Hausarbeit

Apache Airflow

Cloud & Big Data Technologien

Verfasser:	Albrecht, Florian Nico
Arbeit eingereicht am:	16.07.2023
Prüfer:	Prof. Dr. Blochinger, Wolfgang
Fach:	Cloud & Big Data Technologien
Studiengang:	Wirtschaftsinformatik (M.Sc.)
Fachsemester:	1

Inhaltsverzeichnis

1 Einleitung 1
1.1 Motivation 1
1.2 Einführung Apache Airflow 1
1.3 Zielsetzung & Struktur 2
2 Einführung in Apache Airflow 3
2.1 Ursprung 3
2.2 Architektur 3
2.3 Kernkonzepte 5
2.3.1 DAG 5
2.3.2 Tasks & Operatoren 7
3 Verwendung von Apache Airflow 8
3.1 Installation von Apache Airflow 8
3.2 Erstellung & Ausführung von DAGs 9
3.3 Monitoring der Workflows 11
3.3.1 Reiter „Grid“ 11
3.3.2 Reiter „Task Duration“ 12
3.3.3 Reiter „Task Tries“ 13
3.3.4 Log 14
4 Fazit 16
Abbildungsverzeichnis III
Literaturverzeichnis IV
Selbstständigkeitserklärung V

1 Einleitung

Dieses Kapitel hat den Zweck, diese Hausarbeit einzuführen. Zu Beginn wird der Verwendungszweck von Workflow-Management-Systemen motiviert. Anschließend wird Apache Airflow vorgestellt. Abgeschlossen wird dieses Kapitel mit der Zielsetzung und der Struktur dieses Dokuments.

1.1 Motivation

In der Regel trennen traditionelle Anwendungssysteme betriebliche Funktionen voneinander ab und ermöglichen somit keine ausreichende Unterstützung von Geschäftsprozessen, bei denen häufig mehrere Funktionen berücksichtigt werden müssen. Workflow-Management-Systeme (WfMS) hingegen ermöglichen es, Geschäftsprozesse kontinuierlich abzuwickeln und zu steuern. Es wird immer wieder versucht, WfMS im Controlling-Bereich zu nutzen, um stark standardisierte Prozesse zu automatisieren. WfMS sind Softwaresysteme, welche die Durchführung von Aufgaben durch verschiedene Bearbeitern koordinieren. Dies machen sie, indem sie die benötigten Personen über anstehende Aufgaben im Geschäftsprozess informieren, wichtige Informationen zur Verfügung stellen und entsprechende Werkzeuge bzw. Anwendungssysteme zur Unterstützung der einzelnen Bearbeiter zur Verfügung stellen (vgl. Workflow-Management-Systeme 2006, S. 199).

1.2 Einführung Apache Airflow

In der Wirtschaft bieten mehrere Unternehmen diverse WfMS an. Eine Softwarelösung wird von der Apache Software Foundation bereitgestellt: Apache Airflow. Apache Airflow ist eine Open-Source-Lösung, welche verwendet werden kann, um Workflows zu erstellen, zu verwalten und zu überwachen. Airflow hat eine grafische Benutzeroberfläche und wurde in Python geschrieben. Ebenso werden alle Airflow relevanten Dateien in Python entwickelt. Die Software hat sich zu einem Standard für Workflow-Management entwickelt (vgl. Luber und Litzel 2020).

Im Rahmen dieser Hausarbeit wird sich mit Apache Airflow befasst und seine Funktionen, Architektur und Verwendung wird untersucht.

1.3 Zielsetzung & Struktur

Das Ziel dieser Ausarbeitung ist es, Apache Airflow als WfMS näher zu bringen anhand der alltäglichen Nutzung innerhalb eines Unternehmens. Um dieses Ziel zu erreichen wird diese Hausarbeit in der folgenden Struktur aufbereitet.

Kapitel 2 führt Apache Airflow ein. Hier wird zuerst auf den historischen Ursprung eingegangen und anschließend werden die wichtigsten Komponenten von Airflows Architektur vorgestellt und erläutert. Abgeschlossen wird dieses Kapitel mit der Vorstellung der DAGs (Directed Acyclic Graphs) und den dazugehörigen Tasks.

Kapitel 3 hingegen beschäftigt sich mit der initialen Installation von Airflow. Ebenfalls werden hier wichtige Funktionen, wie die Überwachung von Workflows aufgezählt und dargestellt.

Abgeschlossen wird diese Hausarbeit mit einem Fazit. Dieses fasst die Hausarbeit kurz zusammen.

2 Einführung in Apache Airflow

Dieses Kapitel dient dem Zweck Airflow genauer zu durchleuchten. Innerhalb dieses Kapitels wird der Historische Ursprung von Airflow nähergebracht. Zudem werden die einzelnen Architekturkomponenten mit den Kernkonzepten vorgestellt.

2.1 Ursprung

Wie bereits erwähnt ist Airflow eine der beliebtesten Open-Source WfMS. Auch wenn dieses Produkt zur Apache Software Foundation zugehörig ist, wurde Airflow von einem anderen Unternehmen entwickelt und später übertragen.

Anfang der 2010er Jahre hatte Airbnb das Problem, dass diese stetig mehrere und komplexere Workflow in ihren Arbeitsprozessen besitzen. Aus diesem Grund wurde 2014 Airflow als internes Projekt gestartet. Aus diesem Projekt entstand ein zentralisierter und UI-basierter Ansatz für die Erstellung und Planung von Jobs (vgl. Haines 2022, S. 257 f.). Airflow wurde erst 2016 Open-Source, als Airflow von der Apache Software Foundation übernommen wurde (vgl. Singh 2019, S. 67).

Seit dem hat sich Airflow zu einer führenden Lösung für das Workflow-Management entwickelt und wird von einer großen Community von Entwicklern und Benutzern aktiv weiterentwickelt.

2.2 Architektur

Die Architektur von Airflow besteht aus mehreren Komponenten, welche miteinander kommunizieren bzw. Abhängigkeiten besitzen (siehe Abbildung 2.1).

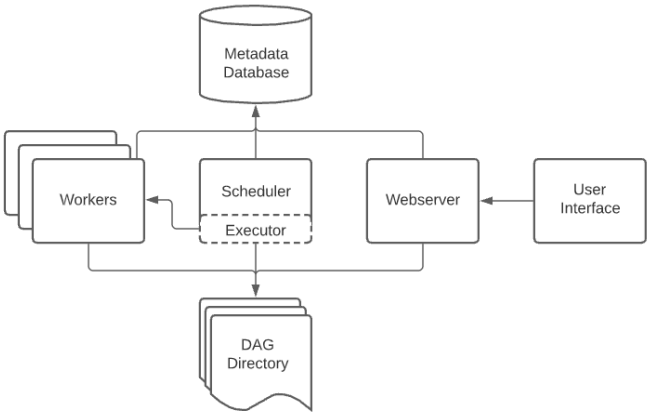

Abbildung 2.1: Airflow-Architektur

(Airflow a)

In diesem Kapitel werden nun die einzelnen Komponenten aus der Abbildung 2.1 genauer beleuchtet.

Der **Webserver** stellt die Benutzeroberfläche mit dem Dashboard bereit. Mit dieser Komponente ist es möglich, die Zustände aller DAGs zu analysieren. Ebenso bietet der Webserver die Möglichkeit Benutzer, Rollen und diverse Konfigurationen zu verwalten (vgl. Villamariona et al.).

Der **Scheduler** ist die zentrale Komponente in der gesamten Architektur. Dieser besteht aus mehreren Threads. Er verwaltet alles, was mit den DAGs zu tun hat (planen der DAG-Durchläufe, Parsing, Speichern von DAGs etc.). Zudem ist dieser verantwortlich für das Worker-Management und den SLAs (vgl. Villamariona et al.).

Zum Scheduler gehört ebenfalls der **Executor** dazu. Dieser ist die tatsächliche Komponente, welche die Aufgaben ausführt. Hierfür gibt es verschiedene Arten von Executor, welche in der Konfigurationsdatei ausgewählt werden kann. Die Möglichen Executor werden im Folgenden aufgezählt (vgl. Villamariona et al.):

1. **Sequential Executor**: Führt nur eine Task-Instanz zur gleichen Zeit aus. Dieser eignet sich nur für Fehlersuchen und für lokale Tests. Zudem ist dieser der einzige Executor, welcher mit SQLite verwendet werden kann, da SQLite keine Mehrfachverbindungen unterstützt (vgl. Villamariona et al.).
2. **Local Executor**: Ist in der Lage, mehrere Aufgaben gleichzeitig auszuführen. Das Limit wird in der Konfiguration festgelegt. Dieser Executor eignet sich wie beim Sequential Executor nur für kleinere Arbeitslasten auf einem einzigen Computer (vgl. Villamariona et al.).
3. **Celery Executor**: Basiert auf Python Celery, welches zur Verarbeitung asynchroner Aufgaben verwendet wird. Celery ist eine asynchrone Task-Queue, die auf verteilte Nachrichtenübermittlungen basiert. Bei dem Celery Executor fügt der Scheduler alle Aufgaben in eine vom User konfigurierte Task-Queue ein. Aus der Queue holt sich der Celery-Worker die Aufgabe und führt sie aus. Nachdem die Ausführung abgeschlossen ist, meldet der Worker den Status der Aufgabe in der Datenbank. Der Scheduler weiß aus der Datenbank, wann eine Aufgabe abgeschlossen ist und führt dann den nächsten Satz von Aufgaben oder Prozesswarnungen aus, je nachdem, was in dem DAG konfiguriert wurde (vgl. Villamariona et al.).
4. **Kubernetes Executor**: Bietet die Möglichkeit an, Airflow-Tasks auf Kubernetes auszuführen. Hierbei startet Kubernetes für jede Aufgabe einen neuen Pod. Kubernetes ist zuständig für den Lebenszyklus des Pods und der Scheduler fragt den Status der Aufgaben von Kubernetes ab. Mit dem Kubernetes Executor wird

jede Aufgabe in einem neuen Pod innerhalb des Kubernetes-Clusters ausgeführt, wodurch die Umgebung für alle Aufgaben isoliert werden kann. Dies verbessert auch die Verwaltung von Abhängigkeiten mithilfe von Docker-Images und erhöht oder verringert die Anzahl der Arbeiter je nach Bedarf (vgl. Villamariona et al.).

Der **Worker** bekommt vom Executor Aufgaben zugewiesen, welche er ausführen soll. Je nach Executor kann es sich um einen separaten Prozess oder einem Container handeln. Diese sind auch für die Ausführung des tatsächlichen Codes verantwortlich, welche in den zugewiesenen Tasks definiert sind. Ebenfalls melden sie deren Status dem Executor zurück (vgl. Lenka 2023).

In der **Metadaten-Datenbank** werden alle Informationen über die DAGs, Tasks und der Ausführungshistorie gespeichert. Hier besteht die Möglichkeit, den Status der Arbeitsabläufe zu erhalten und liefert Daten für das Monitoring und Fehlerbehebung. Airflow unterstützt deshalb diverse Datenbanksysteme wie PostgreSQL, MySQL und SQLite (vgl. Lenka 2023).

2.3 Kernkonzepte

In diesem Unterkapitel werden auf die Kernkonzepte von Airflow genauer darauf eingegangen. Genauer gesagt wird im Folgenden auf die DAGs, die Operatoren und den Tasks eingegangen.

2.3.1 DAG

Ein DAG (Directed Acyclic Graphs) repräsentiert den Workflow als Graph. Der Aufbau eines DAGs besitzt zwei Regeln, welche im Namen beinhaltet ist. Ein DAG muss erstens ein gerichteter Graph sein und zweitens muss dieser azyklisch sein. Was dies bedeutet, wird im folgenden Abschnitt genauer erläutert.

Neben einem gerichteten Graph existieren auch ungerichtete Graphen. Die Abbildung 2.2 zeigt hierbei einen ungerichteten und einen gerichteten Graphen.

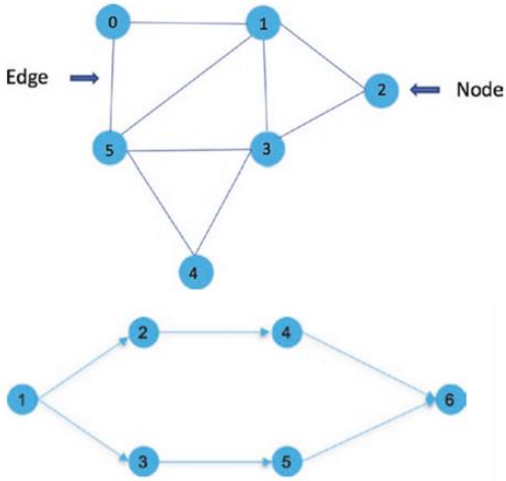

Abbildung 2.2: Ungerichteter (oben) und gerichteter (unten) Graph (vgl. Singh 2019, S. 69f.)

Wenn der obere Graph der Abbildung 2.2 als DAG eingesetzt wird, gibt es keine Informationen, welcher Task von welchem abhängig ist. Bei dem unteren Graphen in der gleichen Abbildung ist dies anders. Hier besitzen die Kanten Richtungen. In dieser Abbildung ist es auch eindeutig, dass der Knoten 2 auf Knoten 1 „warten muss“ damit dieser seinen Task starten kann. Im Kontext der DAGs ist ein Knoten ein Task. Der untere Graph aus der Abbildung 2.2 ist nicht nur ein gerichteter Graph, dieser ist ebenfalls azyklisch (dieser würde als DAG funktionieren).

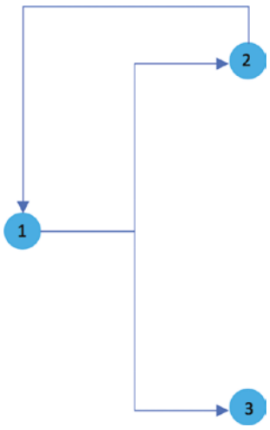

Abbildung 2.3: Zyklischer Graph (vgl. Singh 2019, S. 70)

Die Abbildung 2.3 zeigt hier einen zyklischen Graphen. Dieser würde als DAG nicht funktionieren, da hier der Task 1 auf den Output von Task 2 warten muss. Hier wartet der Task 2 allerdings auch auf den Output von Task 1. Zusammengefasst würde dieser Graph als DAG nicht funktionieren, da die Knoten wegen dem fehlenden Input sich gegenseitig blockieren würden.

2.3.2 Tasks & Operatoren

Wie bereits in 2.3.1 erwähnt, sind die Knoten innerhalb eines DAGs die einzelnen Tasks. Ein Task kann in verschiedenen Formen und Komplexitäten auftauchen. Weit verbreitet sind Python-Skripte, weil die DAG-Skripte ebenfalls in Python geschrieben werden. Ebenfalls weit verbreitet sind Tasks in Form von Shell-Skripten und SQL-Abfragen. Möglich sind auch Cloud-basierte Tasks, wie z. B. Spark-Jobs (vgl. Singh 2019, S. 73).

Tasks werden mit Hilfe von sogenannten Operatoren in einer DAG-Definitionsdatei definiert. Hierzu gibt es eine Menge von Operatoren, welche je nach Task ausgewählt werden müssen. Beispiele hierzu sind die Python-, Bash-, HTTP- oder die SQL-Operatoren (vgl. Singh 2019, S. 73).

Tasks können Abhängigkeiten zueinander haben, die durch spezifizierte Regeln definiert werden. Diese Regeln bestimmen die Ausführungsreihenfolge der Tasks basierend auf ihrem Zustand und dem Erfolg oder Misserfolg anderer Tasks. Mit diesen Abhängigkeiten wird auch definiert, dass der DAG gerichtet und azyklisch ist. Wenn diese Abhängigkeiten nicht definiert werden, werden alle Tasks parallel gestartet.

3 Verwendung von Apache Airflow

Dieses Kapitel geht auf die Verwendung von Airflow ein. Im Folgenden werden drei Unterkapitel vorgestellt, welche sich mit der initialen Installation von Airflow befassen, mit der Erstellung bzw. Ausführung von DAGs. Abschließend wird sich mit dem Monitoring von den Workflows befasst.

3.1 Installation von Apache Airflow

Um Airflow lokal auf einen Computer zu installieren bietet Apache mehrere Möglichkeiten an wie z. B. PyPI, Helm Chart for Airflow oder Docker Images (vgl. Airflow c). In diesem Kapitel wird auf die Installation mit Docker eingegangen.

Bei der Installation mit Docker gibt es ebenfalls mehrere Herangehensweisen. Die erste Variante ist ein Aufsetzen eines eigenen Docker-Images mit allen Abhängigkeiten und Komponenten. Hier besteht die Gefahr, dass bei der Erstellung des eigenen Docker-Images Abhängigkeiten oder Komponenten vergessen werden, sodass das Aufsetzen sich potenziell zeitlich in die Länge ziehen kann. Die zweite und auch vom Verfasser empfohlene Möglichkeit ist ein Aufsetzen mit dem im Docker Hub verfügbaren Docker-Image (vgl. Singh 2019, S. 74).

Die folgenden zwei Zeilen laden das bereits erwähnte Docker-Image vom Docker Hub herunter und starten das Docker-Image auf dem Port 8080:

```
docker pull puckel/docker-airflow
docker run -d -p 8080:8080 puckel/docker-airflow webserver
```

Abbildung 3.1: Docker Funktionen, um Airflow zu installieren & starten

Nachdem diese beiden Zeilen ausgeführt wurden, kann unter der URL: `http://127.0.0.1:8080` die Airflow-Benutzeroberfläche angesehen werden (siehe Abbildung 3.2 als Beispiel).

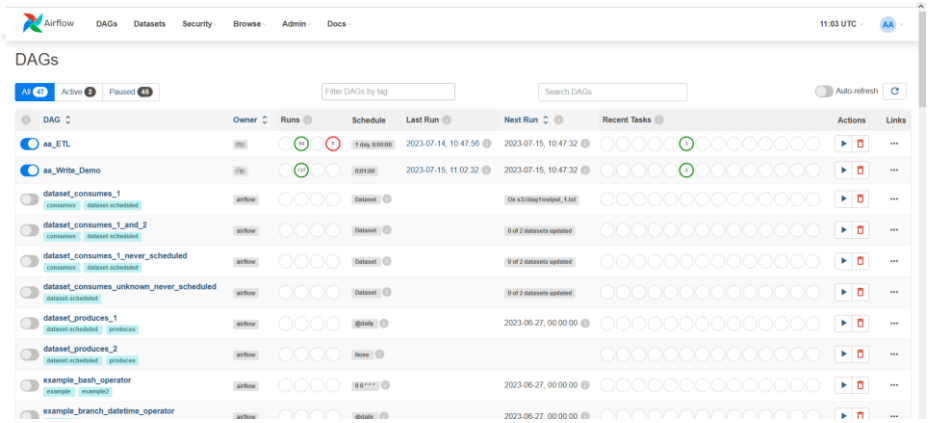

Abbildung 3.2: Airflow-Benutzeroberfläche

In der Abbildung 3.2 ist die Auflistung aller DAGs zu sehen. In dieser Tabelle stellt jede Zeile ein DAG dar. Bis auf die ersten zwei Zeilen sind alle DAGs bei der Installation automatisch dabei. Bei jedem DAG werden mehrere Informationen dargestellt: der Name, der Owner, die Status von den vergangenen DAG-Durchläufen, den Schedule, den letzten und nächsten Durchlauf und die Status der aktuellen bzw. kommenden Tasks. Ebenfalls gibt es in der Spalte Actions ein Button, um den DAG manuell zu starten und ein Button, um den DAG zu löschen. Bei der Spalte Links sind, wie der Name bereits suggeriert, mehrere Links hinterlegt, welche zu den Monitoring-Reiter der DAGs führt. Diese Reiter werden in Kapitel 3.3 genauer erläutert.

3.2 Erstellung & Ausführung von DAGs

Im bisherigen Verlauf dieses Dokumentes ist es bereits an einigen Stellen implizit deutlich geworden, dass die Erstellung und die Ausführung der DAGs der Kernprozess von Airflow ist.

Um ein DAG in Airflow zu erstellen, wird eine DAG-Definitionsdatei benötigt. Diese ist in Python geschrieben und beinhaltet alle nötigen Informationen, welche Airflow benötigt, damit dieser DAG funktionieren kann. Konkret sind die folgenden fünf Schritte nötig (vgl. Singh 2019, S. 73):

1. Importieren der benötigten Module und Python-Libraries.
2. Deklarieren von Standardargumenten: Hier können diverse Argumente übergeben werden wie z. B.: den Besitzer des DAGs, das Startdatum, die Anzahl an Wiederversuchen, falls dein Durchgang fehlschlägt, oder ein SLA. Wenn hier ein SLA angegeben wird, welche bei einem Durchlauf nicht erfüllt wird, wird dieser Durchgang in einem Reiter mit allen anderen nicht erfüllten Durchläufen dokumentiert.
3. Instanziieren des tatsächlichen DAG-Objekts: In diesem Objekt wird u. a. der Name angegeben, eine optionale Beschreibung oder das Schedule-Intervall. Ebenfalls werden die Standardargumente aus Punkt 2 hier ebenfalls übergeben.
4. Definieren aller Tasks: Hier wird für die Tasks der Operator angegeben, die Task-ID, den DAG und ebenfalls das Skript, die Query etc.
5. Definieren der Reihenfolge der Ausführung der Tasks: Hier werden die Abhängigkeiten der Tasks definiert. Hier ist zu beachten, dass der zu definierende DAG keine zyklischen Abhängigkeiten besitzt.

Um diese fünf Schritte besser zu verinnerlichen, zeigt die Abbildung 3.3 eine DAG-Definitionsdatei.

```
1  from datetime import datetime
2  from datetime import timedelta
3
4  from airflow import DAG
5  from airflow.decorators import task
6  from airflow.operators.bash import BashOperator
7
8  default_args = {
9      'owner': "[redacted]",
10     'start_date': datetime(2023, 6, 24),
11     'sla': timedelta(seconds=1)
12     }
13 # A DAG represents a workflow, a collection of tasks
14 with DAG(
15     "aa_Write_Demo",
16     schedule=timedelta(minutes=1),
17     default_args=default_args
18     ) as dag:
19
20     # Tasks are represented as operators
21     hello = BashOperator(task_id="hello", bash_command="echo Hello my task is writing")
22
23     @task()
24     def airflow():
25         print("hello from task airflow")
26
27     # Set dependencies between tasks
28     hello >> airflow()
```

Abbildung 3.3: Beispielversion einer DAG-Definitionsdatei

In den Zeilen 1-6 sind hier die Importe zu sehen (Schritt 1). Zu sehen ist, dass für alle Airflow-Objekte/Funktionen ebenfalls Importe benötigt werden. Hier bietet Airflow ebenfalls Packages von Drittanbietern wie z. B.: Amazon, Atlassian Jira, Docker, Elasticsearch, Facebook, FTP, GitHub, Google, gRPC, Kubernetes, Microsoft Azure, SSH oder andere Cloud-Produkte von Apache wie Flink, HDFS, Hive, Kafka oder Spark (vgl. Airflow b).

Im Zeilenabschnitt 8-12 werden die Standardargumente übergeben (Schritt 2). In diesem Beispiel wurde neben dem DAG-Besitzer auch ein Startdatum vom 24.06.2023 festgesetzt. Ebenfalls wurde hier ein SLA angegeben, welcher besagt, dass jeder Durchlauf, welcher länger als eine Sekunde dauert, als nicht erfüllt kategorisiert wird.

Der Schritt 3 wird in der Abbildung 3.3 in den Zeilen 14-18 realisiert. Hier wird der Name `aa_Write_Demo` festgelegt, einen Ausführungsintervall von einer Minute definiert und abschließend werden die Standardargumente übergeben.

Der vierte Schritt ist das definieren der Tasks. Dies geschieht in den Zeilen 21 und 24/25. In der Zeile 21 wird ein Bash-Befehl als Task definiert. Dessen Aufgabe, den String „Hello my task is writing“ zu schreiben. Die Zeilen 24/25 ist eine Python-Funktion. Dieser schreibt ebenfalls ein String („hello from task airflow“). Würden diese beiden Befehle/Funktionen in einer Befehlszeile ausgeführt werden, würden die

Output-Strings ebenfalls in der Befehlszeile ausgegeben werden. Bei Airflow werden diese Strings in den dazugehörigen Log-Dateien geschrieben.

Der fünfte und letzte Schritt befindet sich in der Zeile 28. Hier werden die Aufgabenabläufe definiert. Hier wird beschrieben, dass der Task `airflow()` wartet, bis der Task `hello` abgeschlossen ist. Würde hier die Zeile nicht angegeben werden, besitzt dieser DAG dann keine Kanten. Dies würde bedeuten, dass beide Tasks gleichzeitig starten werden.

3.3 Monitoring der Workflows

Ein wesentlicher Aspekt der Verwendung von Apache Airflow ist die Überwachung der Workflows und die Behandlung von Fehlern. Apache Airflow bietet diverse Mechanismen zur Überwachung des Fortschritts von Workflows und zur Anzeige von Statusinformationen. Im Folgenden wird auf die relevantesten Funktionen für das Workflow-Monitoring aufgezählt.

3.3.1 Reiter „Grid“

Die erste Analyse Funktion ist der Grid-Reiter des DAGs (siehe Abbildung 3.4). Mit dieser ist es möglich, detaillierte Informationen über vergangene Durchläufe zu erhalten. Hier besteht ebenfalls die Möglichkeit, diese Informationen im Kontext des definierten Gesamtzeitraumes oder für einen einzelnen Durchlauf anzeigen zu lassen.

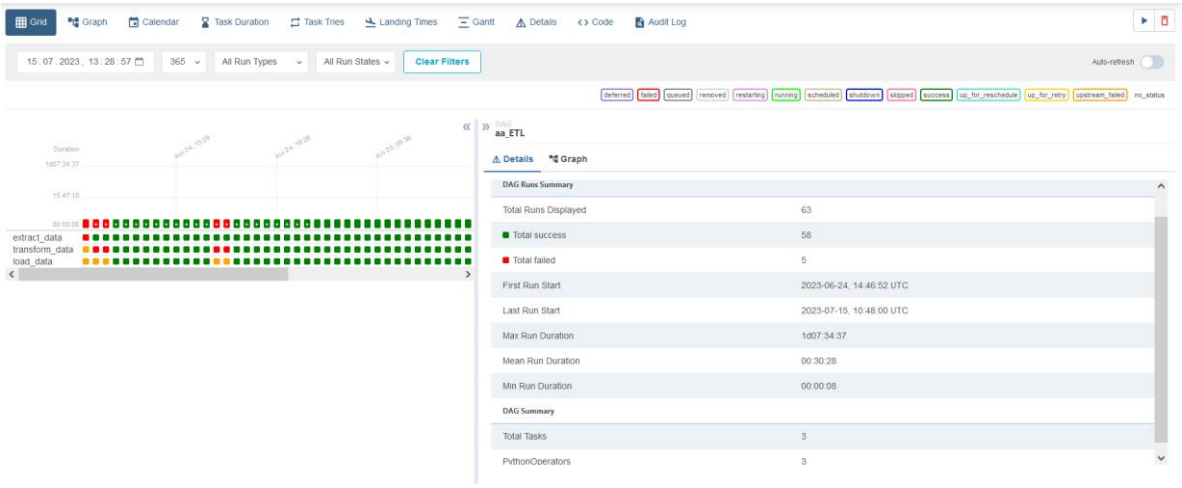

Abbildung 3.4: Reiter Grid eines DAGs

Auf der linken Seite der Abbildung 3.4 sind eine Grafik zu sehen, welche den Status jedes Tasks für jeden Durchlauf wiedergibt. Hierbei ist der obere Teil dieser Grafik ein Säulendiagramm, welche die Dauer der DAG-Durchläufe wiedergibt.

Unterhalb sind die einzelnen Tasks aufgezählt. Die grüne Markierung eines Tasks bedeutet, dass der Task erfolgreich durchgeführt wurde. Rot heißt, dass der Task fehlgeschlagen ist. Orange heißt `upstream_failed`. Dies bedeutet, dass dieser Task nicht gestartet wurde, weil der Task, welcher laut der definierten Abhängigkeit davor ausgeführt werden muss, fehlgeschlagen ist.

3.3.2 Reiter „Task Duration“

Für genauere Zeitanalysen eignet sich der Grid-Reiter nicht. Besser geeignet wäre hierfür der Reiter Task Duration (siehe Abbildung 3.5). Durch die Verwendung des Task Duration-Reiters können Benutzer die Performance ihres Workflows überwachen und Engpässe oder Performance-Probleme identifizieren. Eine längere Durchlaufzeit eines bestimmten Tasks kann auf Probleme in der Task-Ausführung oder Abhängigkeiten hinweisen, die weiter untersucht werden müssen.

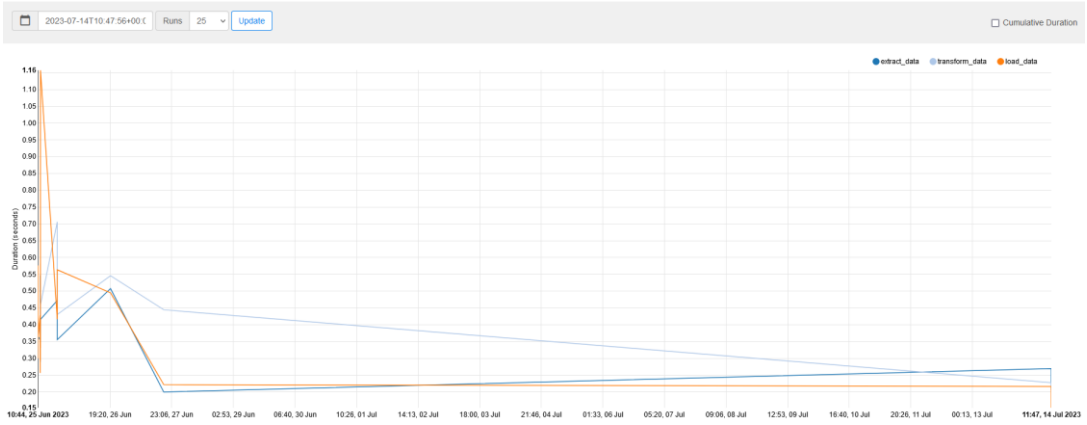

Abbildung 3.5: Reiter Task Duration eines DAGs

Das Liniendiagramm visualisiert die Durchlaufzeiten der einzelnen Tasks über die Zeit. Jeder Task wird als separate Linie dargestellt, wobei die y-Achse die Zeit und die x-Achse den Zeitstrahl der Workflows darstellt. Das Diagramm ermöglicht es den Benutzern, die Durchlaufzeiten der Tasks im zeitlichen Verlauf zu verfolgen und potenzielle Engpässe oder Auffälligkeiten zu identifizieren.

Eine besonders nützliche Funktion dieses Reiters ist die Möglichkeit, zwischen der nicht-kumulierten (zu sehen in Abbildung 3.5) und einer kumulierten Ansicht der Durchlaufzeiten zu wechseln.

In der nicht-kumulierten Ansicht werden die Durchlaufzeiten der einzelnen Tasks separat dargestellt, wodurch Benutzer genauere Informationen über die individuellen Laufzeiten erhalten. Durch das Anklicken der entsprechenden Schaltfläche

oben rechts im Reiter kann die Ansicht zwischen nicht-kumuliert und kumuliert gewechselt werden. In der kumulierten Ansicht werden die Durchlaufzeiten aller vorherigen Tasks bis zum aktuellen Task summiert dargestellt, was hilfreich sein kann, um die Gesamtzeit eines Workflows zu analysieren.

3.3.3 Reiter „Task Tries“

Innerhalb dieses Dokumentes wurde es bereits kurz erwähnt, dass bei den Standardargumenten ein Parameter für die Anzahl an Wiederversuchen angeben kann. Dies hat den Grund, dass es viele Use Cases gibt, bei denen die Tasks mit einer Datenbank kommunizieren oder mit einem Server. Hierbei ist es immer möglich, dass für einige Sekunden die Kommunikation nicht aufgebaut werden kann. Deshalb ist es ratsam, bei solchen Tasks immer ein Wiederholungsmechanismus einzubauen, vor allem, wenn der DAG in Relation zum Durchschnitt sehr groß ist. Die Abbildung 3.6 zeigt hierbei den Reiter, welcher die Anzahl an Versuchen darstellt.

Abbildung 3.6: Reiter Task Tries eines DAGs

Das Liniendiagramm stellt die Anzahl der Versuche für jeden Task über die Zeit dar. Jeder Task wird als separate Linie dargestellt, wobei die y-Achse die Anzahl der Versuche und die x-Achse den Zeitstrahl der Workflows darstellt. Die Visualisierung der Anzahl der Versuche ermöglicht es Benutzern, die Stabilität und Zuverlässigkeit ihrer Tasks zu überwachen. Wenn ein Task eine hohe Anzahl von Versuchen aufweist, kann dies auf mögliche Probleme bei der Kommunikation mit einer externen Ressource oder bei der Verarbeitung der Aufgabe hinweisen. Durch die Überwachung der Anzahl der Versuche können Benutzer Engpässe oder Probleme identifizieren und entsprechende Maßnahmen ergreifen, um die Ausführung der Tasks zu verbessern.

3.3.4 Log

Das letzte Monitoring Werkzeug, welches in diesem Dokument erläutert wird, sind die Logs. Logs sind Protokolle, welche Events von Betriebssystemen, Softwareprodukten etc. aufzeichnet. Der relevanteste Nutzen, welche die Logs mit sich bringen, ist die Fehleranalyse. Hiermit ist es möglich, Systemprobleme zu diagnostizieren, Sicherheitsvorfälle zu untersuchen oder Performanceanalysen durchzuführen.

Airflow stellt ebenfalls ein Log zur Verfügung. Allerding ist dies keine simple Textdatei. Dieser Airflow-Log besitzt eine Ordnerstruktur, um bei möglichen Problemen gezieltere Analysen starten zu können (vereinfachte Version ist in der Abbildung 3.7 zu sehen).

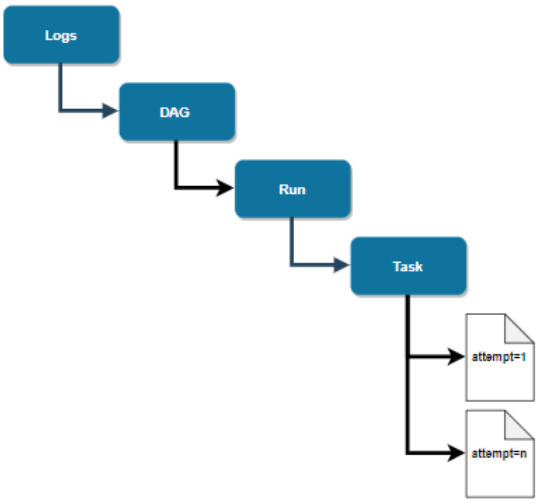

Abbildung 3.7: Vereinfachte Ordnerstruktur des Logs

Alle Logs sind in dem übergeordneten Logs-Ordner. Eine Ebene tiefer werden erhält jeder DAG einen eigenen Ordner, welche den Namen `dag_id=`*`(Platzhalter für den DAG-Namen)* besitzt. Der Scheduler besitzt ebenfalls einen eigenen Ordner auf dieser Ebene.

Wieder eine Ebene tiefer, werden die Logs ebenfalls nach den einzelnen Durchläufen sortiert. Hier ist der Name dieser Ordner `run_id=scheduled_`*`(Timestamp)* oder `run_id=manual_`*`(Timestamp)`*. Dies unterscheidet sich je nachdem, ob dieser Durchlauf manuell gestartet wurde ober ob dieser DAG durch Schedule automatisch gestartet wurde.

Nachdem die Logs nach dem Durchlauf sortiert wurden. Wird innerhalb des Durchlauf-Ordners die Logs nach den jeweiligen Tasks sortiert. Diese Ordner werden nach dem Schema `task_id=`*`(Taskname)* generiert.

Abschließend werden innerhalb dieser Task-Ordner die Log-Dateien geschrieben. Innerhalb dieses Ordners ist mindestens eine Log-Datei enthalten. Falls der erste Versuch fehlschlug und mehrere Wiederversuche eingestellt wurden, werden automatisch die neuen Anläufe in neuen Log-Dateien geschrieben (Namensschema: `attempt=1`, …, `attempt=n`).

Innerhalb des Logs sind mehrere Informationen enthalten, wie z. B.:

- Angabe, ob alle Abhängigkeiten erfüllt sind
- Benachrichtigung, welcher Anlauf gestartet wurde und was die maximale Anlaufzahl ist
- Angabe, welcher Operator zu welchem Zeitpunkt gestartet wurde
- Angabe, ob der Befehl erfolgreich ausgeführt wurde
- Benachrichtigung, wie der Task markiert wurde bei Airflow (`failed`, `success`; siehe Kapitel 3.3.1)
- Falls ein Task nach diesem Task folgt, wird dies ebenfalls hier angegeben
- Optional werden hier auch alle Nachrichten, welche innerhalb des Tasks mit einem `echo` oder `print()` generiert wurden, hier ausgegeben

Zusammenfassend ist der Log ebenfalls ein sehr genaues Monitoring-Werkzeug für die Maintenance aller DAGs.

4 Fazit

Zusammenfassend lässt sich aus dem Dokument herausnehmen, dass Workflow-Management-Systeme wichtige Werkzeuge sind, um komplexe Geschäftsprozesse zu automatisieren. Hierbei ist Apache Airflow eine mögliche Open-Source-Lösung. Airflow bietet eine grafische Benutzeroberfläche und ermögliche damit eine einfache Erstellung, Verwaltung und Überwachung von Workflows.

Airflows Architektur besteht aus mehreren Komponenten wie dem Scheduler, dem Executor, dem Webserver oder der Metadaten-Datenbank. Die Workflows werden in Airflow als DAG (Directed Acyclic Graphs) dargestellt und die einzelnen Tasks werden mit Operatoren in Python realisiert.

Um Airflow lokal auf einen Computer installieren zu können, bietet Apache mehrere Installationsmöglichkeiten an. Eine der schnellsten Möglichkeiten ist die Installation des Docker-Images von Airflow.

Für die Überwachung der einzelnen Workflows bietet Airflow mehrere Reiter innerhalb der DAGs an, wie z. B. Grid, Task Duration oder Task Tries. Ebenfalls sind die Logs für eine detailliertere Analyse sehr nützlich.

Insgesamt bietet Apache Airflow eine leistungsstarke Plattform für das Workflow-Management, die es ermöglicht, komplexe Geschäftsprozesse effektiv zu koordinieren und zu überwachen. Airflow ist eine gute Lösung für Unternehmen, die ihre Workflows verbessern möchten, da es eine flexible Architektur und viele Überwachungsfunktionen hat.

Abbildungsverzeichnis

Abbildung	Titel	Seite
2.1	Airflow Architektur	3
2.2	Ungerichteter (oben) und gerichteter (unten) Graph	6
2.3	Zyklischer Grap	6
3.1	Docker Funktionen, um Airflow zu installieren & starten	8
3.2	Airflow-Benutzeroberfläche	8
3.3	Beispielversion einer DAG-Definitionsdatei	10
3.4	Reiter Grid eines DAGs	11
3.5	Reiter Task Duration eines DAGs	12
3.6	Reiter Task Tries eines DAGs	13
3.7	Vereinfachte Ordnerstruktur des Logs	14

Literaturverzeichnis

Airflow a: Architecture Overview. Apache Software Foundation. Online verfügbar unter https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/overview.html, zuletzt geprüft am 14.07.2023.

Airflow b: Documentation. Online verfügbar unter https://airflow.apache.org/docs/, zuletzt geprüft am 16.07.2023.

Airflow c: Installation of Airflow. Online verfügbar unter https://airflow.apache.org/docs/apache-airflow/stable/installation/index.html#using-official-airflow-helm-chart, zuletzt geprüft am 15.07.2023.

Haines, Scott (2022): Modern Data Engineering with Apache Spark. A Hands-On Guide for Building Mission-Critical Streaming Applications. 1st ed. 2022. Berkeley, CA: Apress; Imprint Apress.

Lenka, Binaya Kumar (2023): Airflow Architecture. Online verfügbar unter https://medium.com/@binayalenka/airflow-architecture-667f1cc613e8, zuletzt aktualisiert am 02.04.2023, zuletzt geprüft am 14.07.2023.

Luber, Stefan; Litzel, Nico (2020): Was ist Apache Airflow? Online verfügbar unter https://www.bigdata-insider.de/was-ist-apache-airflow-a-948609/, zuletzt aktualisiert am 17.07.2020, zuletzt geprüft am 13.07.2023.

Singh, Pramod (2019): Learn PySpark. Build Python-based Machine Learning and Deep Learning Models. New York: Apress (Springer eBook Collection).

Villamariona, Jorge; Chattaraj, Joy Lal; Shrivastava, Prateek: Understand Apache Airflow's Modular Architecture. Online verfügbar unter https://www-qubole-com.translate.goog/tech-blog/understand-apache-airflows-modular-architecture?_x_tr_sl=en&_x_tr_tl=de&_x_tr_hl=de&_x_tr_pto=sc, zuletzt geprüft am 14.07.2023.

Workflow-Management-Systeme (2006). In: *Z Control Manag* 50 (4), S. 199–200.

Selbstständigkeitserklärung