Hausarbeit

Cloud

Apache Airflow

|  |  |
| --- | --- |
| Verfasser: | Albrecht, Florian |
| Arbeit eingereicht am: | 17.07.2023 |
| Prüfer: | Prof. Dr. Blochinger, Wolfgang |
| Fach: | Cloud & Big Data Technologien |
| Studiengang: | Wirtschaftsinformatik (M.Sc.) |
| Fachsemester: | 1 |

Inhaltsverzeichnis

[1 Einleitung 1](#_Toc140241768)

[1.1 Motivation 1](#_Toc140241769)

[1.2 Einführung Apache Airflow 1](#_Toc140241770)

[1.3 Zielsetzung & Struktur 2](#_Toc140241771)

[2 Einführung in Apache Airflow 3](#_Toc140241772)

[2.1 Ursprung 3](#_Toc140241773)

[2.2 Architektur 3](#_Toc140241774)

[2.3 Kernkonzepte 5](#_Toc140241775)

[2.3.1 DAG 5](#_Toc140241776)

[2.3.2 Tasks & Operatoren 7](#_Toc140241777)

[2.3.3 Erstellen einer DAG-Definitionsdatei 7](#_Toc140241778)

[3 Verwendung von Apache Airflow 8](#_Toc140241779)

[4 Anwendungsbeispiele und Best Practices 9](#_Toc140241780)

[5 Vergleich mit anderen WfMS wie Luigi, Azkaban, etc. 10](#_Toc140241781)

[6 Herausforderungen und zukünftige Entwicklungen 11](#_Toc140241782)

[7 Fazit 12](#_Toc140241783)

[Literaturverzeichnis III](#_Toc140241784)

# Einleitung

Dieses Kapitel hat den Zweck, diese Hausarbeit einzuführen. Zu Beginn wird der Verwendungszweck von Workflow-Management-Systemen motiviert.

## Motivation

In der Regel teilen 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 eventuell zu steuern. Es wird immer wieder versucht, WfMS im Controlling-Bereich zu nutzen, um stark standardisierte Prozesse zu unterstützen. WfMS sind Softwaresysteme, welche die Durchführung von Aufgaben durch verschiedene Bearbeiter 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).

## 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 werden wir uns eingehend mit Apache Airflow befassen und seine Funktionen, Architektur und Verwendung untersuchen. Wir werden auch die Vor- und Nachteile von Apache Airflow im Vergleich zu anderen Workflow-Management-Systemen analysieren und prüfen, wie es in verschiedenen Anwendungsfällen eingesetzt wird.

## Zielsetzung & Struktur

Das Ziel dieser Ausarbeitung ist es, Apache Airflow als WfMS näher zu bringen. Um dies zu erreichen wird die Verwendung von Airflow erläutert anhand von verschiedenen Branchen. 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 Graph) und den dazugehörigen Tasks.

Kapitel 3 hingegen beschäftigt sich mit der initialen Installation & Konfiguration von Airflow. Ebenfalls werden hier wichtige Funktionen, wie z. B. die Überwachung von Workflows aufgezählt und dargestellt. Abschließend werden mögliche Anbindung bzgl. anderen Programmiersprachen, Cloud-Services etc. aufgezählt.

Das Kapitel 4 beschäftigt sich mit praktischen Anwendungsfällen. Hier werden Use Cases von verschiedenen Branchen aufgezählt. Ebenso werden Best Practices für DAGs aufgestellt, um die Entwicklung effizienter gestalten zu können.

Kapitel 5 befasst sich im Kern mit dem Vergleich von anderen WfMS. Hier wird Airflow u. a. mit Luigi uns Azkaban verglichen.

Im sechsten Kapitel werden mögliche Herausforderungen bei einer Nutzung von Airflow evaluiert. Zusätzlich werden mögliche Verbesserungen für dieses System aufgezählt.

Abgeschlossen wird diese Hausarbeit mit einem Fazit. Dieses fasst die Hausarbeit kurz zusammen. Ebenfalls werden die großen Vor- und Nachteile explizit aufgezählt.

# 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.

## Ursprung

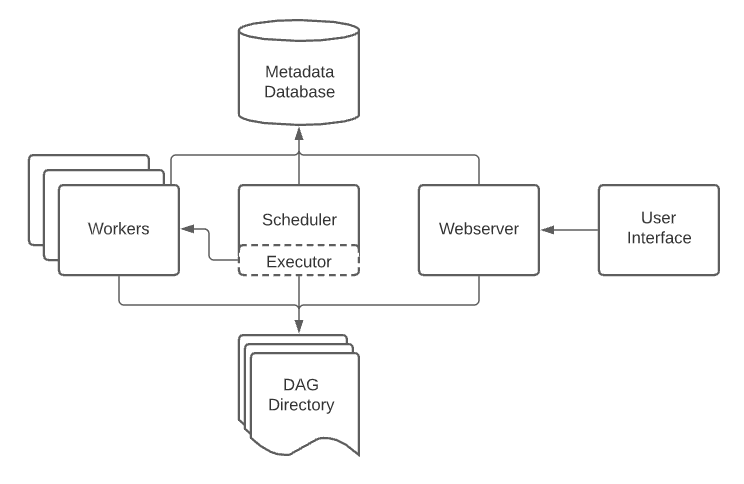
Wie bereits erwähnt ist Airflow eine der beliebtesten Open-Source WfMS. Auch wenn dieses Produkt zu Apache Software Foundation zugehörig ist, wurde Airflow von einem anderen Unternehmen entwickelt und später übertragen.

Anfang der 2010er Jahre hatte Airbnb das Problem, dass diese stetig mehr 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.

## 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)

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, welche aus mehreren Threads besteht, in der gesamten Architektur. Dieser 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 Rechner (vgl. Villamariona et al.).
3. **Celery Ecexutor**: 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 der DAG konfiguriert wurde (vgl. Villamariona et al.).
4. **Kubernetes Executor**: Bietet die Möglichkeit an, Airflow-Tasks auf Kubernetes auszuführen (hierbei würde Kubernetes für jede Aufgabe einen neuen Pod starten). Hier ist Kubernetes 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 - sogar bis hin zu null Arbeiterprozessen (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 Aufgaben 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).

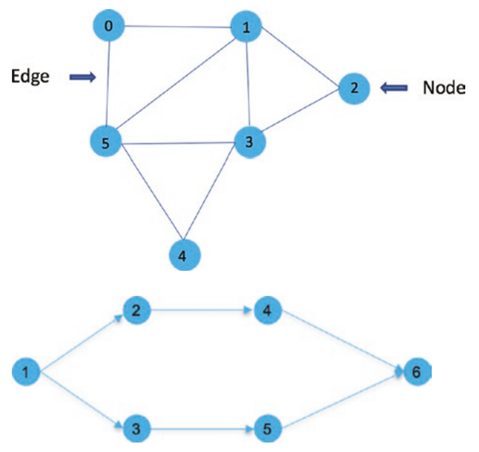
## Kernkonzepte

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

### DAG

Ein DAG (Directed Acyclic Graph) 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.

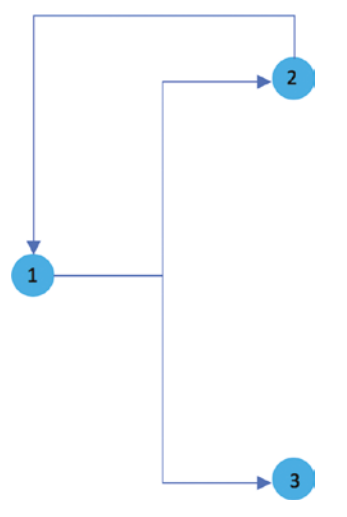
Neben einem gerichteten Graph existieren auch ungerichtete Graphen. Die Abbildung 2.2 zeigt hierbei einen ungerichteten und einen gerichteten Graphen.



**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, welche Aufgabe von welcher 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 seine Aufgabe 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 ebenfalls 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 nicht als DAG funktionieren, da die Knoten wegen fehlenden Input blockieren würden.

### 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. Beispiel hierzu sind die Python-, Bash-, HTTP- oder die SQL-Operatoren (vgl. Singh 2019, S. 73).

### Erstellen einer DAG-Definitionsdatei

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. Instanziierung 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: Hier werden die Abhängigkeiten der Tasks definiert. Hier ist zu beachten, dass der zu definierende DAG keine zyklischen Abhängigkeiten besitzt.

# Verwendung von Apache Airflow

Installation und Konfiguration von Apache Airflow

Erstellung und Ausführung von DAGs

Überwachung, Fehlerbehandlung und Wiederherstellung von Workflows

+ Cloud anbindungen und so

# Anwendungsbeispiele und Best Practices

Praktische Anwendungsfälle von Apache Airflow in verschiedenen Branchen (Data Engineering, Data Science, etc.)

Best Practices für die effiziente Gestaltung und Organisation von DAGs

Performance-Optimierung und Fehlervermeidung in Apache Airflow

# Vergleich mit anderen WfMS wie Luigi, Azkaban, etc.

# Herausforderungen und zukünftige Entwicklungen

Herausforderungen bei der Verwendung von Apache Airflow (Skalierbarkeit, Monitoring, etc.)

Aktuelle Trends und zukünftige Entwicklungen von Apache Airflow

Potenzielle Erweiterungen und Verbesserungen der Plattform

# Fazit

Zusammenfassung der wichtigsten Erkenntnisse und Ergebnisse der Hausarbeit

Bewertung der Vor- und Nachteile von Apache Airflow

Ausblick auf zukünftige Anwendungen und Relevanz von Apache Airflow

Literaturverzeichnis

Airflow: 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.

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.