constitute the most popular and robust options. To drill down and see which ones would be closer to our requirements, we took a closer look at the seven open source options mentioned above.

Luigi and QUARTZ may not fully satisfy our criteria.

Luigi lacks inbuilt support for distributed execution, making it less suitable for environments that require distributed workflow executions. Compared to other options, Luigi's community is not as active. An active community is often indicative of regular updates, support, and feature improvements.

QUARTZ, while mature and robust, lacks the advanced features that might be crucial for complex workflows, and it doesn't have a distributed execution model. QUARTZ is primarily a Java library and does not have a user-friendly visual interface for designing and managing workflows.

Talend Data Integration does have an open-source version, known as Talend Open Studio for Data Integration. This version provides a wide range of capabilities and features that can help with data integration tasks.

However, Talend also offers premium, enterprise-level versions of its software that come with more advanced features, such as collaboration, project management, and advanced technical support. These enterprise-level versions are not open source and require a paid subscription. Therefore, while Talend does offer an open-source product, it's important to understand that some advanced features may only be available in premium paid versions.

Pentaho Data Integration (Kettle) is essentially an ETL tool with scheduling capabilities. It allows you to schedule ETL jobs, but its features lean more toward data transformation and integration rather than complex workflow scheduling.

Apache Airflow, on the other hand, is designed from the ground up as a workflow management and scheduling platform. It is feature-rich in terms of scheduling and managing complex workflows. It supports dynamic pipeline generation, has a robust system for handling dependencies, and has more advanced scheduling and retry capabilities.

Therefore, if scheduling and orchestration of complex workflows are our primary focus, Apache Airflow tends to be the more specialized tool for these tasks. Apache Dolphin Scheduler, like Apache Airflow, is a dedicated workflow scheduler and has many robust and feature-rich scheduling capabilities

Hence, after considering the research on several parameters, including pros and cons, we have decided to evaluate the following options:

- **Apache Airflow**
- **Apache Dolphin Scheduler**

Apache Airflow was an obvious choice, as it is one of the most popular open-source ETL schedulers available, along with having an active community on GitHub. Another promising option was Apache NiFi but we decided against it as NiFi is perfect for basic big data ETL processes, while Airflow is the “go-to” tool for scheduling and executing complex workflows

as well as business-critical processes (13). Airflow also seems to have a broader approval based on the number of GitHub stars and forks, and also the number of contributors (29)

Although Apache Dolphin Scheduler is newer and the community is still growing, it promises to be a major challenger in the future. It offers scalable, fault-tolerant workflow management with extensibility and multi-tenancy support. It simplifies complex workflow orchestration, provides scheduling strategies, and allows monitoring and alerting. Its open-source nature and vital community support make it an ideal choice for automating and optimizing data workflows.

3.4 Evaluation of shortlisted ETL schedulers

This section presents an evaluation of shortlisted ETL schedulers to identify the more suitable scheduler that aligns with our needs. To make an informed decision, we start by setting up a distributed cloud setup on OpenStack. We identified the test scenarios that would be required to test our requirements and then discussed the results with regard to the strengths and weaknesses of each scheduler.

3.4.1 Test Bed Setup

In this section, we elaborate on the test bed setup followed for the two schedulers.

Apache Airflow

Apache Airflow is a powerful open-source platform used for orchestrating complex workflows and data pipelines. One of the key components of Airflow is the Celery Executor, which is a distributed task queue system. It facilitates parallel task execution across multiple workers thereby allowing job scaling.

The setup comprised of four OpenStack VMs. The first VM plays a central role by hosting the scheduler, web server, and a single Celery worker. The scheduler is responsible for managing the workflow, while the web server provides a user-friendly interface to monitor & manage the jobs. The Celery worker is responsible for executing tasks assigned to it by the scheduler.

The other three VMs each host an additional Celery worker. These workers form the distributed workforce, capable of executing tasks in parallel. This design allows for load balancing and increased performance, as tasks can be assigned to any available worker in the cluster.

To pass messages between the scheduler and the workers, a Redis broker is employed. Redis is a fast and efficient message broker that enables communication and coordination between

the various components of the distributed system. It acts as a communication channel, ensuring that tasks are dispatched to the appropriate workers and status updates are sent back to the scheduler.

For the database backend, Postgres is configured. Postgres is a reliable and scalable relational database system, suitable for managing Airflow's metadata, job states, and other essential information related to workflow execution.

With the setup in place, the Airflow native web server provides a clear and organized presentation of the executing jobs. This web-based interface allows users to monitor the progress of their workflows, view logs, and manage tasks effectively.

To monitor each of the Celery workers individually, we set up Celery Flower which is a web-based tool for monitoring and administrating Celery clusters. It provides real-time insights into task execution.

As a final step, an Extract, Transform, Load (ETL) job is set up in the Airflow system. This job handles the extraction of data from a source, applies simple transformations to the data, and loads the processed data into the Postgres database.

Apache Dolphin Scheduler

Apache Dolphin Scheduler is an open-source distributed and extensible workflow scheduler system that coordinates and orchestrates data processing tasks. The platform is designed to be fault-tolerant and scalable, addressing the complex dependencies in process scheduling.

For the setup of a test bed with Dolphin Scheduler, a Docker container was used to deploy a standalone server. Dolphin Scheduler conveniently provides us with a standalone server that can be used for a quick start in order to explore its features. The container includes the essential components of Dolphin Scheduler, such as the master server, worker server, and database, encapsulating everything needed for a quick start. Once the docker container was run, we could access the GUI with the default credentials. An in-memory H2 database simplifies the standalone setup as there is no need to connect to an external database in the beginning

We created a workflow for an ETL job through the web interface of Dolphin Scheduler.
This interface enables users to visually define tasks, dependencies, conditions, and execution paths, and to configure various task types, such as Shell, Spark, HTTP or Python tasks etc. This “visual-ui first” is also a major difference from Apache Airflow which follows a “code-first” approach.

The ETL workflow could be scheduled and monitored through the same interface. We can view execution logs, task status, and system metrics and also set up alerts to notify stakeholders of critical job statuses or failures.

3.4.2 Test Scenarios

Now to gauge if the shortlisted scheduler fits our purpose, we designed 3 test scenarios that could be executed. As both schedulers are different in their underlying setup a 1:1 test is not feasible but we tried to simulate the scenarios under the same conditions as far as possible.

Scenario 1: Restarting Activities after a Down Phase:

Here we simulate a situation where in an ETL job one of the tasks in the task queue is unable to execute or experiences downtime due to upstream factors etc. During this failure, jobs are queued up and awaiting execution. After the failure of the job, the system should automatically resume the execution of the queued jobs without resetting or starting them from scratch. We verify that the system automatically resumes the execution of the queued jobs from the failure point, preserving progress and not restarting.

Scenario 2: Failover Mechanism in a Distributed Setup:

This is essentially a test for redundancy. In a distributed setup with multiple nodes, we simulate a situation where one node fails and see if another one takes over.
We simulate this by setting up multiple worker nodes and then distributing a set of jobs amongst them. Midway through execution, we trigger a failure in one or several worker nodes. The system is monitored to verify that another worker node automatically detects the failure and takes over the tasks assigned to the failed node. We expect that the system should be able to detect this and automatically redirect the workload of the failed nodes to the remaining operational ones. We ensure that the tasks originally assigned to the failed node are completed successfully by the new worker node.

These 2 scenarios help us to assess the effectiveness and reliability of the distributed execution and failover clustering capabilities in handling failures and maintaining workflow continuity in the scheduler.

Scenario 3: Data centricity:

In this last scenario, we verify whether the schedulers support the data-centric execution of ETL Jobs. In a data-centric scheduler, the scheduler is aware of the nature of the job being executed and considers its characteristics when making scheduling decisions.

3.5 Discussion of Results

Apache Airflow effectively handled tasks in scenarios of downtime and node failure, illustrating robust failover and distributed execution capabilities. Scenario 1 showed jobs resuming post-failure without restarting, preserving task progress. In Scenario 2, redundancy was successfully tested -> the system redistributed tasks from failed nodes to operational ones, ensuring task completion.

However, scenario 3 revealed limitations. The current scheduler lacks data-centric execution of ETL jobs, not fully considering job characteristics when scheduling. This area needs an extension, suggesting potential for improvement in customizing task handling based on the job nature. The first two scenarios demonstrated reliability in handling failures and maintaining workflow continuity.

Apache Dolphin Scheduler has also been verified to be able to handle scenarios 1 and 2, but the verification for scenario 3 was not feasible. During the setup and testing of Dolphin Scheduler, a notable challenge encountered was the limited availability of documentation and support for troubleshooting. While the community is growing, the current maturity level may pose some difficulties in resolving minor setup issues.

The approach of defining Directed Acyclic Graphs (DAGs) through the Graphical User Interface (GUI) is indeed intuitive, and I recognize its potential, especially when orchestrating intricate workflows. However, I must admit to some reservations about the exclusive use of no-code solutions for executing complex tasks. While this abstraction layer offers convenience, there might be some specific trade-offs.

One aspect that proved somewhat challenging in my research was the language barrier. It's understandable that, given its broader popularity outside the English-speaking tech community, many educational resources are available only in Mandarin. While this speaks to its international appeal, it may present an obstacle for individuals seeking to learn more about Dolphin Scheduler in other regions. Researching online provided limited assistance, and the predominance of resources in Chinese added a layer of complexity for non-Chinese speakers. This is certainly an area for potential growth, and it would be beneficial to see resources developed to accommodate a more diverse global audience.

As a result of the initial experimentation with both schedulers, we have decided to proceed with Apache Airflow as our chosen scheduler for further exploration in the coming sections.

4. Integrating Distributed & Data-Centric ETL Scheduling: Defining Functional Requirement Specifications

4.1 Motivation

Why do we need distributed job schedulers?

Distributed job schedulers automate scheduled tasks across multiple servers. They can be installed on multiple machines, allowing users to sequence tasks across different servers and creating a distributed workflow. Tasks can be executed periodically or on an ad hoc basis, supporting both routine and one-off operations. This functionality fosters efficient task management and ensures seamless execution of complex workflows, even supporting parallel job executions. These schedulers offer a robust solution for managing automated tasks in diverse IT environments and also replace the fragmented approach of traditional job scheduling, providing a unified platform to schedule and automate tasks across multiple machines and server types.

One of the key benefits of distributed job schedulers is their fault tolerance. Unlike traditional schedulers, which halt critical jobs if a machine goes down, distributed systems can reroute the affected jobs to operational machines, ensuring uninterrupted workflow. Hence, distributed job schedulers play a pivotal role in ensuring seamless, reliable task scheduling and execution in today's complex IT landscape.

What would be the significance of designing a data-centric job scheduler?

Data-centric scheduling is an approach in which the job scheduling process is assisted based on the data insights derived from the source data and metadata that it is processing. In a data-centric scheduling system, the state and characteristics of the data can contribute to making better scheduling decisions. For example, in a scenario where a task encounters multiple failed retries, leveraging metadata can guide the adoption of an alternative execution strategy to potentially secure a successful result.

In a data-centric setup, the scheduler needs to be aware of the nature of the job that is being executed. It can, for example, be annotated with some characteristics of the job that is to be executed. This gives the scheduler an idea of the kind of job that it is dealing with.

Understanding the unique characteristics of the data involved in an ETL job can be a difficult task. This complexity is mainly due to the inherent diversity and dynamicity of data sources, which may be structured or unstructured, static or real-time, and with different levels of data quality and completeness. [36]

One of the common illustrations of data-centric ETL scheduling is event-driven scheduling. This approach triggers the ETL process based on data-related events rather than relying on a pre-set timetable. An example might be a retail business implementing an ETL process that automatically runs when a new batch of transactional data is generated at the Point-Of-Sale (POS) systems. This enables real-time or near-real-time data integration, which is essential for operational reporting and decision-making [38, 39]

Altering existing job schedulers to accommodate data-centric features can be a challenging task. There might be technical difficulties in understanding the specific characteristics of each data element involved in an ETL workflow. Adding data-centric capabilities to these systems may require significant changes to the underlying architecture and coding, which can be time-consuming and costly.

By considering the characteristics of the data involved, data-centric schedulers can make more informed decisions about job scheduling [37,13]. For instance, jobs involving large data sets can be scheduled during periods of low system load, while jobs involving real-time data can be prioritized to ensure data freshness.

4.2 Benefits of the chosen system

Airflow strongly aligns with our listed functional needs, thus making it advantageous to extend this scheduler. If specific scenarios don't permit scheduler extension, it's possible to incorporate either a prescheduling or post-scheduling step to address this issue.

Building a scheduler from the ground up can be resource-intensive and time-consuming. Extending Airflow offers a more efficient solution, leveraging its existing capabilities while saving significant time and resources. It allows us to focus on customization based on our specific needs, rather than tackling the complex process of creating a brand-new scheduler.
As a battle-tested, open-source solution, Airflow has been adopted and improved by a large community of developers, and thus it has a robust, reliable infrastructure that we can depend on.

4.3 High-Level Functional and Non-functional Design Requirements

In this section, we delve into the high-level functional and non-functional requirements based on the criteria provided by our industry partner Data Lizard. Due to the reasons mentioned in preceding sections 3.5 & 4.2, we've resolved to advance our prototype implementation by extending the functionalities of Apache Airflow.

In a world where data volume is ever-increasing, the inability to scale can cripple a system, so all logic must be parallelized to facilitate scalable data integration. The system needs to provide a solution where tasks can run concurrently. This, for instance, could involve splitting a job into smaller tasks, each of which could be processed by a separate worker in the cluster, thus optimizing time and resource utilization.

An API is needed to access each task and job's internal state and trigger jobs asynchronously. Applications utilizing the scheduler require a well-structured interface with context or runtime IDs, allowing them to resume work after disconnection. This ensures streamlined operation and seamless interaction with the system, even after interruptions.

The next requirement is a user-friendly interface, ideally suited for defining tasks and workflows, with the capability to manage them during runtime. A web-based interface built with modern technologies can deliver the desired level of interactivity, accessibility, and flexibility. The interface should provide options for the manual start, stop, or restart of tasks, allowing for intervention when necessary.

Our proposed system must also support multi-tenancy. Given the potential of the system to be used across different domains and organizations, the ability to define and execute jobs per user is crucial. This involves proper isolation of data, processes, and configurations among tenants to ensure privacy and eliminate possible interference.

Job versioning is another requirement. This means there is a history of when each version was executed and when the version changed, and ideally some statistics. This allows users to trace back the performance and execution details of each version. For example, if the current version of a job is failing 90% of the time, compared to the 10% failure rate of the previous version, engineers could make informed decisions to revert back to the previous version.

The system must also include a failover mechanism to provide resilience against downtimes. If a machine fails during job execution, the system should be capable of automatically rescheduling the job to another available and capable machine. Furthermore, it's essential to consider whether there was any potential interference with writing on the target system and provide the ability to roll back changes in such scenarios.

The requirement for an event-based triggering mechanism, including time-based events like CRON syntax, must also be provided.

The scheduler should be cross-platform compatible, thereby facilitating its usage across different operating systems like Linux and Windows.

Lastly, the system should also have the ability to reference the transmitted data and metadata, especially timestamps, during rescheduling. This requirement allows an application using the scheduler to restrict data integration based on conditions, like for example not repeating integration with data older than a certain time. This way, the system can be configured to focus on the most relevant or fresh data, optimizing its performance.

These requirements, grounded in the practical challenges of modern data processing and management, ensure the Apache Airflow extension would be robust, efficient, user-friendly, and above all, highly adaptable to diverse operational contexts.

Now that we understand the requirements, the subsequent sections will focus on expanding the test bed to verify if the functional requirements are met.

5. Implementation

5.1 Validation of Functional Requirements

To assess the satisfaction of functional requirements, a comparison between the requirements and the existing state is necessary. The following table represents the current state of the functional requirements in Airflow.

Table 1: Overview of the functional requirements, their verification and current status

Requirement	Verification	Status
# Req. 1 Parallelization of Tasks	In Apache Airflow, tasks within a DAG can be executed in parallel, provided they have no explicit dependencies on each other. This enables you to break down your data integration logic into smaller tasks that can run concurrently, allowing for efficient utilization of resources and faster data processing. Since tasks in Apache Airflow can be executed on distributed systems like clusters, you can scale your data integration processes by adding more worker nodes. This allows you to handle larger workloads and process data more quickly. You can design your workflows to execute tasks in parallel by defining appropriate dependencies between tasks. This way, tasks that do not depend on each other can run concurrently, achieving parallelism and improving overall efficiency.	OK
# Req. 2 Robust API to query internal state of tasks	Apache Airflow provides an API that allows you to query the internal state of tasks and jobs, as well as trigger jobs asynchronously. The Airflow REST API enables you to interact with the Airflow system programmatically, giving you the ability to monitor task and job states, trigger executions, and perform other management tasks. With the Airflow API, you can: ○ **Query Task and Job States:** You can use the API to retrieve information about the current state of tasks and jobs within your workflows. This includes information about whether a task is running, succeeded, failed, or in other states. ○ **Trigger Jobs Asynchronously:** The API allows you to trigger the execution of workflows (DAGs) asynchronously. This means you can start a workflow run through the API without waiting for it to complete synchronously. ○ **Monitor Execution Progress:** The API provides endpoints to monitor the progress of workflow runs, including individual task states and overall run status. ○ **Retrieve Logs:** You can access logs generated by tasks through the API, enabling you to diagnose issues and analyze task execution details.	OK

	○ **Manage DAGs:** The API allows you to manage DAGs, including pausing and unpausing them, enabling or disabling them, and more. To interact with the Airflow API, you typically use HTTP requests to specific API endpoints. You can use tools like curl, programming languages like Python, or API clients to make these requests. The API responses are usually in JSON format, making it easy to parse and utilize the data. This is not an exhaustive list of functions, the API offers many other options as well.	
# Req. 3 User interface for task management	Apache Airflow provides a user interface for task management. The Airflow UI is a web-based dashboard that allows users to monitor, manage, and interact with their workflows (DAGs), tasks, and job executions. This UI makes it easy to visualize the state of your workflows, track task progress, and manage various aspects of your data pipelines. In the Airflow UI, you can: ○ **Monitor Workflow Runs:** The UI displays a list of your DAGs and their corresponding runs. You can see whether a run is queued, running, succeeded, failed, or in other states. ○ **View Task Status:** For each workflow run, you can view the status of individual tasks. This includes information about task states, execution times, and logs. ○ **Access Logs:** The UI provides access to logs generated by tasks, making it easier to troubleshoot issues and understand task behavior. ○ **Trigger Manual Runs:** You can trigger manual executions of workflows (DAG runs) through the UI, useful for testing or re-running specific tasks. ○ **Pause and Unpause DAGs:** You can pause or unpause entire DAGs through the UI. This is useful when you want to temporarily stop the execution of certain workflows. ○ **View DAG Graphs:** The UI can display a graphical representation of your DAGs, showing the task dependencies and execution flow. ○ **Configuration and Parameters:** You can set or modify parameters and configurations for DAGs and tasks through the UI. ○ **View Scheduler Information:** The UI can provide information about the Airflow scheduler, such as the last execution time and the next scheduled run. The Airflow UI enhances the user experience by providing an intuitive interface for managing tasks, workflows, and job executions. It's particularly helpful for monitoring ongoing data integration processes, identifying bottlenecks, and taking actions to ensure the smooth execution of tasks. This is not an exhaustive list of functions, the GUI offers many other options as well.	OK
	Although Airflow does not currently support multitenancy, it is currently being actively	Not fulfilled

# Req. 4 The system must support multiple tenants. Jobs are defined and executed per user.	developed and a first draft of the feature is likely due for Airflow 3.0. Here are some links with the discussions around implementation of multitenancy at Airflow: **Multitenancy @ Airflow** **Extensible user management @ Airflow** Apache Airflow does not natively support multi-tenancy in the way some traditional multi-tenant applications do. However, organizations have been able to implement a form of multi-tenancy using a combination of namespaces, role-based access controls, and other mechanisms. Here are a few ways to achieve this: ○ **DAG-level Access Control:** Airflow supports DAG-level access, meaning you can control which users or roles have access to which DAGs. This allows different teams to have their own set of DAGs that other teams cannot access or modify. ○ **Pools:** Airflow supports the concept of "pools" that can be used to limit the parallel execution of Tasks. By assigning specific teams or users to specific pools, you can somewhat isolate the resources they use. ○ **Connections & Variables:** Connections and Variables in Airflow can be permissioned as well, allowing admins to specify which users or roles can view or edit them. ○ **Multiple Airflow Instances:** While not truly multi-tenancy, some organizations choose to run separate Airflow instances for different teams or departments. With container orchestration platforms like Kubernetes, it's easier to spin up isolated Airflow instances for different teams. This approach guarantees complete isolation but at the cost of overhead and potential redundancy. In light of the information we currently possess, it might be worth waiting for Apache Airflow to release the multitenancy feature by itself as the other approaches mentioned does not provide full isolation or has overheads.	yet, but is being actively developed by the Airflow community
# Req. 5 Job versioning: History of when each version was executed, changed, some statistics.	Job versioning feature of Airflow is not yet implemented, but it is on the roadmap. Airflow users over the years have found some workarounds to get around this limitation, some of which have been discussed in the next section.	Not fulfilled yet, methods to achieve it discussed in next section
# Req. 6	This feature has already been demonstrated with our test scenarios in the section above. Airflow's failover mechanism ensures uninterrupted workflows by	OK

Failover mechanism	leveraging retries, distributed execution with Celery or Kubernetes, and metadata database redundancy, enabling recovery from task failures and node outages. Airflow provides failover capabilities via retries, distributed execution, and database redundancy. The failover mechanism ensures that if a primary node fails, task executions are automatically shifted to a standby node, providing uninterrupted workflow execution and enhancing system reliability.	
# Req. 7 An event-based triggering mechanism including time (e.g. cron syntax) must be supported.	Airflow offers versatile triggering mechanisms, including time-based scheduling via cron syntax and external triggers for event-based runs through its API or CLI. It incorporates sensors, which await specific events, and the TriggerDagRunOperator for event-driven DAG runs. Airflow also introduced data-driven scheduling with "Dataset," optimizing task dependencies. Additionally, deferrable operators allow task delays based on conditions.	OK
# Req. 8 Cross-platform compatibility, hence facilitating its usage across different operating systems like Linux and Windows.	Apache Airflow is cross-platform compatible, which means it can run on different operating systems. Apache Airflow is written in Python, and as long as you have Python and the necessary libraries installed, you can run Airflow on various platforms. It is also compatible with a variety of databases, including MySQL, PostgreSQL, and Amazon Redshift. This makes it a versatile tool that can be used in a wide range of environments. Airflow can also be deployed on cloud platforms, such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). Airflow is developed in Python, which is inherently cross-platform. This means wherever Python runs, theoretically, Airflow can run too. Many organizations and developers run Airflow within Docker containers. Docker offers cross-platform compatibility because it abstracts the underlying operating system.	OK
# Req. 9 HTTP APIs should be provided for integration into Applications	Apache Airflow provides HTTP APIs, which means it can be integrated with applications, including those written in Java, using the Airflow REST API. With the release of Apache Airflow 2.0+, a stable REST API was introduced which allows users to perform CRUD operations (Create, Read, Update, Delete) on many of Airflow's core	OK

	resources. This can be consumed by Java or any other language that can make HTTP requests.	
# Req. 10 Ability to refer to the transmitted data and metadata in rescheduling instructions, in particular timestamps.	Airflow's primary mechanism for passing messages and metadata between tasks is through XComs (short for cross-communication). XComs, or cross-communication, is a mechanism in Airflow that allows tasks to communicate with each other. XComs can be used to pass data between tasks, or to share information such as the status of a task. Tasks can push and pull data to and from XComs, which means you can use XComs to store and retrieve any metadata or data that you want to reference in later tasks or even in later runs of the same DAG. XComs are stored in the Airflow metadata database and are accessible to all tasks in the same DAG. They are identified by a key, which is essentially the name of the XCom, as well as the task_id and dag_id that it came from. XComs can be used in a variety of ways, such as: ○ Passing data from one task to another ○ Sharing information about the status of a task ○ Logging errors or warnings ○ Triggering other tasks XComs are a powerful tool that can be used to improve the flexibility and scalability of your Airflow workflows. However, it is important to note that XComs can also be a source of performance bottlenecks, so it is important to use them wisely.	OK

After analyzing the delta between the functional requirements and the current state of the system, as a next step, we zoom into the functional requirements to get a clearer view of the existing state and propose extensions to bridge the gap between the requirements and the existing system. This approach ensures that any necessary modifications or enhancements are made to meet the desired functional requirements.

5.2 Zooming In: A Closer look at some of the Requirement Specifics

5.2.1 Parallelism Factors

Airflow can support truly parallel execution, especially with distributed executors like Celery Executor or Kubernetes Executor. Tasks within an Airflow DAG can be executed in parallel, provided they have no explicit dependencies on each other. This enables the breakdown of data integration logic into smaller tasks that run concurrently, allowing for efficient utilization of resources and faster data processing.

However, the degree of parallelism is influenced by configuration, the chosen executor, DAG design, and external factors. If maximizing parallelism is a goal, we need to carefully configure and tune the Airflow setup and be mindful of external bottlenecks.

```
# The executor class that airflow should use. Choices include
# ``SequentialExecutor``, ``LocalExecutor``, ``CeleryExecutor``, ``DaskExecutor``,
# ``KubernetesExecutor``, ``CeleryKubernetesExecutor`` or the
# full import path to the class when using a custom executor.
executor = CeleryExecutor

# This defines the maximum number of task instances that can run concurrently per scheduler in
# Airflow, regardless of the worker count. Generally this value, multiplied by the number of
# schedulers in your cluster, is the maximum number of task instances with the running
# state in the metadata database.
parallelism = 32

# The maximum number of task instances allowed to run concurrently in each DAG. To calculate
# the number of tasks that is running concurrently for a DAG, add up the number of running
# tasks for all DAG runs of the DAG. This is configurable at the DAG level with ``max_active_tasks``,
# which is defaulted as ``max_active_tasks_per_dag``.
#
# An example scenario when this would be useful is when you want to stop a new dag with an early
# start date from stealing all the executor slots in a cluster.
max_active_tasks_per_dag = 16
```

Figure 5.1: Some Configuration factors affecting the degree of parallelism in Airflow [Screenshot]

GANTT Chart of an Airflow DAG using Celery Executor run with 10 parallel tasks executed with 1, 2 and, 3 Celery workers. From these charts, it is clear that the one of the factors affecting the parallelism capability of airflow is the amount of resources.

Figure 5.2: Gantt Chart depicting the resource to parallelism bottleneck [Screenshot]

5.2.2 Trialing the Failover Mechanism

Airflow provides failover capabilities via retries, distributed execution, and database redundancy. The failover mechanism ensures that if a primary node fails, task executions are automatically shifted to a standby node, providing uninterrupted workflow execution and enhancing system reliability.

In our experiment, we established a setup with three distributed Celery workers that were all online. We accessed the workers via their shells to simulate an outage for two of the three workers. After initiating the Directed Acyclic Graph (DAG) and verifying task allocation to all workers, we mimicked the outage by terminating two workers. Upon doing so, we observed that the tasks were seamlessly redistributed to the remaining worker, that completed them successfully. This indicated that the failure of two workers did not impede the tasks, showcasing the effective failover capabilities of the system.

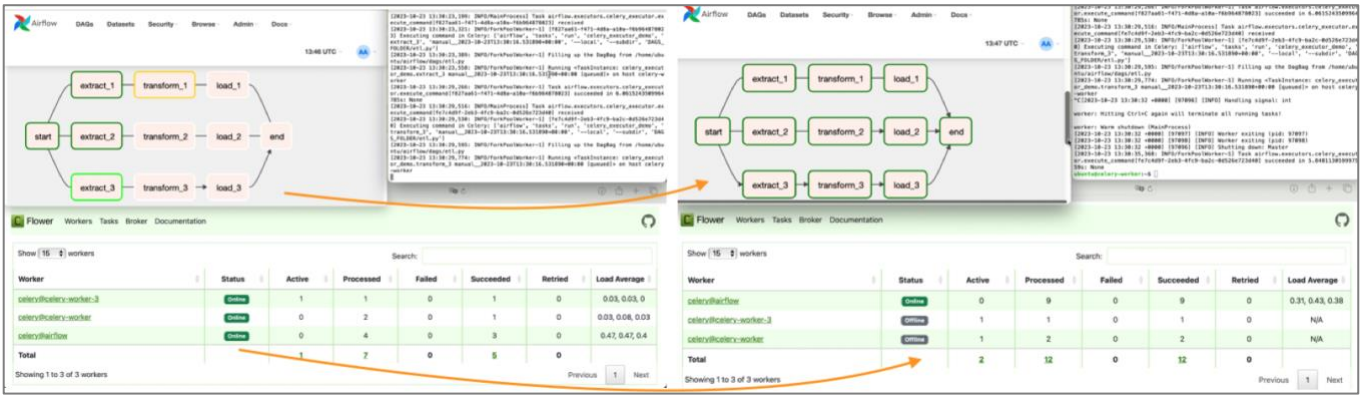

Figure 5.3: Execution scenario depicting the failover distribution. Successful execution despite some terminated workers. Integration Tools [Screenshot]

5.2.3 Experimenting with the Airflow API

We experimented with the Airflow API v2.7.1 to explore its functionality and applications. The API is well-documented and follows RESTful principles, making interaction with the Airflow application straightforward.

Our experimentation involved the creation of Directed Acyclic Graphs (DAGs), tasks, and the monitoring of their execution. The API allowed us to programmatically create workflow structures and observe their execution, offering insights into task dependencies, timings, and statuses.

Airflow supports many authentication methods, and it is even possible to add your own method. For our experiment, we used to authenticate to the Airflow API using basic authentication, where we needed to provide the username and password in the HTTP Authorization header of the requests.

In assessing the functionalities of the Airflow API, a demonstrative DAG "demo_api_usage.py" was constructed, and the collection was subsequently imported into Postman for experimental purposes. The Postman collection was used, the API specification was downloaded and the demo DAG has been uploaded to GitHub. The latest stable version of

the API can be retrieved from here: Airflow API. At the time of writing the latest stable API was Airflow API (Stable) (2.7.1) which was also the version used in our experiments.

The API performed up to expectations, yielding comprehensive responses. Significantly, the API emitted resilient error messages, facilitating swift debugging. Most of the endpoints accept JSON as input and return JSON responses. The platform also supports **CRUD** operations on most resources and also supports the monitoring of connections to resources. Below are a few responses from queries directed to the API via Postman:

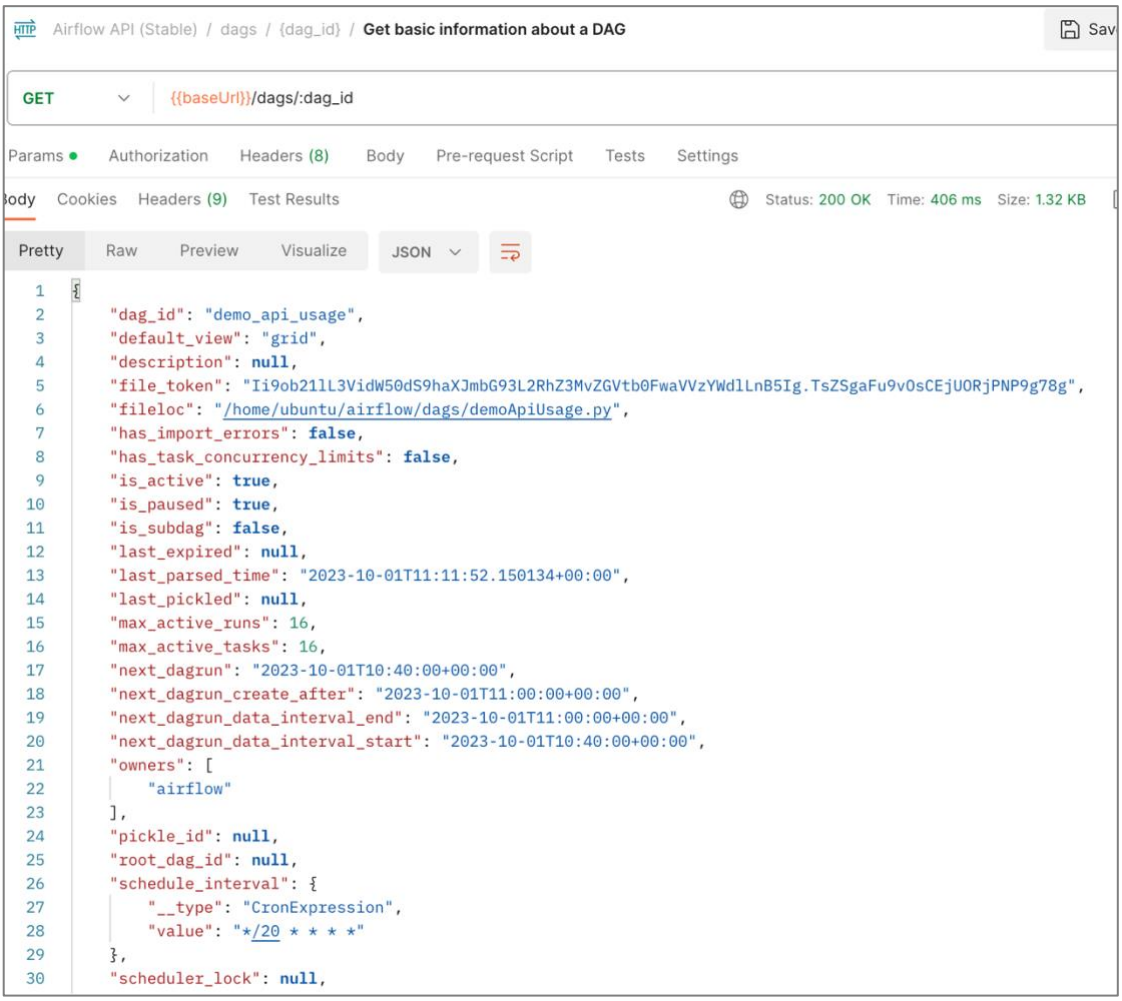

Figure 5.4: API response for call executed to get the details of a Task [Screenshot]

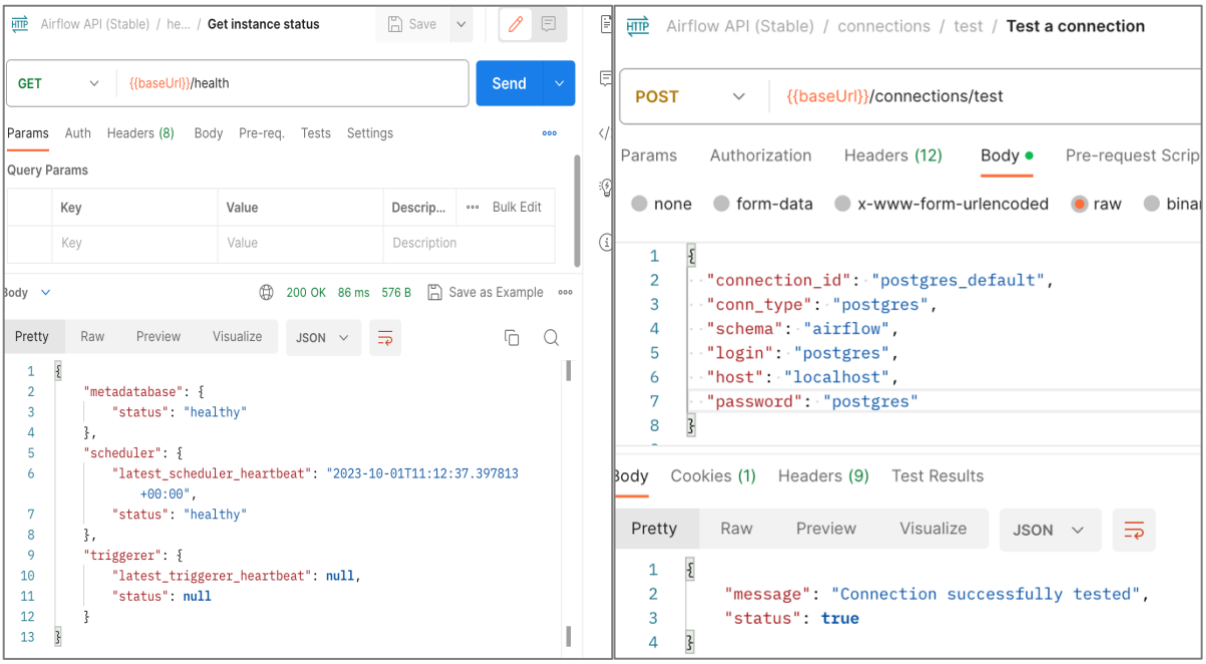

Figure 5.5: API responses for administrative tasks to check Instance's health or test a Database connection [Screenshot]

The API adheres to the error response format outlined in RFC 7807, or "Problem Details for HTTP APIs". Like the standard API responses, clients must be equipped to efficiently manage additional response members. This approach necessitates the implementation of robust client-side mechanisms to seamlessly interact with the diverse and potential supplementary elements in the API's responses. Here are some error messages provoked by wrong queries to the API:

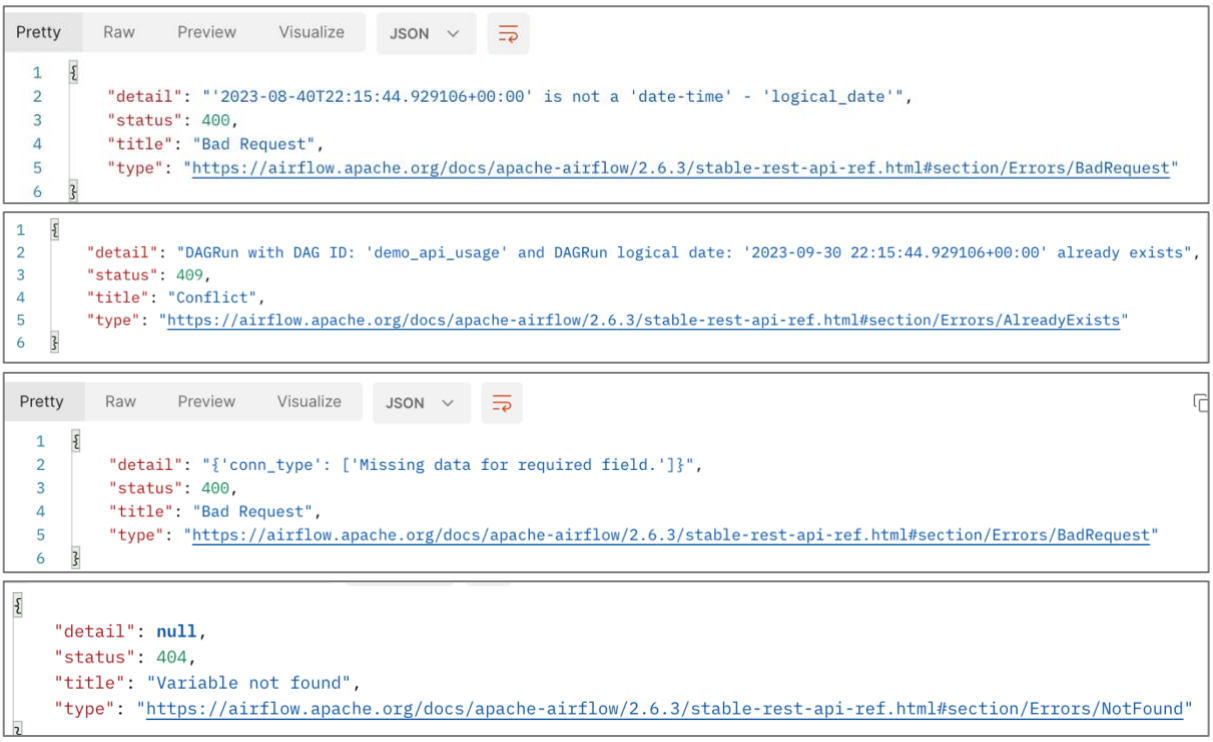

Figure 5.6: API Error handling responses on failed calls showing useful error messages [Screenshot]

For an enhanced visualization and understanding of the API's specifications, they were also imported into Swagger. This enabled a clearer, more detailed view of the specifications, providing insights into the extent of the API's utility for executing tasks through client applications, as illustrated by the subsequent screenshots.

In the screenshot below, the array of potential operations that can be executed is displayed, detailing mechanisms to both retrieve information about and initiate the DAGRun within the Airflow environment. This representation provides visual insights into the actionable procedures available for interacting with and managing DAGRun processes.

Figure 5.7: Overview of the different operations available to control Task executions [Screenshot]

We see from the below Swagger UI screenshot that the Airflow API provides endpoints to manage user roles within the Airflow system as well. The screenshot depicts the diverse set of operations and actions available, allowing for comprehensive management and manipulation of user roles, including creation, modification, and deletion, thus offering detailed insights into user role governance via programmatic interfaces within the Airflow environment.

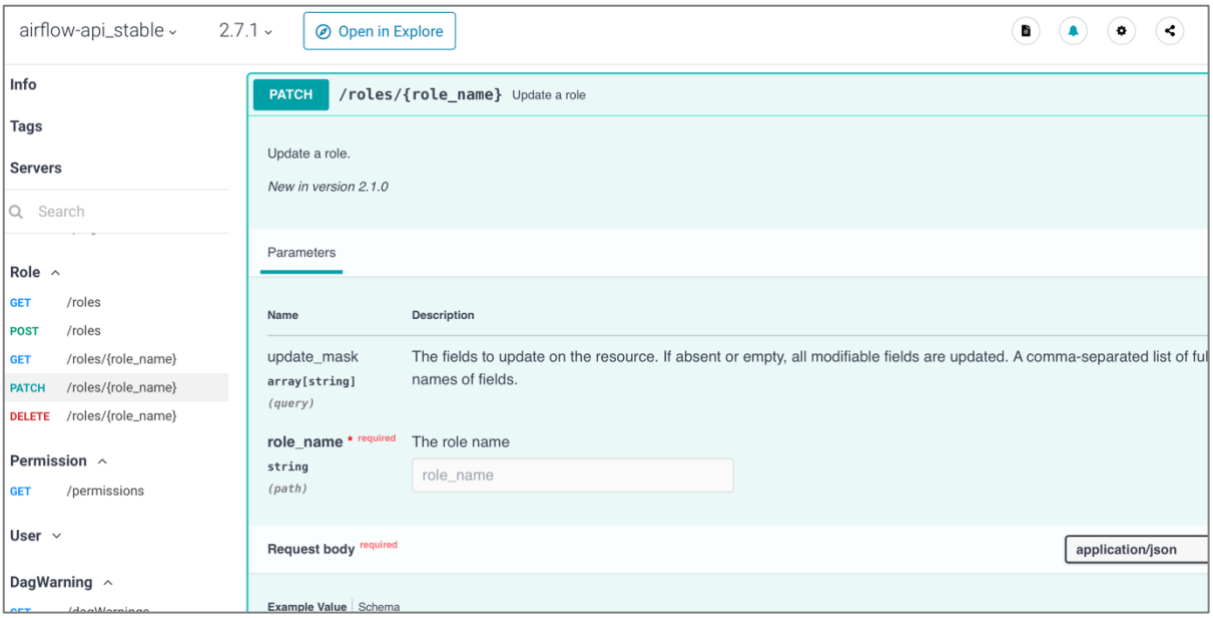

Figure 5.8: Overview of some of the operations available to govern Users and Roles [Screenshot]

The API also furnishes endpoints facilitating the monitoring of events, import errors, and system health. These endpoints enable the users to systematically observe and manage eventless states, track importation errors, and monitor the overall health and performance of the system. Similarly, here are some operations that are available to manipulate Global Variable and XComs in Airflow via the API:

Figure 5.9: Overview of some of the operations available to govern Users/Roles & to control Variable/XComs [Screenshot]

The API proved to be efficient and adaptable, allowing seamless integration with different workflows. The capability to manipulate DAGs and tasks programmatically provided flexibility in handling dynamic workflow requirements. The API responses were quick and accurate, contributing to effective monitoring and management of the workflow processes.

The API's adherence to RESTful principles facilitated easy interactions, with structured JSON responses allowing for convenient parsing and processing of the acquired data. However, thorough knowledge of Airflow's concepts, like DAGs and operators, is imperative for effective utilization of the API.

5.2.4 Experimenting with the Airflow UI

The Airflow UI is instrumental for managing and debugging data workflows, providing several views to efficiently monitor and operate on Directed Acyclic Graphs (DAGs). Below is a succinct overview of some of these views and their functionalities:

- **DAGs View:** Lists all DAGs and provides quick access to useful pages. Enables filtering of DAGs via tags and viewing task statuses. Offers the option to hide completed tasks.

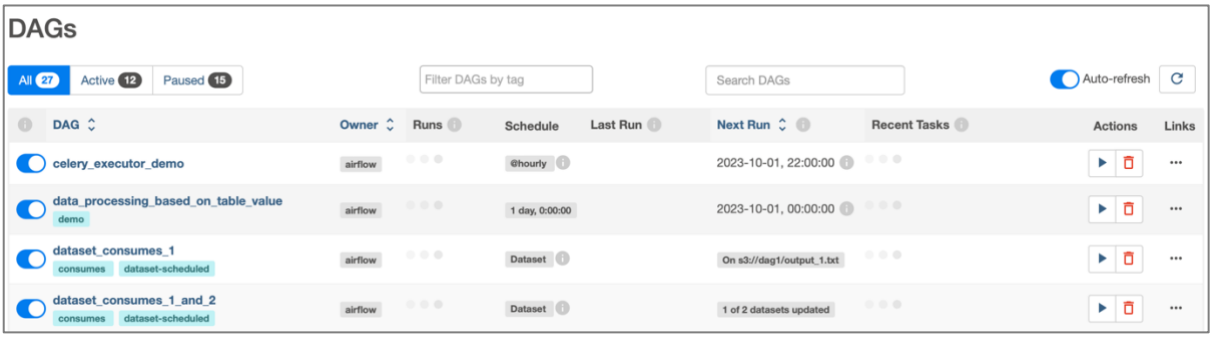

Figure 5.10: A dashboard with quick view of all DAGs and their details including execution statuses [Screenshot]

- **Datasets View**: Illustrates current datasets and their interaction with DAGs, allowing users to visualize dataset relationships and the task history affecting them.

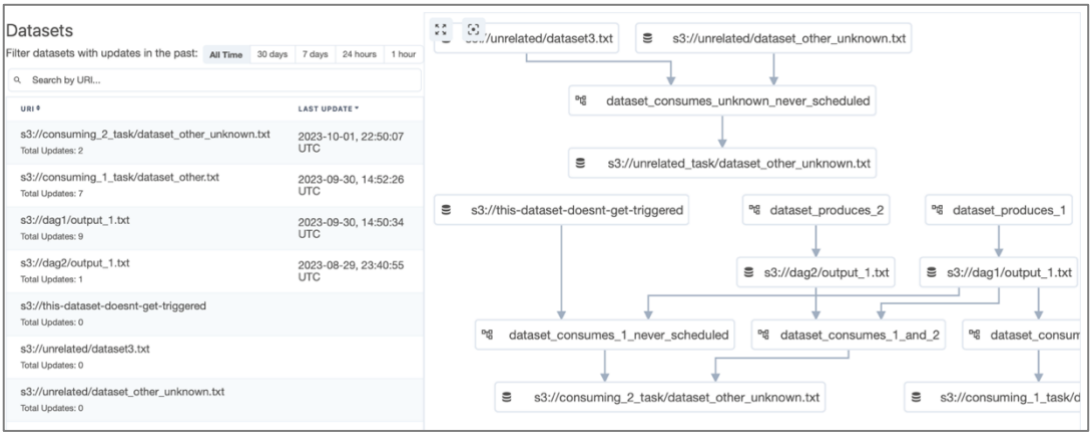

Figure 5.11: View showing how Datasets in Airflow are linked to each other [Screenshot]

- **Grid View:** Presents a time-spanning bar chart and grid representation of DAGs, allowing quick identification of pipeline blockages and delays.

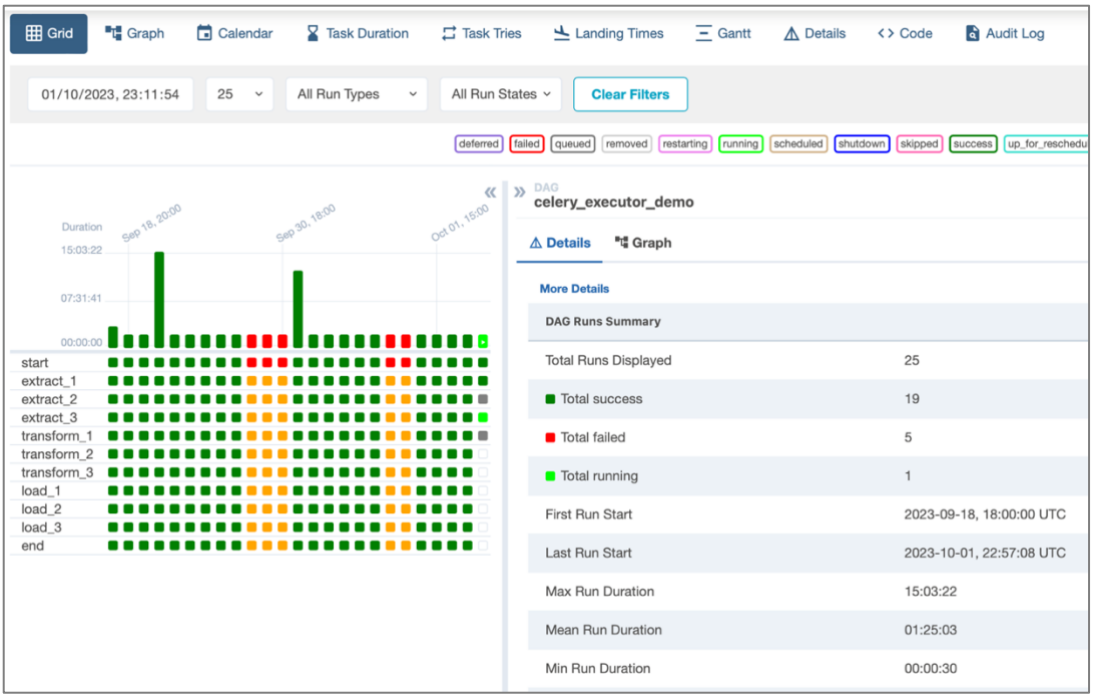

Figure 5.12: On current & history executions of a DAG, the status of each task is shown in real time [Screenshot]

- **Graph View**: Offers comprehensive visualization of DAG dependencies and statuses for specific runs.

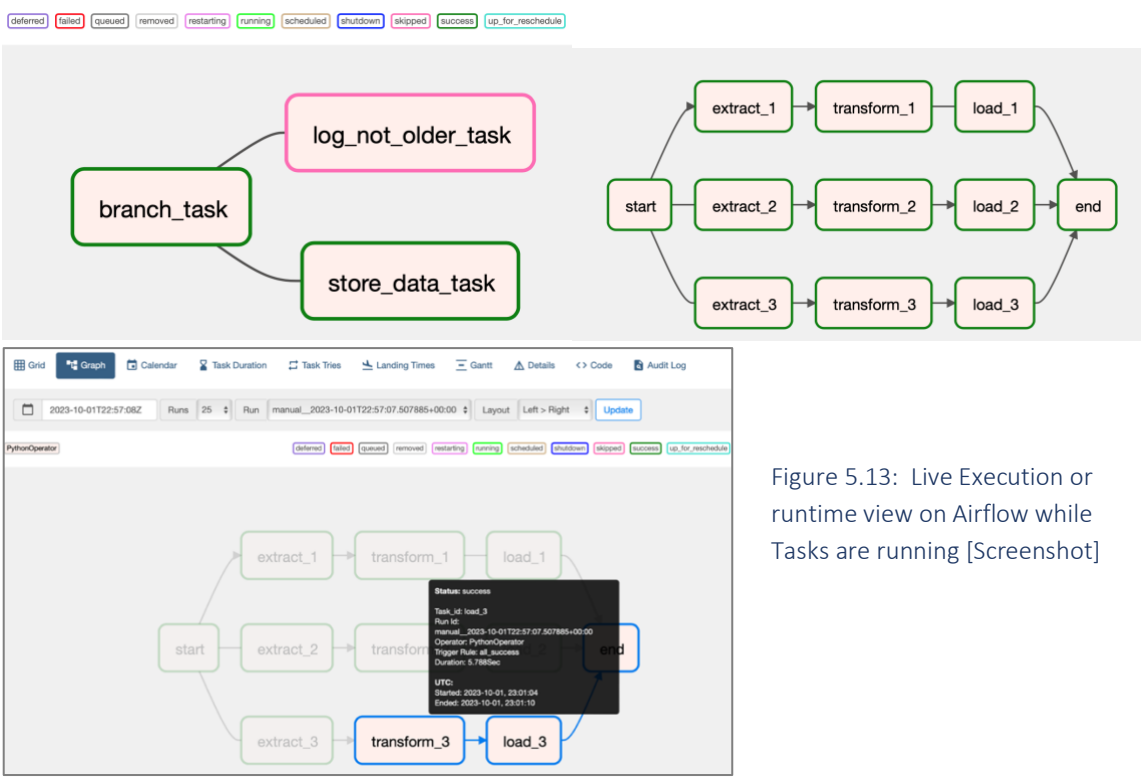

Figure 5.13: Live Execution or runtime view on Airflow while Tasks are running [Screenshot]

- **Calendar View:** Provides an overview of DAG history over extended periods, highlighting trends in run success/failure rates

Figure 5.14: Calendar view depicting when the next executions of the DAG are planned including a bird's eye view of the previous success/failure statuses [Screenshot]

- **Gantt Chart:** Aids in analyzing task duration and overlap, identifying bottlenecks, and analyzing time consumption for specific DAG runs.

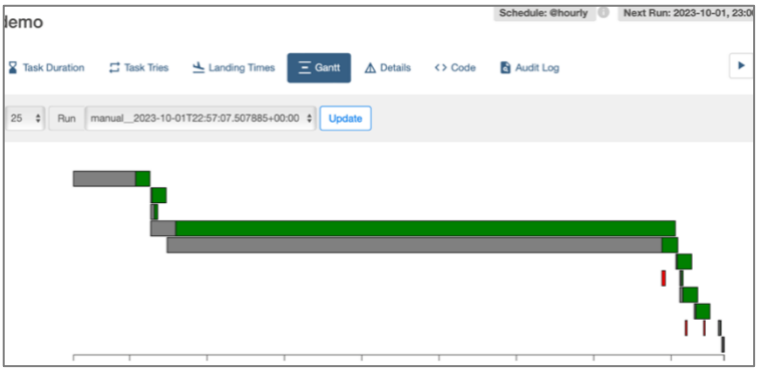

Figure 5.15: Gantt chart view help us in recognizing any potential bottlenecks that might have been introduced over the course of executions [Screenshot]

- **Code View:** Enhances transparency by providing quick access to the source code generating the DAG.

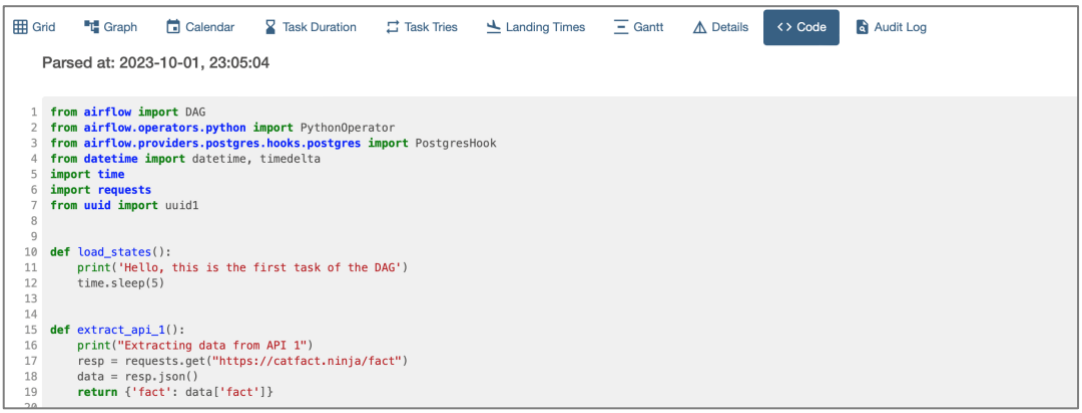

Figure 5.16: An extremely practical outlook that helps us check and view the source code [Screenshot]

Although this is not an exhaustive collection of all the views available to the consumer, it gives an idea of the variety available. These views collectively contribute to simplified navigation and operation of DAGs from the UI, fostering an efficient and user-friendly environment for managing data pipelines.

5.2.5 What will multi-tenancy @ Airflow look like?

Multitenancy is a feature that is currently not natively available among Airflow's offerings but one that is being very actively developed. Recent talks at the Airflow Summit in September 2023 suggest that the solution is comprehensive and the rollout is on the horizon

Development: GitHub Link
Board: Multitenancy @ Airflow.

Airflow's present architecture integrates user management within its core, utilizing Flask-AppBuilder (FAB) to manage users, roles, and permissions. This integration implies that each augmentation to user management demands core Airflow alterations. Given the broad spectrum of user needs, from individual users to expansive corporate teams, achieving a one-size-fits-all solution is challenging hence an idea emerged: relocating user management from the core to a novel Airflow component— the auth manager. This proposed manager would harbor a generic interface, charting common API/functions, making user management in Airflow adaptable and expandable.

The envisioned proposal involves removing user management from Airflow's core and introducing the auth manager. This manager's mission is to oversee functionalities and resources linked to users, roles, and permissions. Users would be free to opt between a simplistic authentication manager and a sophisticated version, recognizing groups or tenants. The manager's interface would be the sole integration point with core Airflow, and "Security" on the navigation bar would be configured individually by each manager. These managers are designed to be "pluggable", allowing users to interchange them per their needs.

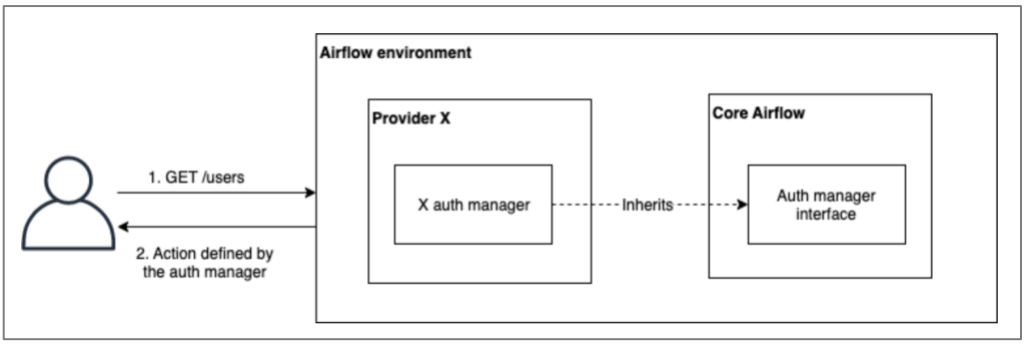

Figure 5.17: Simple representation of the core components envisioned to implement Multitenancy. [Source: Taken from Airflow Wiki - AIP-56 Extensible user management (https://cwiki.apache.org/confluence/display/AIRFLOW/AIP-56+Extensible+user+management)]

Considerations:

Problem Resolution: This strategy renders Airflow's user management adaptable, facilitating a more advanced user management mechanism than the current system. It bridges the existing gap by enabling providers to support user management natively.

Necessity: The heterogeneity of user requirements makes a universal user management system unattainable. The proposition is to provide users with an adaptable system to shape their user management. Additionally, the integration of native user management in cloud platforms will seamlessly map roles to identity providers, addressing current Role-Based Access Control (RBAC) challenges in diverse cloud environments.

Affected users: Every user will experience this change. By default, the transition is seamless, using the backward-compatible FAB auth manager. However, the experience may vary if administrators opt for different managers.

Completion Criteria: The realization of this proposal involves defining the auth manager interface, instituting a new FAB provider, and integrating the FAB auth manager as part of this new provider.

5.2.6 How can we implement a Job versioning structure?

Current Status:
Airflow's current deployment design tends to overwrite existing DAG files, and no API exists for their modification. This leads to several complications.

- New deployments overwrite old DAG files, making version tracking challenging.
- There's no trace of the old DAG version in the user interface.

Discussion:
Airflow does not currently support complete DAG versioning, although there is a draft proposal, AIP-36, on the subject. It's recommended that *DAGs should evolve slowly, primarily in additive ways. Significant structural changes in a DAG should prompt the creation of a new DAG to avoid complications*. A straightforward solution to track versions would be to embed a version number within the dag_id. Alternatively, using tags might visually appeal to the user interface, though the primary design of tags is for filtering, potentially leading to some issues.

The DAG Versioning AIP (AIP-36) author mentioned that while there were initial plans to implement Webserver-only DAG Versioning to enhance visibility without affecting execution, the scope now extends to full end-to-end DAG Versioning. This means Airflow aims to rectify the current problem where altering a task (adding/removing) reflects across all prior Dag runs in the Webserver. The future plan is to incorporate both Remote DAG Fetcher and DAG

Versioning, enabling DAG versioning on the worker side. This would allow users to run older versions of a DAG. The implementation timeframe is projected to be around 2023/24.

Workaround suggestions to implement job versioning in Airflow:

- Add a version suffix to filenames in the DAGs directory to prevent new deployments from overwriting old ones. Ensure the DAG file name corresponds to the DAG ID, which includes the version number. Appending the version suffix to DAG IDs ensures traceability when you view the DAG in the Airflow UI. The DAG file name matches the DAG ID from the content of the DAG itself (including the DAG version).
  E.g., for a DAG with id my_unique_dag-v1.0.9, the file name would be my_unique_dag- v1.0.9.py.

- Manipulate runtime environment variables to influence running DAGs. The start and end times of DAGs can be adjusted by using environment variables appended with version suffixes. When deploying a new version, the end time for the old version and the start time for the new one can be altered, ensuring a smooth transition.

- Version Control with Git:
  Store all DAG files in a Git repository. Utilize a Continuous Integration pipeline (like Atlassian Bamboo, Jenkins, Circle CI, etc.) to initiate a new build for every merge request to the master branch. The build process then packages the DAG files into a zip. The deployment script (shell) unzips the package and places the DAG files on the Airflow server in the DAG folder.

- Separate DAG files for separate environments (Production/UAT/DEV) are also a good option for maintaining order

- Many companies have found different ways of working around this Airflow limitation which will hopefully be addressed soon. One of the different approaches discussed involves placing each DAG in its own separate repository. In this approach, to track which DAG version to capture while building the Airflow image in the Airflow repository, they used GIT submodules where each DAG repository is being served as a submodule to the Airflow repository. In this way, they could maintain a specific state of the DAG by referring to its master, working, or commit branch.

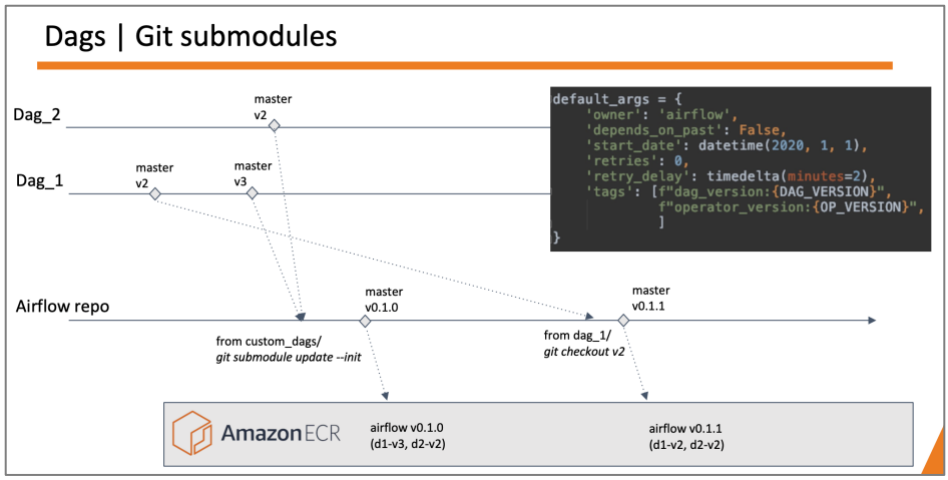

Figure 5.18: Job versioning approach using GIT submodule where each DAG repo is being served as a submodule to the Airflow repository.[Source: Figure taken from Airflow Summit Talk, (https://airflowsummit.org/slides/2022/a3-ManagingScale-Anum.pdf)]

Benefits:

- Version Visibility: The inclusion of a version in the DAG file name ensures that older versions aren't overwritten, making it easy to revert if needed.
- Easy Identification: As the version number is included, the newly loaded DAG is instantly recognizable in the Airflow UI.
- Automation: As the DAG file name matches the DAG ID, one can enhance the deployment script to use Airflow commands to automatically switch on / activate new DAGs post-deployment.
- Historicization: Storing every DAG version in Git allows for easy retrieval and rollback to previous versions when necessary.

5.2.7 Exploring Types of Distributed Executors in Apache Airflow

Apache Airflow's Executors are pivotal components ensuring efficient task management and execution, with each offering unique attributes and optimal use cases.

1. Local Executor:
This is the default executor in Airflow, designed for single-node deployments, executing tasks on the same machine as the Airflow scheduler, and employing multiprocessing to enhance performance. It is suitable for small to medium-sized workflows that don't necessitate distributed execution or elaborate scaling, and it is easy to set up without any additional configuration, making it ideal for simpler ETL jobs.

2. Celery Executor:
Optimal for distributed Airflow deployments, it leverages the Celery distributed task queue to run tasks across multiple nodes, allowing for horizontal scaling and enhanced performance

for larger workflows. It uses a message-queueing system to delegate tasks to independent workers, ensuring high availability and adaptability, with RabbitMQ and Redis as underlying support databases. This executor is suited for workflows necessitating distributed execution, complex parallelism, and scalability. It is suitable for running time-sensitive DAGs in production due to its high availability and scalability. However, it demands additional setup and a separate Celery installation.

3. Kubernetes Executor:
This newer executor runs tasks as Kubernetes pods, ideal for workflows that demand dynamic resource allocation in Kubernetes environments, providing scalability and flexibility. It suits workflows designated for Kubernetes clusters and those requiring interaction with other Kubernetes resources. However, setting it up necessitates an understanding of Kubernetes concepts and additional configuration.

4. Dask Executor:
An experimental option, it utilizes the Dask distributed computing framework, offering flexibility and scalability in parallel computing for workflows requiring distributed execution and fine-grained control over resource allocation. It is especially beneficial for workflows already leveraging Dask for parallel computing, but like others, it requires additional setup and a separate Dask installation.

To summarize the executors:
Local Executor: Best for development, easy to set up but less scalable. LocalExecutor suffices for smaller, simpler workflows
Celery Executor: Ideal for production with high availability and scalability, but complex to set up. Celery and Kubernetes Executors are preferable for extensive, distributed workflows
Kubernetes Executor: Offers dynamic scaling and task-level configurations with efficient resource utilization, but requires Kubernetes knowledge. Particularly suited for Kubernetes environments and dynamic resource allocation.
Dask Executor: Ideal for advanced, distributed computing needs; requires a nuanced understanding of its implementation. DaskExecutor aligns well with workflows utilizing Dask for parallel computing.

5.2.8 Triggering Mechanisms

Airflow supports a vast array of triggering mechanisms, including Cron and Event-based triggering mechanisms. Some of the triggering mechanisms include:

- **Time-based Scheduling with Cron Syntax:** Classic Scheduling approach. DAGs in Airflow can be scheduled using Cron-like expressions, allowing for fine-grained control over when tasks are run.

- **External Triggers:** DAGs can also be triggered externally through Airflow's API or CLI, enabling event-based runs outside of the regular schedule.

- **Sensors:** Sensors are Airflow operators that wait for a certain event to occur before they trigger a DAG. For example, you could create a sensor that waits for a new file to be created in a directory. Once the file is created, the sensor will trigger the DAG to run.

```
Some Sample Sensors in Airflow
#  [example_time_delta_sensor]
T0 = TimeDeltaSensor(task_id="wait_some_seconds", delta=timedelta(seconds=2))
# [example_bash_sensors]
T1 = BashSensor(task_id="Sensor_succeeds", bash_command="exit 0")
# [example_file_sensor]
T2 = FileSensor(task_id="wait_for_file", filepath="/tmp/temporary_file_for_testing")
# [example_python_sensors]
T3 = PythonSensor(task_id="success_sensor_python", python_callable=success_callable)
```

Figure 5.19: Examples of various sensors available to implement event-based handling [Screenshot]

- **TriggerDagRunOperator:** This operator allows you to trigger a DAG run based on a specific event, such as a message being published to a queue or a database record being updated.

```
Triggering with TriggerDagRunOperator
with DAG(
    dag_id="example_trigger_controller_dag",
    start_date=pendulum.datetime(2021, 1, 1, tz="UTC"),
    schedule="@once",
    tags=["example"],
) as dag:
    trigger = TriggerDagRunOperator(
        task_id="test_trigger_dagrun",
        # Ensure this equals the dag_id of the DAG to trigger
        trigger_dag_id="example_trigger_target_dag",
        conf={"message": "Hello World"},
    )
```

Figure 5.20: Excerpt of how Triggers can be used in Airflow to initiate downstream tasks [Screenshot]

- **Datasets:** The newest scheduling mechanism from Airflow, first introduced in version 2.4 presents data-driven scheduling through a new object type, Dataset, enabling built-in logic to manage data-dependent tasks, thus making Airflow data-aware & extending its scheduling capabilities beyond time-based methods like cron.

A Dataset is an output from one or more 'producer' tasks serving as a data source for downstream DAGs or 'consumers'. This feature ensures that consumer DAGs run only when the datasets they rely on are updated. For instance, a data analytics DAG will only execute once the data engineering team's DAG, creating a dataset, has finished running, solving common dependency issues efficiently.

```
example_dataset = Dataset("s3://dataset/example.csv")

with DAG(dag_id="producer", ...):
    BashOperator(task_id="producer", outlets=[example_dataset], ...)

with DAG(dag_id="consumer", schedule=[example_dataset], ...):
```

Figure 5.21: Producer/Consumer format of Datasets that form the base of Data Driven Scheduling [Screenshot]

- **Deferrable Operators:** Deferrable operators are operators that can be delayed until a certain condition is met. For example, you could create a deferrable operator that waits for a certain amount of time to pass before it triggers a DAG run.

5.2.9 Choosing the Best Airflow Installation Method for Your Needs

While Airflow is primarily developed and used in Unix-like environments, particularly Linux, it has a few elements that make it cross-platform compatible:

Language: Apache Airflow is written in Python, a platform-agnostic language. Python is known for its capability to run seamlessly on various operating systems like Windows, macOS, and Linux. So, the core logic and functionalities of Airflow can be executed on these platforms without significant modifications.

Database Support: Airflow uses databases to maintain and track the state of tasks. It supports several databases like PostgreSQL, MySQL, and SQLite. All these databases have versions for multiple platforms, ensuring that the metadata storage aspect of Airflow is cross-platform.

Docker & Kubernetes: For users who might find native installation challenging on platforms like Windows, Docker offers a solution. Airflow can be containerized using Docker, which provides an isolated environment for running applications across various platforms. Additionally, with KubernetesExecutor, Airflow can be scaled and operated in Kubernetes environments.

Web-Based UI: Airflow's interface is web-based. No matter what operating system you're on, if you can run a browser, you can access Airflow's dashboard. This ensures that user interactions remain consistent across platforms.

However, some challenges and nuances arise when using Airflow in non-Linux environments:

System-specific Packages: Some Airflow operators and hooks are designed for specific systems. For example, the SSHOperator or Bash-related tasks might pose challenges on Windows due to their Unix-centric design.
File System Paths: The way file paths are handled can differ between Unix-like systems and Windows. Care must be taken when defining file paths in DAGs to ensure compatibility.
Installation: Installing Airflow natively on Windows can be more challenging compared to Linux or macOS. However, this is mitigated with containerization solutions like Docker.

In conclusion, while Apache Airflow can technically be considered cross-platform due to its Python foundation and web-based UI, achieving smooth functionality, especially on Windows, may require some additional steps or adjustments. Nevertheless, with container solutions and an active community, many of these challenges are consistently addressed.

The Airflow documentation provides various methods to install Airflow, catering to different user requirements:

- **Using Released Sources:** Ideal for users who value software integrity and prefer installing from official Apache Software Foundation sources.
- **Using PyPI:** Enables installation via pip with constraints, ideal for users familiar with Python environments. Software is pre-built, eliminating the need for building from sources.
- **Using Production Docker Images:** Suited for users proficient with the Docker stack. It uses official Docker images from Airflow, ensuring isolated and standardized setups.
- **Using the Official Airflow Helm Chart**: Aimed at Kubernetes users, this method leverages a Helm chart, facilitating easy maintenance and configurations using official Docker images.
- **Using Managed Airflow Services:** Offers a hassle-free experience for users preferring third-party managed installations.

In essence, Airflow provides a spectrum of installation choices, from hands-on approaches to turnkey solutions, catering to various expertise levels.

5.2.10 What are the ways we can transmit Data to DAGs in Airflow?

Exchanging data between tasks is a prevalent need in Airflow. When composing DAGs, it's advisable to fragment them into smaller tasks for efficient debugging and swift recovery from failures, a practice well-known amongst those who write DAGs. Here are some methods to manage when a downstream task needs metadata from an upstream task or needs to process the results of its preceding task.

➔ Different Ways of Passing Arguments into Airflow DAGs:

1. **Default_args:** Define a dictionary of default arguments (`default_args`) during DAG initialization. These default parameters get automatically applied to all tasks within that DAG.

2. **Intermediary Data Storage:** Instead of relying on XCom, which is ideal for minimal data, leverage intermediary storage systems for larger data chunks. Store your data in platforms external to Airflow, such as S3, GCS, Azure Blob Storage, or databases, and fetch it in subsequent tasks. However, remember that Airflow's primary function is orchestration; for substantial data, employ processing frameworks like Spark or data warehouses like Snowflake or dbt.

3. **Managing Variables**: Variables in Airflow provide a convenient key-value store. Administer these via the UI (Admin -> Variables), code, or CLI. Moreover, it's feasible to manage Airflow Variables using Environment Variables. Adopt the naming convention AIRFLOW_VAR_{VARIABLE_NAME} to align with this method.

4. **TriggerDagRunOperator with Config**: To initiate another DAG run and pass configurations, utilize the TriggerDagRunOperator. The `conf` parameter allows for the transmission of a configuration dictionary to tasks in the target DAG. For manual DAG triggers from the UI with a specific configuration, there's no need for this operator.

5. **ExternalTaskSensor**: This isn't a direct data-passing mechanism, but it establishes inter-DAG dependencies. By setting up such dependencies, you can indirectly convey the state or result of one task or DAG to another.

As comprehended above, there exist a lot of methods to pass information between tasks. Let’s take a look at 2 of the means with some example scenarios.

In Scenario 1, an Extract task retrieves data from a table, including timestamps, followed by a conditional task determining the age of the timestamp against X. If older, it transitions to a task printing a LOG statement; if not, it moves to another task performing minor transformations and updating the table with a new execution timestamp. This task utilizes Branching Operators.

```python
def check_data_age(**kwargs):
    pg_hook = PostgresHook(postgres_conn_id="postgres_default")
    sql = "SELECT created_on FROM accounts WHERE id=1 LIMIT 1;"
    timestamp = pg_hook.get_first(sql)[0]
    specific_timestamp = datetime.strptime('2023-09-15 15:30:00', '%Y-%m-%d %H:%M:%S')

    return 'store_data_task' if timestamp < specific_timestamp else 'log_not_older_task'

def store_transformed_data(**kwargs):
    pg_hook = PostgresHook(postgres_conn_id="postgres_default")
    # Transformation logic and SQL update statement
    update_sql = """
        UPDATE accounts
        SET unit = unit + 5
    """
    pg_hook.run(update_sql)

def log_data_message():
    print("Value in table is newer!")

branch_task = BranchPythonOperator(
    task_id='branch_task',
    python_callable=check_data_age,
    provide_context=True,
    dag=dag
)
```

Figure 5.22: Example scenario showing a branching decision on the basis of data from DB [Screenshot]

Scenario 2 involves triggering a DAG directly in the Airflow UI, incorporating the execution timestamp from an external config JSON. Tasks refer to this timestamp and use a conditional to execute minor transformations or print a LOG statement. This shows the usage of Configs to Trigger DAG Run.

Figure 5.23: Example scenario in which a DAG is passed info externally through the Airflow UI [Screenshot]

6. Exploration Of Data Centric Scheduling Approaches

Complexities in Multi-Workflow Pipelines

In the expansive domain of data management, particularly in the realms of ETL workflows, the orchestration of tasks is crucial. As organizations delve deeper into the data world, they end up building a plethora of workflow pipelines, each teeming with intricate tasks. As these proliferate, the ability to delineate individual task responsibilities or discern the relationships between them becomes an uphill battle. The solution? A shift towards a data-centric approach to scheduling.

6.1 The Potential of Data-Centric Scheduling in ETL Workflows

Traditional command-based schedulers typically run at specified intervals, irrespective of the state or quality of the data. They don't have a deep-seated understanding of data attributes, resulting in inefficiencies. For instance, if the data source has not changed since the last update, a conventional scheduler would still initiate the ETL process, leading to unnecessary resource consumption.

In contrast, data-centric scheduling is rooted in the principle of deriving insights directly from the data. Instead of relying on fixed schedules, this approach contemplates the data's intrinsic attributes, ongoing queries, and even manually annotated metadata. The result? A more intelligent, responsive, and efficient ETL process.

This leads us to think about the pivotal role data plays in implementing ETL workflows. Transformed data is crucial to gain insights and forecast growth, but can it also be used to aid the comprehension of scheduling pipelines? The essence of this approach is simple yet transformative: use the same data designed to extract business insights to also refine the scheduling processes.

While our focus is on exploring the approaches and benefits of data-centric scheduling, several other strategies could be used for ETL scheduling. A few of them are listed in the figure below. These include event-driven (triggering ETL jobs based on specific events or conditions within the system), user-driven (allowing end-users to trigger ETL processes on-demand), resource-centric (allocating ETL processes based on available system resources to maintain performance), command-centric (starting ETL jobs through direct commands or scripts -> provides flexibility and can be useful for ad-hoc ETL tasks that do not fit into a regular, sequenced workflow), etc.

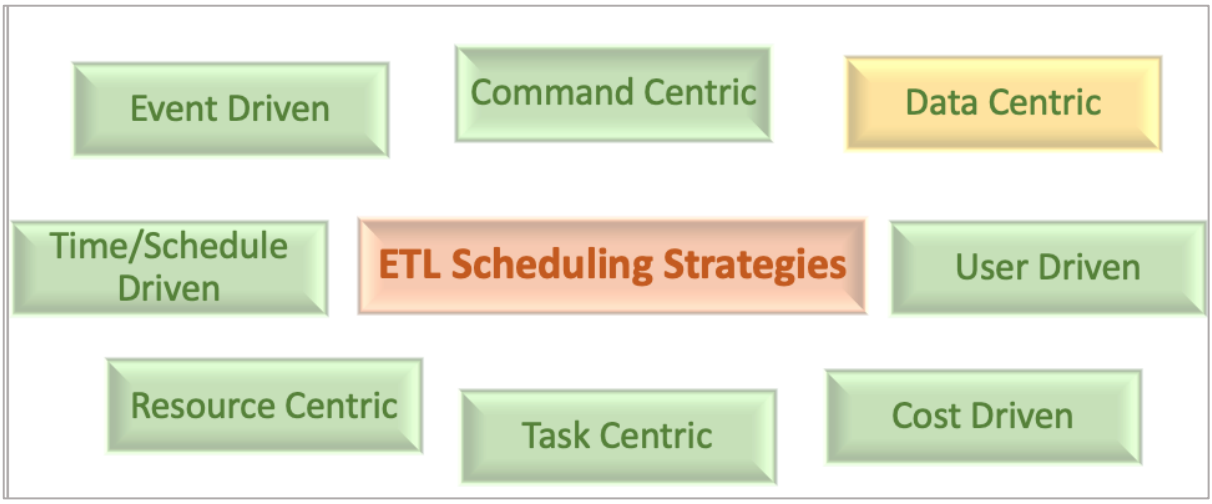

Figure 6.1: Various Scheduling Strategies that can be used to schedule or optimize ETL workflows

Workflow pipelines that are built are rarely short-lived. The very first time we build and adapt the workflow, the metadata might not seem to be of significant value to us as we exactly know what the requirement specifications are, and hence the pipeline is tailor-made for the purpose. However, the subsequent modifications and optimizations would become increasingly difficult as the pipeline grows.

Consider a hypothetical scenario of a retail company that originally built an ETL workflow pipeline tailored for its brick-and-mortar store sales data. It would be designed to extract daily transactions, transform them to consolidate product sales, and load them into a central warehouse for analyzing sales performance.

Over the years, the company ventured into online sales, repurposing the original pipeline to incorporate online data. This presented challenges: the structure, once optimized for physical store data, was now handling diverse data sources such as online sales, reviews, and physical inventories. Without relevant metadata, the pipeline could face inefficiencies.

In this case, if we had metadata that clearly documented exceptions, upstream, downstream, and other specifics on the tasks -> any required modifications, such as adding a new source or adjusting transformation rules, become streamlined. As data infrastructures evolve, maintaining and dynamically updating this metadata ensures that workflows remain efficient and relevant, adapting gracefully to the ever-changing data landscape.

Workflow pipelines that are initially built with a specific narrow purpose in mind might have transformed over the years and be repurposed into something else entirely. As we do not have any metadata or data insights about the kind of data that we are dealing with, this pipeline might still go on working albeit with much lesser efficiency than it was before when it was originally designed

This illustrates the importance of data-centric approaches as opposed to purely task-based approaches in adapting and optimizing ETL pipelines over time.

6.2 Enhancing Decision-Making with Data Insights and Metadata

To elucidate the transformative potential of dynamic data insights, consider a pipeline that has been running for some time and has partially failed. Traditional approaches would leave developers in the dark, struggling to ascertain the next best move. Is a complete pipeline restart necessary? Or can the process be resumed from the point of failure?

For instance, if a pipeline malfunctioned post-extracting data from 99% of sources, reinitiating the entire pipeline would be counterproductive. Instead, targeting the remaining 1% would suffice. This instance underscores the power of dynamic metadata. Such information, constantly updated based on real-time data, provides invaluable insights into the data's origin, purpose, unique fields, and ultimate destination.

This is just an example of one of the many ways in which dynamic metadata that is based on the data can help us make smarter optimization decisions.

When ETL pipelines are first constructed, they are typically laser-focused on specific business objectives. In these nascent stages, the associated metadata—information about data sources, transformation logic, destination points, etc.—may appear somewhat superfluous. Developers, at this juncture, have a crystal-clear understanding of the pipeline's spec and business goals.

However, the digital landscape is never static. Over time, that once sharply defined ETL pipeline may evolve, with its initial purpose diluting or morphing entirely. Without well-documented metadata, this evolution can render the pipeline less efficient than its original design. Due to the lack of insights into the existing data, it's a herculean task to identify inefficiencies or areas ripe for optimization.

Without an understanding of the data and its trajectory, making modifications or optimizations to these pipelines becomes akin to navigating a maze in the dark.
Dynamic metadata is akin to having a compass in the aforementioned maze. It provides insights into where the data originates, its current state, purpose, special attributes, and ultimate destination.

Think of metadata as a blueprint: for instance, in an ETL process, metadata could denote source system details, transformation logic applied, and the intended final destination in the data warehouse. Such insights are invaluable for maintaining and updating ETL workflows.

6.3 Extension of ETL Scheduler to Introduce Data-Driven Features

In Apache Airflow, we extend the scheduler to introduce Data-Aware Scheduling for managing intricate production scenarios which often need elaborate Directed Acyclic Graphs (DAGs).

Envision a multifaceted workflow pipeline constructed of fifty diverse tasks, each extracting data from varying sources such as CSV files, AWS, OpenStack, or databases. Other tasks may be doing some transformation work relevant to accounts, shipping, distribution, or supply chain. Then we have 10 other tasks that are designated to load these transformed data into multiple different targets including data warehouses, caches, file stores, or be intermittently stored for further transformation.

When the number of such sophisticated DAGs escalates to fifty, the complexity intensifies, complicating the understanding of these workflows contributed partly by the absence of a universal anchor illustrating the nature of tasks, data, or transformations being dealt with.

This necessity for insight and clarity steers the conversation toward the implementation of a customized, predefined set of annotations. Such annotations are beneficial for swiftly categorizing tasks and intuitively understanding, upon first viewing a DAG, the specific operations that a task is designed to execute. This organized categorization optimizes workflow management and facilitates prompt and effective interactions with tasks, ensuring seamless and efficient handling of data and transformations within the extensive and diversified workflow pipelines in Airflow.

To comprehend the function performed by these tasks, the visibility of these predefined task annotations from the UI is valuable.

As discussed in the previous section, in order to illustrate an implementation of a data-centric approach in Airflow, we came up with some methods through which we could better understand the workflow pipeline and aim to address the workflow complexity using the data from the pipeline for use in supplementing the metadata as well as for deriving insights

There are many methods in which we can enhance the data-centricity of a scheduler. These methods mainly involve the introduction of intelligence derived from data-driven insights into the scheduling process. Let’s take a look at a few ways in which we might achieve it in the figure below,

Context-Aware Task Prioritization: Prioritize tasks based on their contextual importance to business operations, ensuring critical data-centric tasks are addressed first for optimal operational effectiveness.

Machine Learning-Enhanced Scheduling: Employs machine learning to analyze data access and processing patterns, optimizing task allocation over time for continual improvement of scheduling efficiency.

Predictive Load Balancing: Utilizes historical data and algorithms to anticipate system load, distributing tasks optimally to prevent bottlenecks and enhance overall performance efficiency.

Data Locality Optimization: Focusing on minimizing data movement by scheduling tasks close to where the data resides, reducing latency and improving overall system performance. This is particularly relevant in distributed computing environments like Hadoop.

Data Quality-Based Scheduling: Prioritizes tasks based on the quality of the data they process. High-priority is given to tasks handling high-quality, reliable data to ensure better outcomes and decision-making

Figure 6.2: A selection of approaches that can be used to achieve data centric scheduling

Now that we have seen a few approaches to data-centric scheduling, let's look at 3 more methods that we can adopt to introduce data-driven insights in the scheduling process. These 3 methods will then be prototyped in our selected scheduler along with concrete demonstration use cases representing how they could be used to optimize the scheduling capabilities of a scheduler. Below we outline the methods in brief:

Enhanced Metadata Storage: By embedding richer metadata in files or databases, each task, data point, or transformation can self-describe its properties, sources, dependencies, and purpose. This not only allows for a deeper understanding of each workflow component but can also inform the scheduler to make intelligent decisions based on data characteristics.

Intuitive UI Visual Cues: Visual cues in user interfaces can provide users with a clearer picture of the workflow dynamics. Through color-coding, icons, or other graphical representations, users can quickly discern the type of task, the nature of data being processed, or the kind of transformation applied. This not only augments understanding but also aids in rapid troubleshooting, optimization, and enhancement of the data integration processes.

Data-driven Insights: This approach utilizes data sources to derive critical insights for intelligent task and workflow scheduling. It combines data analytics with workflow management systems, enabling dynamic scheduling based on real-time data insights. This approach goes beyond static schedules by assessing metrics like data freshness, change rates, and activity patterns within database systems. This assessment enhances operational efficiency by reducing unnecessary processing, optimizing resource usage, and ensuring timely task execution.

Incorporating these three dimensions would undoubtedly bring more dynamism and clarity to complex workflows, ensuring that data integration is not just powerful but also adaptive and insightful.

6.4 Technical and Development Aspects

As mentioned in the previous section, to introduce data-centricity into Airflow, we came up with three methods, among the many possible. In this section, we go into the technical development aspects of the prototype to bring to life two of the methods involving metadata. The third approach will be undertaken in the next section.

6.4.1 Building a Custom Metadata Plugin

Being open source, one of the advantages of Airflow is its extensibility. We can customize Airflow to adapt to our requirements by writing plugins.

In Apache Airflow, tasks can represent a wide array of operations. However, tasks that solely perform operations like deletions are not data-centric. Data-centric tasks are mainly those that manipulate or manage data in some meaningful manner, tasks that involve data processing, like querying a database and persisting the results for subsequent tasks etc.

Oftentimes times developers/maintainers hardly plough through the code base and rely significantly on the UI to help understand the workflow. So to illustrate, we defined a set of custom annotations that could be used to annotate the individual tasks (one or many tasks make up a DAG). Along with the annotations they could also include some relevant information like dependencies, DB tables, special fields, warnings, upstream, exceptions, failure contacts, and descriptions.

The syntax for the annotate decorator is simple -> it just involves placing the “@annotate” keyword decorator on the task to annotate followed by a list of comma-separated metadata attributes enclosed in round brackets. The `ALLOWED_METADATA_ATTRIBUTES` are a set of keys that define permissible metadata for a given system. Each attribute serves a specific purpose and the list is fluid which means that it can be easily extended as requirements progress.

@annotate(list of comma separated allowed metadata attributes in the format -> attribute=”value”, attribute=”value”, attribute=”value”……)

- Annotation Name: @annotate
- Possible Parameters that can be passed into the metadata:
    - category: This designates the classification of the task within the workflow

- desc: Provides a brief explanation of what the task does.
- dependency: Lists other tasks or conditions that must be completed or met
- source_db: Specifies the source database from which the task retrieves data.
- target_db: Indicates the target database where the task outputs its data.
- upstream: Defines tasks that should be executed before the current one
- downstream: Identifies tasks that should be executed after the current one
- failure_contact: Contact information for who should be notified if task fails.
- special_fields: Remarks field that could denote any special data fields that the task handles, possibly requiring additional attention or processing.
- long_running_task: A flag to indicate if the task typically takes a long time to complete, which might affect scheduling and resource allocation.
- data_sensitivity_level: Describes the confidentiality level of the data processed by the task (e.g., public, confidential, secret).
- exceptions: Outlines any special cases or conditions where the normal operation of the task might be altered or skipped.
- sla: Stands for "Service Level Agreement" and would detail the expected performance standards or deadlines the task is required to meet.
- last_modified_by: Tracks who last made changes to the task's definition or code.
- log_performance: Indicates whether the task should log its performance metrics, such as execution time or resource usage.

The above is not an exhaustive list of possible parameters. The following picture shows us a sneak peek of how annotations are declared in the code on individual tasks.

```
# Custom Annotated Tasks #

@annotate(category="ExtractFromCSV", desc="Extract Client Data from CSV file", dependency="none",
data_sensitivity_level="high", upstream= "monthly_dqm_file", special_fields="Confidential | Contai
ns PII")
def extractLegalInfo():
    print("Running my task 1")


@annotate(category="SalesTransform", desc="Quarterly Sales Transform", dependency="extract_store_s
ales, extract_online_sales", target_db= "t_04", exceptions="Will not be run on public holidays")
def transformSales():
    print("Running my task 2")


@annotate(category="LoadToCloud", desc="Loads generated report to AWS", dependency="validateInfo,
transformSales", downstream= "AWS S3 Bucket", log_performance="true", failure_contact="support_per
son@aws.com")
def loadReport():
    time.sleep(2)
    print("Running my task 3")
```

Figure 6.3: Annotations are used to decorate the tasks with metadata that can be leveraged in tasks [Screenshot]

These annotations serve a dual purpose. First, they facilitate the swift categorization of tasks, enabling users to instantly discern the specific operations associated with a task upon the initial inspection of a DAG. For example, the task could be annotated as ExtractFromAWS, ExtractFromCSV, SQLTransform, SalesTransform, FooBarTransform, LoadToCSV, LoadToCloud, etc. This also guards against "black box" operations, where the underlying function of a task is obscure or unclear.

The Plugin Structure in Airflow is defined as follows:

```
airflow/
 |— plugins/
|   ├— metadata_plugin/
|   |   ├— __init__.py
|   |   ├— decorators.py
|   |   ├— utils.py
```

In our metadata_plugin, we set a path (METADATA_PATH) for storing generated files. A list (ALLOWED_METADATA_ATTRIBUTES) of valid attributes for metadata and a list (ALLOWED_CATEGORIES) of valid categories for some tasks or processes are also defined. These lists deter rogue info from being entered into the metadata.

Currently, in this prototype, most of the attributes do not have restrictions in terms of the values that can be assigned to them. Stricter checks on the attribute values can be handled once the prototype is enriched. To illustrate this, we have imposed one of the attributes “category” with a constraint in the form of a list of “ALLOWED_CATEGORIES”. Any value deviating from this list will not be allowed for the attribute “category”. Similarly, we can impose constraints and sanity checks for each of the other attributes.

```
# Define where the metadata files are stored
METADATA_PATH = "/home/ubuntu/airflow/metadata"
if not os.path.exists(METADATA_PATH):
    os.makedirs(METADATA_PATH)

# Valid metadata attributes
ALLOWED_METADATA_ATTRIBUTES =
    ["category", "desc", "dependency", "source_db","target_db",
     "upstream", "downstream", "failure_contact", "special_fields",
     "long_running_task", "data_sensitivity_level", "exceptions",
     "sla", "last_modified_by", "log_performance"]

# Validation for categories allowed
ALLOWED_CATEGORIES =
    ["LoadToCSV", "SalesTransform", "ShippingTransform", "ExtractFromAWS",
    "ExtractFromCSV", "SQLTransform", "FooBarTransform", "LoadToCloud"]
```

Figure 6.4: Metadata can be protected from rogue entries with the help of extendable whitelists [Screenshot]

Second, by incorporating these predefined annotations directly into the tasks, we intend to provide users with immediate visual insights from the UI. This approach ensures that the function of each task is transparently evident without delving deep into the code/implementation details. During a DAG run, the annotation of a task should be presented in the user interface. This can be in the form of a label or any other visual representation that sources information directly from the task's annotations. Given in the following picture is a screenshot of how the UI looks during the DAG run enriched with the additional insights from our annotations. This info is now available when you hover over the Tasks and is customizable in case you want more or less data.

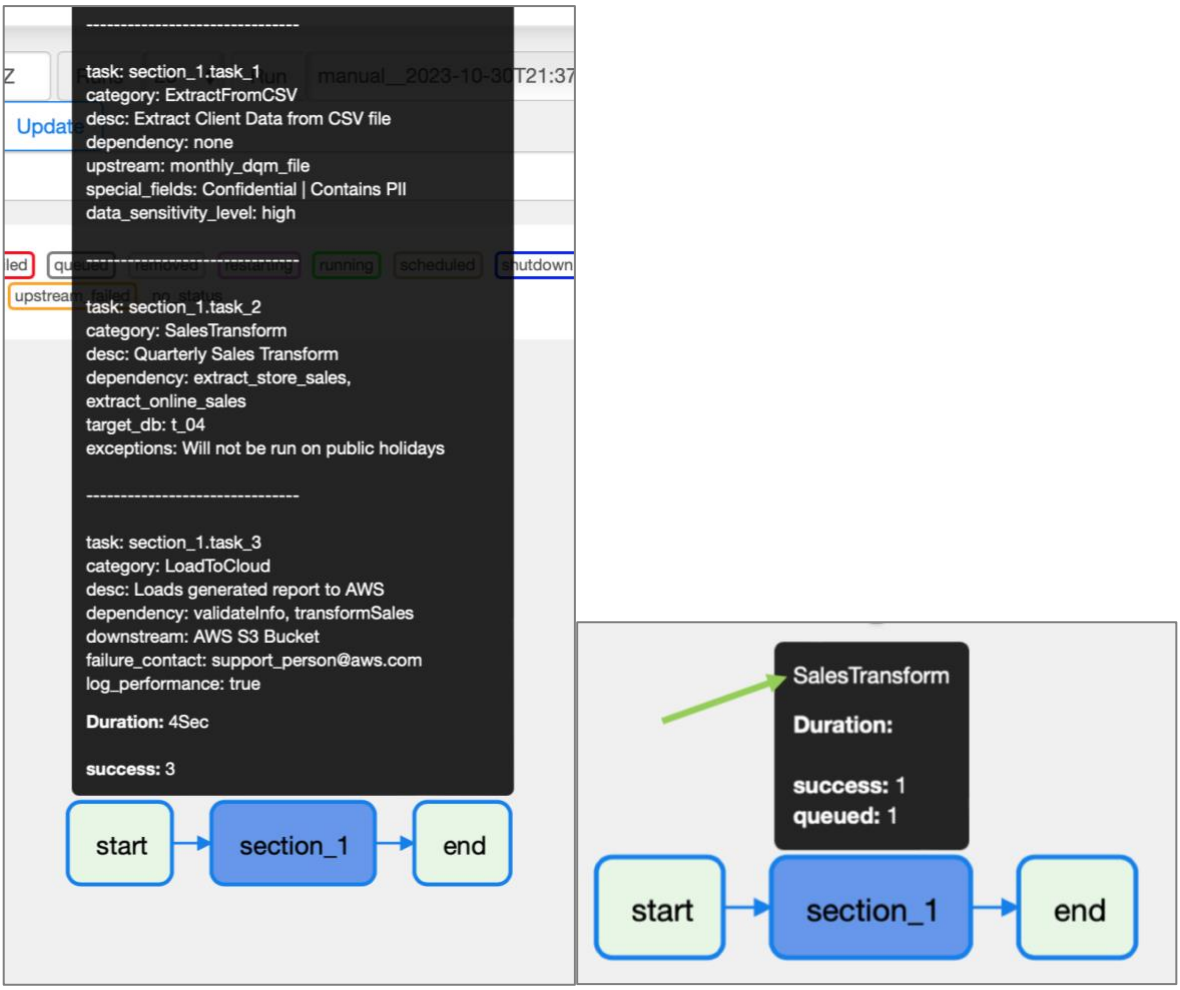

Figure 6.5: Representation of metadata through visual cues in the UI. Can be customized to project more of less information in the UI [Screenshot]

Another option would have been to name the tasks appropriately, but naming is a weak option that does not offer consistency across DAGs. Naming is dependent on the individual developers and can vary according to the knowledge of the individual.

In addition to the above enhancements, we also create a mechanism to store metadata associated with each task, achieved by saving metadata in a dedicated file, ensuring that

essential information remains easily accessible and can be leveraged for future reference or analysis.

```
def annotate(**kwargs):
    for key in kwargs:
        if key not in ALLOWED_METADATA_ATTRIBUTES:
            raise ValueError(f"Invalid metadata attribute: {key}")
        if key == "category" and kwargs[key] not in ALLOWED_CATEGORIES:
            raise ValueError(f"Invalid category value: {kwargs[key]}")

    def decorator(func):
        metadata = {key: kwargs.get(key, "") for key in ALLOWED_METADATA_ATTRIBUTES}

        @functools.wraps(func)
        def wrapper(*args, **kwargs):

            # Fetch the dag_id from Airflow's Variables
            dag_id = Variable.get("dag_id", default_var=None)

            # Save the annotation to CSV when the task is executed
            save_annotation_to_csv(dag_id, func.__name__, metadata)
            return func(*args, **kwargs)

            # Demo: Log Performance metrics if annotated
            # Check if performance logging is required
            if metadata.get('log_performance') == "true":
                return PerformanceMetrics.log_performance(func, *args, **kwargs)

        wrapper._annotation = metadata
        return wrapper

    return decorator
```

Figure 6.6: Excerpt from the plugin employed for manipulation, storage and processing of metadata [Screenshot]

Metadata Publication: The metadata publication process involves creating metadata manually for each task, which describes the task and the data it processes. This metadata is saved in files with standardized names such as "example_dag1_10_09_23_dagRunId.csv." These files are then stored in a centralized directory that has been defined in advance. This centralization facilitates access to other Directed Acyclic Graphs (DAGs). Each DAG is responsible for generating a singular metadata file, within which the information is organized into sections according to the related tasks. Below we see an example of how the metadata publication could look like.

```
-rw-rw-r-- 1 ubuntu ubuntu  751 Oct 30 21:37 demo_annotations_2023_10_30_21_37.csv
ubuntu@airflow:~/airflow/metadata$ cat demo_annotations_2023_10_30_21_37.csv
task: extractLegalInfo
category: ExtractFromCSV
desc: Extract Client Data from CSV file
dependency: none
upstream: monthly_dqm_file
special_fields: Confidential | Contains PII
data_sensitivity_level: high

----------------------------------------------

task: transformSales
category: SalesTransform
desc: Quarterly Sales Transform
dependency: extract_store_sales, extract_online_sales
target_db: t_04
exceptions: Will not be run on public holidays

----------------------------------------------

task: loadReport
category: LoadToCloud
desc: Loads generated report to AWS
dependency: validateInfo, transformSales
downstream: AWS S3 Bucket
failure_contact: support_person@aws.com
log_performance: true

----------------------------------------------
```

Figure 6.7: Sample file data containing metadata information preserved for future access [Screenshot]

Post-task execution, it's vital to identify where the transformed data resides, ensuring that subsequent tasks can locate and utilize this data seamlessly. This enables traceability and understanding without the need to inspect the task's intricate details.

6.5 Validation Of The Prototype Through Some Practical Use Cases

In the previous section, we have seen the construction of the metadata plugin. Now, in this section let’s take a look at some of the practical use cases where the data-centric scheduling approach demonstrated via the metadata plugin could be used and specific instances where the metadata information could be leveraged.

For better clarity, let’s revisit the hypothetical example of the retail company from section 6.1 examining how the metadata plugin could be leveraged here. In each of the following use cases, we see the benefits or enhancements that using the metadata plugin could bring into the day-to-day working of the retail store across different sections -> By bringing swift resolution & recovery through automation in the event of a task failure (case 1), simplifying DQM by reducing unnecessary checks and providing developers with the flexibility to easily modify these checks through annotation adjustments (case 2), taking a smarter approach to task execution by ensuring tasks are not blindly retried but are instead executed using alternative execution strategies after multiple (case 3).

Collectively, these improvements demonstrate the plugin’s ability to not only save resources but also to enhance overall operational efficiency in a retail setting.

Use Case 1

In this use case, we tackle a scenario where the plugin assists in the automation of failure responses and helps in accelerating the response times of developers or system maintainers to respond and address issues, enhancing overall system reliability and efficiency."

Consider the task “extractLegalInfo” below which is annotated with data_sensitivity_level=high. In the given use case, a high data sensitivity level is assigned to a task that is deliberately allowed to fail by triggering an exception.

```
@annotate(category="ExtractFromCSV", desc="Extract Client Data from CSV file", dependency="none", data_sensitivity_level="high"
 upstream= "monthly_dqm_file", special_fields="Confidential | Contains PII")
def extractLegalInfo():
    print("Running my task 1")
    # Demo - send out email to team on failure of a task that has annotation with data_sensitivity_level="high"
    raise Exception("Simulating a task failure!")
```

Figure 6.8: Prototype segment depicting an attribute like data sensitivity level manipulated [Screenshot]

A dedicated class, AlertUtils, which is based on the metadata plugin is implemented to manage such failures. It uses the task's metadata to determine the need for email alerts. In this case, the data sensitivity level—marked high—dictates the AlertUtils class's behavior: if the task fails, it generates and sends a notification email to the designated support team, ensuring that tasks with critical data sensitivity are promptly attended to upon failure. The use case illustrates a robust error-handling mechanism within a data processing workflow, where task sensitivity is accounted for.

Figure 6.9: Code illustrating use case where the alerting is triggered on the basis of the annotation values [Screenshot]

Expanding the use case further, suppose a file is sourced from an upstream department, such as Legal. To automate failure alerts directly to the origin department, the metadata could be enhanced by adding a "failure_contact" attribute. This attribute would store the contact

information of the concerned department. In the event of a task failure, the AlertUtils class would retrieve this information from the task's metadata and automatically send a failure notification to the specified contact. This automation streamlines the process, ensuring direct communication and significantly reducing manual intervention in the error-handling workflow.

```
# Custom Annotated Tasks #

@annotate(category="ExtractFromCSV", desc="Extract Client Data from CSV file", dependency="none", data_sensitivity_level="high",failure_contact="xyz@legal.com",
 upstream= "monthly_dqm_file", special_fields="Confidential | Contains PII")
def extractLegalInfo():
    print("Running my task 1")
    # Demo - send out email to team on failure of a task that has annotation with data_sensitivity_level="high"
    raise Exception("Simulating a task failure!")
```

Figure 6.10: Extension annotations for the use case with an additional attribute “failure_contact” [Screenshot]

Use Case 2

In this second use case, we demonstrate another scenario where the metadata plugin can be used to enhance the processing time of the ETL workflows. Data Quality checks are an indispensable part of ETL and regular DQM jobs ensure that the data is of standard quality. In our use case, the annotations processed by the metadata plugin, enable us to perform the DQM checks after the extraction of data thereby avoiding data upload failure further down the lane into the databases. DB upload operations are typically long-running, resource-intensive, and costly operations, and hence benefit from the early interception of data issues, fostering more efficient workflows.

The plugin's annotation-driven system enables maintainers to direct DQM procedures, targeting essential fields and conditions within the data. The maintainer who has specific knowledge about the task, can use the plugin to supply metadata information that can be used to customize the DQM checks.

This tailored approach makes the whole workflow more efficient. It conserves time by streamlining code maintenance, allows developers to easily adjust the required annotations, and also refines the workflow by eliminating redundant checks and skipping unnecessary conditions.

As seen below, the core plugin can be extended in multiple ways to adapt to the task at hand, in this case -> DQM checks. Here is a simple depiction of what a probable DQM check class looks like when using the plugin to process the annotations.

```python
import re
from metadata_plugin.utils import get_annotations

class DataQualityChecker:

    @staticmethod
    def check_data_quality(task, data):
        metadata = get_annotations(task)
        print("Metadata: {metadata}")

        # Extended checks for long-running tasks
        if metadata.get("long_running_task") == "true":
            print("Performing extended data quality checks for long-running task.")
            DataQualityChecker.validate_data_accuracy(data)
            DataQualityChecker.validate_data_completeness(data)

        # Extended checks for tasks with special fields
        if metadata.get("special_fields"):
            print(f"Performing special field checks for: {metadata['special_fields']}")
            DataQualityChecker.validate_special_fields(metadata["special_fields"], data)


    @staticmethod
    def validate_data_completeness(data):
        # Check if essential fields in the data are present and not null
        print("Data completeness check passed.")

    @staticmethod
    def validate_data_accuracy(data):
        # Implement checks for accuracy
        print("Data accuracy check passed.")

    @staticmethod
    def validate_special_fields(special_fields, data):
        # Example: Validate sales_volume integrity
        print("Special fields validation check passed.")

    @staticmethod
    def dqm_check(context):
        task_instance = context['task_instance']
        # Determine the data location from the context
        data_location = "/path/to/data/" + context['ds'] + "_extract.csv"
        DataQualityChecker.check_data_quality(task=task_instance.task, data=data_location)
```

Figure 6.11: Demo code processing the annotations and consequently directly influencing the DQM tasks [Screenshot]

As a demo, we have a couple of tasks with suitable annotations. In the example below we see that the extract task on successful data extraction will be subjected to a DQM check. The DQM check will, in turn, process the annotations present on the task to verify or skip the conditions. The maintainers can fine-tune the DQM operations by supplying metadata that directs the plugin to conduct checks on specific fields or under certain conditions. This tailoring is visible in the annotations like long_running_task="true" and data_sensitivity_level="low".

```python
from dqm_checker import DataQualityChecker

# Custom Annotated Tasks #

@annotate(category="ExtractFromDB", desc="Extract sales data from SQL database", long_running_task="true", source_db="main_sql_db", data_sensitivity_level="low", upstream="N
def extractSalesData():
    pass

@annotate(category="SQLTransform", desc="Transform raw sales data", dependency="extractSalesData", target_db="analytics_db", exceptions="Handle missing fields")
def transformRawSales():
    pass

@annotate(category="LoadToReportDB", desc="Load transformed data to data warehouse", long_running_task="true", dependency="transformRawSales", target_db="dw_sales", sla="24
def loadToDataWarehouse():
    pass
```

Figure 6.12: Tailor-made attributes can be used to decorate the tasks which can be used to adapt the DQM tasks [Screenshot]

The plugin's efficacy stems from its ability to interpret annotations—applied via the metadata plugin—on ETL tasks. For instance, when the extractSalesData task successfully retrieves data from a SQL database, as denoted by the @annotate decorator, the DataQualityChecker.dqm_check method is triggered as a callback. This method processes the task's metadata and flags and conducts DQM checks pertinent to that data's context.

```
with TaskGroup("etl_process") as etl_group:
    extract_task = PythonOperator(task_id="extract_sales_data",
            python_callable=extractSalesData,
            provide_context=True,
            on_success_callback=DataQualityChecker.dqm_check)
    transform_task = PythonOperator(task_id="transform_sales_data", python_callable=transformRawSales)
    load_task = PythonOperator(task_id="load_to_dw", python_callable=loadToDataWarehouse)
```

Figure 6.13: Example demonstrating the activation of a DQM check within a DAG [Screenshot]

Such customization proves instrumental in mitigating time spent on code maintenance, allowing developers to easily add or remove annotations based on the requirements of individual ETL tasks.

```
{taskinstance.py:1327} INFO - Executing <Task(PythonOperator): etl_process.extract_sales_data> on 2023-11-19 15:50:28.253236+00
{standard_task_runner.py:57} INFO - Started process 34632 to run task
{standard_task_runner.py:84} INFO - Running: ['***', 'tasks', 'run', 'retail_data_pipeline', 'etl_process.extract_sales_data',
{standard_task_runner.py:85} INFO - Job 1489: Subtask etl_process.extract_sales_data
{task_command.py:410} INFO - Running <TaskInstance: retail_data_pipeline.etl_process.extract_sales_data manual__2023-11-19T15:5
{factory.py:78} ERROR - Did not find openlineage.yml and OPENLINEAGE_URL is not set
{factory.py:37} WARNING - Couldn't initialize transport; will print events to console.
{console.py:29} INFO - {"eventTime": "2023-11-19T15:50:30.643763Z", "eventType": "START", "inputs": [], "job": {"facets": {"own
{taskinstance.py:1545} INFO - Exporting env vars: AIRFLOW_CTX_DAG_OWNER='***' AIRFLOW_CTX_DAG_ID='retail_data_pipeline' AIRFLOW
{python.py:183} INFO - Done. Returned value was: None
{taskinstance.py:1345} INFO - Marking task as SUCCESS. dag_id=retail_data_pipeline, task_id=etl_process.extract_sales_data, exe
{logging_mixin.py:150} INFO - Metadata: {metadata}
{logging_mixin.py:150} INFO - Performing extended data quality checks for long-running task.
{logging_mixin.py:150} INFO - Data accuracy check passed.
{logging_mixin.py:150} INFO - Data completeness check passed.
{logging_mixin.py:150} INFO - Performing special field checks for: sales_volume
{logging_mixin.py:150} INFO - Special fields validation check passed.
```

Figure 6.14: By processing information retrieved from metadata, extended DQM checks are performed [Screenshot]

It facilitates task-specific DQM interventions, avoiding unnecessary checks that do not pertain to the given data context, thereby conserving time and resources. Ultimately, the plugin ensures that the data loaded into DBs is of high quality, supporting more efficient and reliable data operations.

Use Case 3

In this final use case, we focus on how the plugin could be used in case of a restart or retry of a task. In cases where we have tasks that fail multiple times, the unlimited retry or restart of a task without any change in the execution strategy means that the task will keep on failing putting a strain on the resources. If the task has knowledge of an alternative execution strategy, then it means that we could try the task again using another strategy.

To add some context: In typical Airflow workflows, tasks that encounter failures are often retried using the same execution parameters, leading to repeated failures and unnecessary resource consumption. This scenario is especially prevalent in complex data pipelines where tasks may fail due to transient external dependencies or fluctuating data conditions. Continuously retrying these tasks without any change in their execution strategy not only strains system resources but also delays the entire pipeline.

To address this challenge, we propose an ExecutionManager, a utility designed to modify the execution approach of tasks based on previous failures. This utility extends the metadata plugin, allowing tasks to be retried with alternative strategies, thereby increasing the likelihood of successful execution upon subsequent attempts.

Here is an example of how the utility function could be implemented using the metadata plugin in order to choose alternative execution strategies:

```python
from metadata_plugin.utils import get_annotations

class ExecutionManager:

    @staticmethod
    def determine_execution_strategies(metadata):
        """
        Determines the execution strategy for a task based on its annotations.
        """
        strategy = metadata.get('execution_strategy', 'default_strategy')
        action = None
        print(f"Strategy from task: {strategy}")

        # Define actions based on strategy
        if strategy == 'in_order':
            action = 'wait_for_prior_tasks'
        elif strategy == 'in_parallel':
            action = 'proceed_without_waiting'
        elif strategy == 'latest_only':
            action = 'skip_if_newer_exists'
        else:
            action = 'default_action'

        return action

    @staticmethod
    def handle_task_restart(context):
        """
        Handler called when a task fails and needs to restart.
        """
        task_instance = context['task_instance']
        metadata = get_annotations(task_instance.task)

        if not metadata:
            return
        else:
            action = ExecutionManager.determine_execution_strategies(metadata)
            print(f"Action determined for retry based on strategy: {action}")
            return action
```

Figure 6.15: ExecutionManager which helps to select alternative execution strategies to avoid endless retrys with the same failed strategy [Screenshot]

In this example, the ExecutionManager is integrated into a demo DAG, where it's invoked when the task is retried more than twice. The process begins when a task, like load_data fails. On subsequent failure, instead of a straightforward retry, the ExecutionManager is called to assess the task's current context and previous failures. It then determines an alternative

execution strategy, such as delaying the retry, skipping the task if certain conditions are met (e.g., if a newer instance of the task is running), or altering resource allocations.

For instance, if load_data task fails twice due to a data unavailability issue, the ExecutionManager might recommend a delayed retry, allowing time for the data to become available. This approach, based on the task's annotations and failure context, makes the retry process more intelligent and context-aware.

```python
from utils import annotate, set_annotations
from execution_strategy_manager import ExecutionManager


# Custom Annotated Task #

@annotate(category="LoadToReportDB", desc="Load transformed data to data warehouse", execution_strategy="in_order", target_db="dw_sales", sla="24 hours")
def load_data(**context):
    task_instance = context['ti']
    if task_instance.try_number < 3:
        raise ValueError("Try Number less than 3")
    else:
        # Checking for action determined by the execution manager after 2 failed retries
        action = ExecutionManager.handle_task_restart(context)
        print(f"Task failed twice. Trying new strategy. Action for retry: {action}")

        # Define behavior based on the action
        if action == 'wait_for_prior_tasks':
            print("Waiting for prior tasks to complete before retrying.")
            time.sleep(10)
        else:
            # Possible options: Execute follow up task, Alert for failure, Skip execution etc.
            print("Proceeding without waiting for task to complete")


default_args = {
    "retries": 4,
    "retry_delay": timedelta(seconds=4),
}
```

```
{task_command.py:410} INFO - Running <TaskInstance: load_data_pipeline.load_group.load_data manual__2023-11-20T
{factory.py:78} ERROR - Did not find openlineage.yml and OPENLINEAGE_URL is not set
{factory.py:37} WARNING - Couldn't initialize transport; will print events to console.
{console.py:29} INFO - {"eventTime": "2023-11-20T19:44:22.591276Z", "eventType": "START", "inputs": [], "job":
{taskinstance.py:1545} INFO - Exporting env vars: AIRFLOW_CTX_DAG_OWNER='***' AIRFLOW_CTX_DAG_ID='load_data_pip
{logging_mixin.py:150} INFO - Strategy from task: in_order
{logging_mixin.py:150} INFO - Action determined for retry based on strategy: wait_for_prior_tasks
{logging_mixin.py:150} INFO - Task failed twice. Trying new strategy. Action for retry: wait_for_prior_tasks
{logging_mixin.py:150} INFO - Waiting for prior tasks to complete before retrying.
{python.py:183} INFO - Done. Returned value was: None
{taskinstance.py:1345} INFO - Marking task as SUCCESS. dag_id=load_data_pipeline, task_id=load_group.load_data,
```

Figure 6.16: Simulated failure scenario showing the ExecutionManager in action [Screenshot]

This behavior is also easy to modify. Suppose we discover that the “in_order” execution strategy does not provide the desired results in this case then in the next scheduled run of the DAG, the maintainer could easily modify the strategy and trial it for a few days. A little tweaking of the annotations after observation can lead to a more optimized workflow in the long run. Here only the metadata changes and not the code logic. Here is an example of a change in execution strategy:

```python
# Custom Annotated Task #

@annotate(category="LoadToReportDB", desc="Load transformed data to data warehouse", execution_strategy="latest_only", target_db="dw_sales",
def load_data(**context):
    task_instance = context['ti']
    if task_instance.try_number < 3:
        raise ValueError("Try Number less than 3")
    else:
        # Checking for action determined by the execution manager after 2 failed retries
        action = ExecutionManager.handle_task_restart(context)
        print(f"Task failed twice. Trying new strategy. Action for retry: {action}")

        # Define behavior based on the action
        if action == 'wait_for_prior_tasks':
            print("Waiting for prior tasks to complete before retrying.")
            time.sleep(10)
        else:
            # Possible options: Execute follow up task, Alert for failure, Skip execution etc.
            print("Proceeding without waiting for task to complete")
```

```
{taskinstance.py:1545} INFO - Exporting env vars: AIRFLOW_CTX_DAG_OWNER='***' AIRFLOW_CTX_DAG_ID='load_data_pipeline'
{logging_mixin.py:150} INFO - Strategy from task: latest_only
{logging_mixin.py:150} INFO - Action determined for retry based on strategy: skip_if_newer_exists
{logging_mixin.py:150} INFO - Task failed twice. Trying new strategy. Action for retry: skip_if_newer_exists
{logging_mixin.py:150} INFO - Proceeding without waiting for task to complete
{python.py:183} INFO - Done. Returned value was: None
{taskinstance.py:1345} INFO - Marking task as SUCCESS. dag_id=load_data_pipeline, task_id=load_group.load_data, execut
{console.py:29} INFO - {"eventTime": "2023-11-20T20:00:10.299889Z", "eventType": "COMPLETE", "inputs": [], "job": {"fa
```

Figure 6.17: Simulated failure scenario which shows a change in strategy by simply adapting the metadata annotations [Screenshot]

Benefits: By implementing this method, the DAG becomes more resilient to failures and efficient in resource usage. It ensures that tasks are not blindly retried but are instead executed based on a strategy that considers the reasons behind their failures. This approach not only reduces the load on the system but also accelerates the overall workflow by minimizing bottlenecks caused by repetitive task failures.

6.6 Making Smarter Scheduling Decisions With The Help of Data Insights

In the preceding sections, we explored two metadata-driven data-centric scheduling approaches. In this section, our focus shifts towards harnessing data insights directly from data sources to enhance scheduling optimization.

This approach capitalizes on the capabilities of databases to derive critical insights, thereby enabling the intelligent scheduling of tasks and workflows. Leveraging databases for scheduling decisions involves an intricate interplay between data analytics and workflow management systems, such as Apache Airflow.

This synergy is achieved through methods that extract, analyze, and interpret data patterns and trends. By doing so, it facilitates a more nuanced and context-aware scheduling mechanism that goes beyond static schedules, allowing for dynamic adjustments based on real-time data insights.

Key to this methodology is the development of analytics techniques within database systems that are designed to assess various metrics, such as data freshness, change rates, and activity patterns, which are pivotal in determining the optimal timing and frequency for executing

data-related tasks. This assessment can lead to significant enhancements in operational efficiency, including the reduction of unnecessary processing, optimal utilization of computing resources, and timely execution of data-dependent tasks.

In practical terms, the application of database-driven scheduling decisions can be observed in diverse domains ranging from data warehousing and business intelligence to cloud computing and content management systems. For instance, in data warehousing, intelligent scheduling might involve triggering data processing pipelines only when fresh data is available, thereby optimizing resource usage. In cloud environments, where resource utilization directly impacts cost, scheduling tasks during low-activity periods identified through database insights can lead to cost-effective operations.

Overall, the approach of making smarter scheduling decisions with the aid of database insights is a good example of how data-centric scheduling techniques can boost workflow efficiency and underscores the transition toward more adaptive, responsive, and efficient operational models, driven by the power of data analytics.

6.6.1 Validation Through Some Practical Use Cases

To understand the benefits of a data-centric scheduling approach that derives insights from the data, we'll examine three real-world scenarios that illustrate its capabilities. In each of these use cases, we introduce some derived intelligence into the workflow so that it makes smarter scheduling decisions.

For better understandability, let’s select a common example to demo the cases -> Report generations. Retail institutions usually generate a large number of reports for business analysts and other product owners to comprehend the overview, trends, or forecast of sales data. We attempt via these 3 use cases to introduce some intelligence in the report generation process.

 # Use Case 1

Data Freshness is a crucial concept in optimizing the ETL process. Let us consider a scenario requiring regular report generation, like hourly reports from database tables based on the data received from a table. Oftentimes, there might not be any new changes at all in the DB. In these circumstances, it would be a waste of resources and redundant to generate reports and perform subsequent tasks on unchanged data every hour. As the process of report generation is not intelligent enough to recognize the dearth of new data, we might end up doing costly repetitive operations on the unchanged dataset each hour.

Now, we propose to introduce some data centricity into this process of report generation by implementing a method to evaluate the recency of the records in a database. This method, `analyze_data_freshness`, ensures that the data on which decisions are made is current and reliable. The goal is to automate the process of data monitoring in a manner that is tightly integrated with the scheduling system, thereby allowing for more dynamic and responsive data workflows.

```python
from datetime import datetime, timedelta
from collections import Counter
from airflow.providers.postgres.hooks.postgres import PostgresHook

class DataInsightScheduler:

    # Evaluates the data freshness of the records for a given timespan
    @staticmethod
    def analyze_data_freshness(db_name, timespan):
        hook = PostgresHook(postgres_conn_id='postgres_default')
        connection = hook.get_conn()
        cursor = connection.cursor()

        current_time = datetime.now()
        lookback_time = current_time - timedelta(hours=timespan)
        query = f"""SELECT COUNT(*) FROM {db_name} WHERE last_updated > %s;"""
        cursor.execute(query, [lookback_time])
        result = cursor.fetchone()
        print(f"Result of Data Freshness check: {result} new records found for time period {timespan} hours in {db_name}")
        cursor.close()
        connection.close()
        return result[0] > 0
```

Figure 6.18: Method delivering data insights, checks if there are any new or updated records [Screenshot]

In technical terms, the method `analyze_data_freshness` serves as a static utility within a larger data processing framework, likely an orchestration tool like Apache Airflow. It accepts two parameters: `db_name`, which represents the name of the database to check, and `timespan`, which specifies the lookback period in hours to assess data updates. In the figures below, we see the call to the static utility method which checks for Data Freshness.

```python
from airflow import DAG
from airflow.operators.python_operator import PythonOperator, BranchPythonOperator
from datetime import datetime, timedelta
from data_insights_plugin import DataInsightScheduler


# Database and parameters
db_name = 'dw_report_sources'
timespan = 24  # Last 24 hours
threshold = 10  # Threshold for change rate (%)

def analyze_and_check_change_rate():
    change_rate = DataInsightScheduler.analyze_data_change_rate(db_name, timespan)
    is_below_threshold = DataInsightScheduler.is_change_rate_below_threshold(db_name, timespan, threshold)
    print (f"In the last {timespan} hours, the database exhibited a change rate of: {change_rate}%. Provided threshold required for report generation: {threshold}
    return 'no_report_task' if is_below_threshold else 'generate_report_task'

with DAG('demo_smart_scheduling_data_change_rate', start_date=datetime(2023, 1, 1)) as dag:
    decide_task = BranchPythonOperator(
        task_id='analyze_and_check_change_rate',
        python_callable=analyze_and_check_change_rate
    )

    no_report_task = PythonOperator(
        task_id='no_report_task',
        python_callable=lambda: print("Report generation was not initiated due to the absence of any data updates within the specified time frame.")
    )

    generate_report_task = PythonOperator(
        task_id='generate_report_task',
        python_callable=lambda: print("Report generation criteria satisfied. Initiating the report creation process utilizing the recently obtained data....")
    )

    decide_task >> [no_report_task, generate_report_task]
```

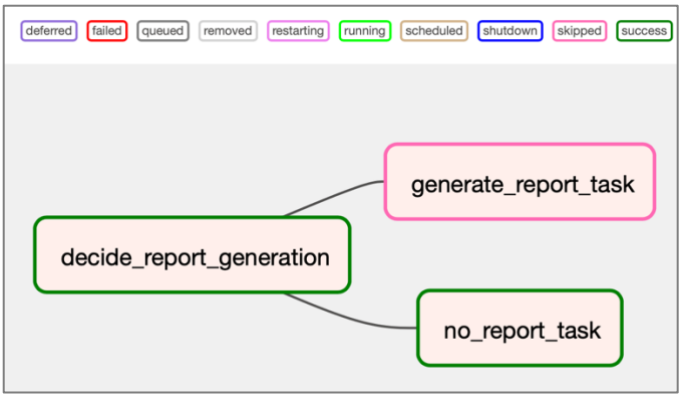

Figure 6.19: Demo DAG and Runtime view that shows the Data Freshness check in use [Screenshot]

As we can see below, from the logs and the graphs, the absence of fresh data skipped the report generation task and the arrival of fresh data triggered the subsequent tasks.

```
[2023-11-25, 18:11:43 CET] {base.py:73} INFO - Using connection ID 'postgres_default' for task execution.
[2023-11-25, 18:11:43 CET] {logging_mixin.py:150} INFO - Result of Data Freshness check: (0,) new records found for time period 24 hours in dw_report_sources
[2023-11-25, 18:11:43 CET] {python.py:183} INFO - Done. Returned value was: no_report_task
[2023-11-25, 18:11:43 CET] {python.py:216} INFO - Branch callable return no_report_task
[2023-11-25, 18:11:43 CET] {skipmixin.py:164} INFO - Following branch no_report_task
```

Figure 6.20: Demo DAG code and runtime view that puts the data freshness check into use [Screenshot]

On adding a few records to the DB, the report is now being generated.

```
{base.py:73} INFO - Using connection ID 'postgres_default' for task execution.
{logging_mixin.py:150} INFO - Result of Data Freshness check: (19,) new records found for time period 24 hours in dw_report_sources
{python.py:183} INFO - Done. Returned value was: generate_report_task
{python.py:216} INFO - Branch callable return generate_report_task
{skipmixin.py:164} INFO - Following branch generate_report_task
{skipmixin.py:224} INFO - Skipping tasks ['no_report_task']
{taskinstance.py:1345} INFO - Marking task as SUCCESS. dag_id=demo_smart_scheduling_data_freshness, task_id=decide_report_generation,
```

Figure 6.21: Display of success/failure scenarios in use case –> Report generation is influenced by data insights result [Screenshot]

In essence, this method embodies the data-centric approach by programmatically ensuring data freshness and automating the validation of data recency, for example, by preventing the triggering of further data pipeline tasks when no new data is present, as evidenced in the logs and graphs showing report generation skipped or initiated based on data updates.

Use Case 2

The high-level goal of this use case is to analyze the rate at which the data has changed over a given time period, which in turn can be used to trigger data-centric scheduling decisions. For example, if the rate of change is above or below certain thresholds, it may initiate different workflows, such as increased data backup frequency for highly volatile datasets or alerts for unexpectedly low activity.

```python
# Analyses the rate at which the data is being updated for a given timespan
@staticmethod
def analyze_data_change_rate(db_name, timespan):
    hook = PostgresHook(postgres_conn_id='postgres_default')
    connection = hook.get_conn()
    cursor = connection.cursor()

    current_time = datetime.now()
    lookback_time = current_time - timedelta(hours=timespan)
    change_rate_query = f"""
        SELECT COUNT(*) FROM {db_name}
        WHERE last_updated BETWEEN %s AND %s;
        """
    cursor.execute(change_rate_query, [lookback_time, current_time])
    changes_count = cursor.fetchone()[0]

    total_records_query = f"SELECT COUNT(*) FROM {db_name};"
    cursor.execute(total_records_query)
    total_count = cursor.fetchone()[0]

    change_rate_percentage = (changes_count / total_count) * 100
    print(f"Data Change Rate: {change_rate_percentage}% of records were updated in the last {timespan} hours in {db_name}")

    cursor.close()
    connection.close()
    return change_rate_percentage

# Evaluates if the data change rate is below a given threshold
@staticmethod
def is_change_rate_below_threshold(db_name, timespan, threshold):
    change_rate_percentage = DataInsightScheduler.analyze_data_change_rate(db_name, timespan)
    return change_rate_percentage < threshold
```

Figure 6.23: Method delivering data insights, checks if rate of change of data is below a given threshold [Screenshot]

The method employs a statistical approach to determine the update frequency of records in a database. This is achieved by executing a SQL query that counts the number of records with a last_updated timestamp between the current time and a defined lookback_time. It then calculates this as a percentage of the total records, providing a clear metric of change over the specified period.

Applying this method again to our report generation use case, we check if the rate of change of data is below a defined threshold value. In case it exceeds the threshold value, a report is generated, otherwise, it is not.

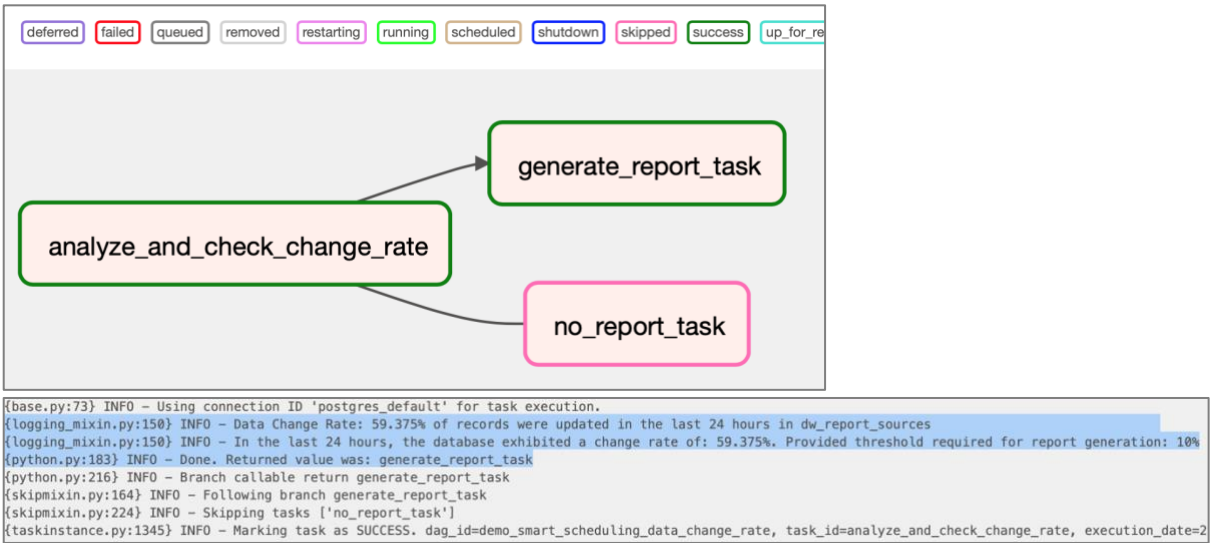

Figure 6.24: Report generation is influenced by the data insights -> Rate of change of data [Screenshot]

This use case demonstrates the strategic advantage of using data insights to prevent unnecessary processing. By analyzing the rate of data changes within a database, the system can discern if significant updates have occurred. If changes are minimal, such as less than 1% of records being updated, it can then decide to forego subsequent resource-intensive tasks. This data-driven decision-making process enhances operational efficiency by ensuring that only meaningful and practical actions are taken, thus avoiding the costs associated with processing largely unchanged data.

Use Case 3

Several reasons affect smart scheduling decisions. One of them is the ability to forecast periods of high or low activity. Here we demonstrate a use case that leverages a method to make intelligent scheduling decisions by identifying periods of high or low activity within the previous year.

By determining these activity patterns, the system can forecast optimal times to schedule resource-intensive jobs, thus avoiding peak times when system load is high, and instead utilizing periods of low activity where the impact on overall system performance is minimized.

```python
# Finds out periods of low activity for a given timespan and returns the intervals
@staticmethod
def find_low_activity_periods(db_name, interval_hours=1, lookback_duration=timedelta(days=365)):
    hook = PostgresHook(postgres_conn_id='postgres_default')
    connection = hook.get_conn()
    cursor = connection.cursor()

    current_time = datetime.now()
    lookback_time = current_time - lookback_duration
    interval_duration = timedelta(hours=interval_hours)
    interval_counts = Counter()

    while lookback_time < current_time:
        interval_end = lookback_time + interval_duration
        activity_query = f"""
            SELECT COUNT(*) FROM {db_name}
            WHERE last_updated BETWEEN %s AND %s;
            """
        cursor.execute(activity_query, [lookback_time, interval_end])
        updates_count = cursor.fetchone()[0]
        interval_counts[lookback_time.strftime("%Y-%m-%d %H:%M:%S")] = updates_count
        lookback_time = interval_end

    cursor.close()
    connection.close()

    # Find the intervals with the lowest activity by reversing the most common list
    activity_periods = interval_counts.most_common()[::-1]  # Reverse to get least common intervals

    # Get top 5 least activity periods
    low_activity_periods = activity_periods[:5]

    print(f"Periods with low activity: {low_activity_periods}")
    return low_activity_periods
```

Figure 6.25: Method delivering data insights, checks for periods of low inactivity over given time interval [Screenshot]

These queries in the method count the number of records updated within each predefined time interval across the previous year. It then uses a Counter object to tally the frequency of

updates in each interval. By analyzing the collected data, the function identifies the intervals with the fewest updates, which indicates low activity periods.

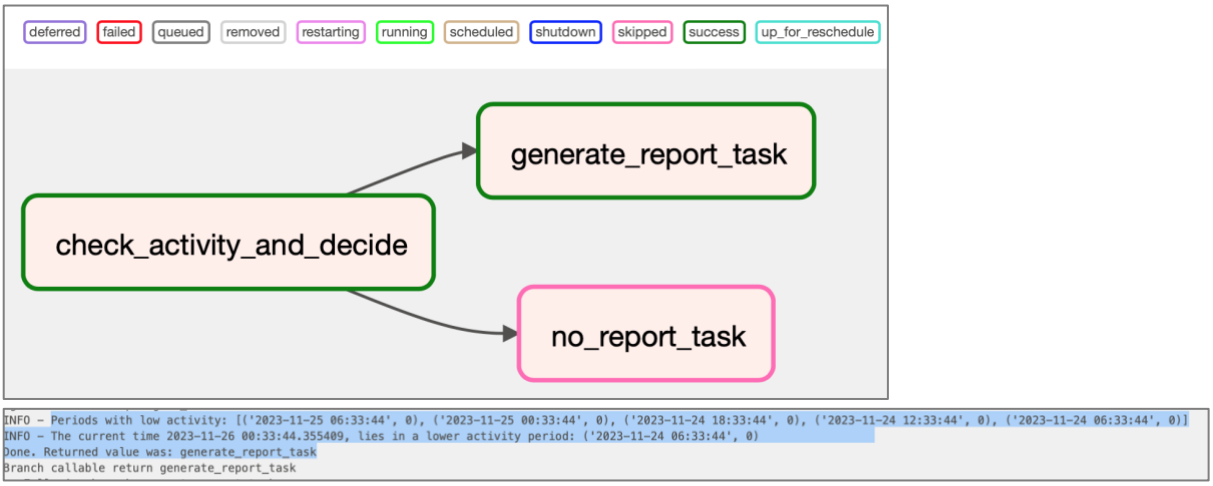

Figure 6.26: Determination of scheduling periods can be assisted with the data insights from plugin[Screenshot]

These periods are ideal candidates for scheduling resource-intensive operations, as they are less likely to interfere with peak usage times and can utilize system resources more efficiently. This predictive scheduling ensures that heavy jobs do not disrupt regular database operations, leading to better performance and cost savings.

7. Limitations and Alternatives

7.1 Airflow Limitations & Mitigations

As with any other scheduler in the market, Airflow also comes with its own set of technical limitations. Let us take a quick look at some of the main limitations and possible mitigations.

Scheduling interval

As Airflow has its own scheduler and adopts the schedule interval syntax from cron, the smallest data and time interval in the Airflow scheduler world is a minute.
As a workaround to this, we can set the schedule interval to a “timedelta object” to schedule tasks at intervals smaller than one minute.

Scheduler latency

There's an inherent delay between when a task is ready to be executed and when it's actually picked up, especially if the scheduler is under load or if the parsing process is slow.
Several methods are available to improve the response time including by adapting the airflow configuration to improve DAG performance.

No native Windows support

It is not straightforward to natively run Airflow on Windows

This can be mitigated by using Docker or setup with Windows WSL

Web UI Performance

For installations with a very large number of tasks or DAG runs, the web interface can become sluggish or unresponsive.

Pagination and limiting the number of DAGs displayed on the UI can help.

Learning curve and Configuration challenges

It is not the most intuitive platform, and its powerful capabilities and built-in extensibility also present us with a steeper learning curve. Not only the syntax but also the configuration and production setup with different executors need some time to get a grip on.

Fortunately, the large & active open-source community is a great help in both the above cases

Database Dependency

Airflow relies on its metadata database, so the performance and uptime of the database are critical. This can be a single point of failure if not set up correctly.

To mitigate this, we can set up database replication and backups. Use high-availability configurations, and monitor database performance and health closely**.**

7.2 Other Limitations

This section delineates some of the key technical limitations encountered during the extension of the ETL scheduler with data-centric capabilities. It critically examines constraints in metadata management, scalability, performance, and integration that emerged during the development and testing phases of the prototype.

Dependency on Metadata Quality

Firstly, the complexity of metadata management poses a significant constraint. The efficacy of data-centric scheduling is contingent on the richness and accuracy of metadata, which in turn depends on the robustness of metadata extraction and storage processes. This thesis does not address the potential for metadata degradation over time or the mechanisms required to maintain metadata integrity across diverse data sources and formats. Limitations may arise from incomplete or inaccurate metadata, which could misguide the scheduler.

Performance Implications

Secondly, the introduction of data-centric features necessitates additional computational overhead for metadata processing and analysis. The extended scheduler's performance could be compromised, potentially negating the efficiency gains from more intelligent scheduling.

While data-centric scheduling aims to improve efficiency, there may be overhead associated with collecting and analyzing metadata that could impact system performance.

User Experience Testing

Furthermore, while the scheduler's user interface is enhanced to offer better visibility into data-centric operations, the thesis might not account for the cognitive load placed on the user. The effectiveness of visual cues and annotations in aiding user comprehension and troubleshooting is assumed without extensive user testing.

Technological Evolution and Compatibility

Lastly, the thesis does not fully address the challenge of ensuring technology stack compatibility. The reliance on specific versions of Airflow and integrations with particular data storage solutions could restrict the scheduler's broader applicability and future-proofing.

The limitations outlined underscore the need for further research and development to surmount the technical challenges associated with metadata management, performance optimization, and system adaptability in the quest for a truly dynamic and intelligent ETL scheduling solution.

8. Conclusion and Future Work

In conclusion, the endeavor to extend a traditional ETL scheduler with data-centric capabilities has revealed promising avenues for enhancing the efficiency and intelligence of workflow management. However, as explored in the preceding sections, this integration is not without its technical impediments. The challenges of maintaining metadata integrity, managing computational overhead, ensuring scalability, and achieving seamless integration into diverse data ecosystems are pivotal considerations that must be addressed as this field evolves.

The research conducted offers a foundational framework upon which future work can be built. It highlights the need for sophisticated metadata management systems that can evolve dynamically with the data they describe. Additionally, the balance between computational efficiency and the depth of data-centric analysis requires further exploration to optimize performance in real-time, large-scale data environments.

Future work should concentrate on refining the data-centric model to enhance its generalizability across various industries and data workflows. This involves developing more advanced methods for metadata analysis that can adapt to different data types and sources with minimal configuration.

Another potent avenue to explore would be the investigation of techniques for minimizing the performance overhead introduced by data-centric features, ensuring the system scales effectively with growing data volumes and workflow complexity.

This thesis explores data-centric scheduling approaches, particularly focusing on the incorporation of source data and metadata. A novel aspect that was unexplored is leveraging SQL queries in the tasks for insightful deductions. By developing a system capable of deconstructing these queries, we gain the potential to not only derive insights independently but also to craft queries that can influence scheduling decisions.

In the thesis, we introduced visual cues dependent on metadata into the scheduler. Future iterations should aim to enhance the user interface and experience to reduce complexity and cognitive load. Also, we have to focus on conducting comprehensive testing and extensive validation across diverse real-world scenarios of the scheduler to ensure its robustness and reliability.

Lastly, ensuring technological agility is paramount. As data processing technologies continue to advance, the system must remain compatible with new tools and platforms. This could involve adopting a modular architecture that allows for easy updates and integration with emerging technologies.

The path forward is ripe with opportunities for innovation. By pursuing these avenues, future research can significantly enhance the capabilities of data-centric ETL schedulers, pushing the boundaries of what is possible in automated data workflow management. By addressing the limitations identified and leveraging the insights gained, the next wave of research can propel data-centric ETL scheduling from a promising prototype to a decisive tool in the data-driven landscapes of tomorrow.

9. References

1) Xu, Z., Li, H., Chen, M., Dai, Z., Li, H., Zhu, M. (2020). Multi-objective Scheduling Optimization for ETL Tasks in Cluster Environment. In: Silhavy, R. (eds) Intelligent Algorithms in Software Engineering. CSOC 2020. Advances in Intelligent Systems and Computing, vol 1224. Springer, Cham. https://doi.org/10.1007/978-3-030-51965-0_14
2) Seenivasan, Dhamotharan. (2023). Exploring Popular ETL Testing Techniques. International Journal of Computer Trends and Technology. 71. 32-39. 10.14445/22312803/IJCTT-V71I2P106.
3) A. Karagiannis, P. Vassiliadis dan AlkisSimitsis, "Scheduling strategies for efficient ETL execution," Elsevier, 2013, ISSN 0306-4379, https://doi.org/10.1016/j.is.2012.12.001.
4) Wojciechowski, A. (2013). E-ETL: Framework For Managing Evolving ETL Processes. In: Pechenizkiy, M., Wojciechowski, M. (eds) New Trends in Databases and Information Systems. Advances in Intelligent Systems and Computing, vol 185. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-32518-2_42

5) Yang, D., Xu, C. (2019). A Distributed Scheduling Framework of Service Based ETL Process. In: Jin, H., Lin, X., Cheng, X., Shi, X., Xiao, N., Huang, Y. (eds) Big Data. BigData 2019. Communications in Computer and Information Science, vol 1120. Springer, Singapore. https://doi.org/10.1007/978-981-15-1899-7_2

6) Oliveira, Bruno and Orlando Belo. "Task Clustering on ETL Systems - A Pattern-Oriented Approach." *International Conference on Data Technologies and Applications* (2015). https://doi.org/10.5220/0005559302070214

7) J. Song, Y. Bao and J. Shi, "A Triggering and Scheduling Approach for ETL in a Real-time Data Warehouse," *2010 10th IEEE International Conference on Computer and Information Technology*, Bradford, UK, 2010, pp. 91-98, doi: 10.1109/CIT.2010.57.

8) Berliantara, Agung. (2017). Scheduling Optimization For Extract, Transform, Load (ETL) Process On Data Warehouse Using Round Robin Method (Case Study: University Of XYZ). Journal of Information Technology and Computer Science. 2. 10.25126/jitecs.20172232.

9) A. Suleykin and P. Panfilov, "Metadata-Driven Industrial-Grade ETL System," *2020 IEEE International Conference on Big Data (Big Data)*, Atlanta, GA, USA, 2020, pp. 2433-2442, doi: 10.1109/BigData50022.2020.9378367.

10) E. Tools, "How do you measure and report the impact and value of your ETL scheduling and monitoring strategy?," *www.linkedin.com*, [Online].

Available: https://www.linkedin.com/advice/0/how-do-you-measure-report-impact-value-your-etl-scheduling

11) “ETL Scheduling and Automation - Top Tools,” Feb. 14, 2023. https://portable.io/learn/etl-scheduler

12) Xu, Guoyao, "Data-Driven Intelligent Scheduling For Long Running Workloads In Large-Scale Datacenters" (2019). *Wayne State University Dissertations*. 2194. https://digitalcommons.wayne.edu/oa_dissertations/2194

13) Astronomer, “Data-driven Scheduling in Airflow 2.4 | Astronomer,” *Astronomer*, Sep. 29, 2022. https://www.astronomer.io/blog/airflow-2-4-and-data-driven-scheduling-at-astronomer/

14) “What are the key features of a good ETL scheduling and monitoring dashboard?,” *www.linkedin.com*, [Online]. Available: https://www.linkedin.com/advice/0/what-key-features-good-etl-scheduling-monitoring-dashboard

15) Qu, W. & Deßloch, S., (2017). Incremental ETL Pipeline Scheduling for Near Real-Time Data Warehouses. In: Mitschang, B., Nicklas, D., Leymann, F., Schöning, H., Herschel, M., Teubner, J., Härder, T., Kopp, O. & Wieland, M. (Hrsg.), Datenbanksysteme für Business, Technologie und Web (BTW 2017). Gesellschaft für Informatik, Bonn. (S. 299-308).

16) “A Brief History of Dataflow Automation,” Prefect. https://www.prefect.io/guide/blog/a-brief-history-of-dataflow-automation/

17) O. Alqaryouti and N. Siyam, “Serverless Computing and Scheduling Tasks on Cloud: A Review”, ASRJETS-Journal, vol. 40, no. 1, pp. 235–247, Mar. 2018.

18) “CEUR-WS.org/Vol-3369 - Design, Optimization, Languages and Analytical Processing of Big Data 2023.” https://ceur-ws.org/Vol-3369

19) “10 Best ETL Job Scheduling Software in 2023 [TOP RATED ONLY],” Software Testing Help, May 28, 2023. https://www.softwaretestinghelp.com/best-etl-job-scheduling-software/

20) https://vaporvm.com/10-open-source-etl-tools/

21) T. King, “Top 13 Free and Open Source ETL Tools for Data Integration,” Best Data Integration Vendors, News & Reviews for Big Data, Applications, ETL and Hadoop, Apr. 2022, [Online]. Available: https://solutionsreview.com/data-integration/top-free-and-open-source-etl-tools-for-data-integration/

22) https://www.quora.com/Which-is-the-best-ETL-scheduling-tool

23) "Gartner Magic Quadrant for Data Integration Tools" reports, and "ETL Tools: Comparison and Guidance" by Bart Baesens, et al. in IEEE Computer (Vol. 49, No. 8, 2016).
24) Chen, G., Zhang, W., Li, D., Zhao, J., Tian, C., Rajkumar, R. R. (2020). Energy-Aware Scheduling for Infrastructure as a Service Clouds with SLA Constraints. IEEE Transactions on Parallel and Distributed Systems.
25) Shehabi, A., Smith, S., Sartor, D., Brown, R., Herrlin, M., Koomey, J., Masanet, E., Horner, N., Azevedo, I., Lintner, W. (2016). United States Data Center Energy Usage Report. Lawrence Berkeley National Laboratory.
26) Gardner, B. (2017). The Art and Science of Data Integration. Journal of AHIMA.
27) Cao, Y., Chen, S., Liu, P., Wang, C., DeWitt, D. J. (2015). Energy-Aware Query Processing on Commodity Hardware. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.
28) Gartner 2022, Magic Quadrant for Data Integration Tools. "https://www.gartner.com/doc/reprints?id=1-2AWA2A6O&ct=220822&st=sb"
29) https://www.astronomer.io/blog/apache-nifi-vs-airflow/#:~:text=Summary,the%20"Stream%20Processing"%20category.

30) https://luigi.readthedocs.io/en/stable/design_and_limitations.html

31) https://www.codingninjas.com/studio/library/pentaho-and-big-data

32) https://medium.com/codex/a-brief-comparison-of-apache-dolphinscheduler-with-other-alternatives-177826a0315c

33) https://medium.com/@ApacheDolphinScheduler/from-airflow-to-apache-dolphinscheduler-the-evolution-of-scheduling-system-on-youzan-big-data-ec897f310f91

34) https://www.advsyscon.com/blog/distributed-job-scheduler-scheduling/

35) https://web.obsidianscheduler.com/why-you-shouldnt-use-quartz-scheduler/

36) Simitsis, A., Wilkinson, K., Castellanos, M., & Dayal, U. (2009, June). QoX-driven ETL design: reducing the cost of ETL consulting engagements. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*(pp. 953-960).

37) https://docs.astronomer.io/learn/airflow-datasets

38) *https://dx.doi.org/10.5441/002/edbt.2016.65*, StreamLoader: an event-driven ETL system for the on-line processing of heterogeneous sensor data

39) https://www.tinybird.co/blog-posts/event-driven-architecture-best-practices-for-databases-and-files

List of citations/references to add:

1. https://github.com/apache/airflow/discussions/30272
2. https://cwiki.apache.org/confluence/display/AIRFLOW/AIP-36+DAG+Versioning
3. https://github.com/apache/airflow/issues/33236
4. https://www.youtube.com/watch?v=a-4yRne3ba4
5. https://airflow.apache.org/docs/helm-chart/stable/manage-dags-files.html
6. https://engineering.zalando.com/posts/2022/06/accelerate-apache-airflow-testing-through-dag-versioning.html
7. https://docs.astronomer.io/learn/managing-airflow-code
8. https://docs.astronomer.io/learn/dag-best-practices#avoid-top-level-code-in-your-dag-file
9. https://www.youtube.com/watch?v=uA-8Lj1RNgw
10. https://docs.astronomer.io/learn/airflow-passing-data-between-tasks
11. Running Airflow at Scale: https://shopify.engineering/lessons-learned-apache-airflow-scale
12. https://towardsdatascience.com/apache-airflow-in-2022-10-rules-to-make-it-work-b5ed130a51ad
13. https://www.youtube.com/watch?v=uA-8Lj1RNgw
14. https://towardsdatascience.com/is-apache-airflow-good-enough-for-current-data-engineering-needs-c7019b96277d
15. https://www.startdataengineering.com/post/apache-airflow-review-the-good-the-bad/
16. https://airflow.apache.org/docs/apache-airflow/stable/faq.html
17. https://towardsdatascience.com/airflow-schedule-interval-101-bbdda31cc463#:~:text=As%20Airflow%20has%20its%20scheduler,Airflow%20scheduler%20world%20is%20minute.
18. https://airflowsummit.org/sessions/2023/future-of-the-airflow-ui/
19. https://airflowsummit.org/sessions/2023/circumventing-airflows-limitations-around-multitenancy/
20. https://www.informatica.com/ch/de/products/data-integration/powercenter.html
21. https://learn.microsoft.com/en-us/sql/integration-services/
22. https://www.ibm.com/products/datastage
23. https://www.oracle.com/middleware/technologies/data-integrator.html
24. https://cloud.google.com/scheduler
25. https://fastercapital.com/keyword/data-integration-process.html
26. https://medium.com/codex/a-brief-comparison-of-apache-dolphinscheduler-with-other-alternatives-177826a0315c
27. https://chess.vstecb.cz/wp-content/uploads/buy2y/8362b3-data-modeling-tools-gartner-magic-quadrant

28. https://ebin.pub/intelligent-algorithms-in-software-engineering-proceedings-of-the-9th-computer-science-on-line-conference-2020-volume-1-1st-ed-9783030519643-9783030519650.html
29. https://research.aimultiple.com/cron-alternative/
30. http://repository.utm.md/bitstream/handle/5014/24038/Conf-TehStiint-UTM-StudMastDoct-2023-v4-p-332-336.pdf?isAllowed=y&sequence=1