

# Apache Airflow

Constantin Dörler, Alexander Hennecke, Simon Hermansdorfer,  
Christian Pritzl, Dietrich Wall  
Fakultät für Informatik

WS 2020/21

## Abstract

In der Workflow Verarbeitung gewinnt Apache Airflow bei vielen Unternehmen an Bedeutung. Die einfache Konfigurierbarkeit der Aufgaben ermöglicht für die Unternehmen viele Use Cases, die schnell und ohne große Einarbeitung umgesetzt werden können. Das Ecosystem mit der Integration in Kubernetes ermöglicht es zusätzlich vielen Unternehmen, eine Airflow Infrastruktur einfach auf beliebigen Cloud Anbietern zu verteilen. In der folgenden Arbeit wird Apache Airflow als Workflow Managementsystem näher betrachtet. Dies wird anhand der Architektur und Funktionsweise analysiert. Dabei werden die Grundlagen zum einfacheren Verständnis erläutert und anhand eines realen Use Cases aufgezeigt. Hierfür werden die aktuellen Corona Daten aus der RKI Datenbank über eine API für einen Landkreis geladen und verarbeitet. Anschließend werden Informationen zu den aktuellen Corona Fällen der letzten 7 Tage, die Anzahl genesener Personen, akut infizierter und der Todesfälle an hinterlegte Benutzer gesendet. Dies geschieht simultan über einen Telegram Bot sowie über E-Mail. Anhand dieser Demo Applikation wird ein Überblick über den Entwicklungsprozess sowie der Weboberfläche von Apache Airflow gegeben.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>3</b>
<b>2</b>	<b>Apache Airflow</b>	<b>3</b>
2.1	Funktionsweise . . . . .	3
2.2	Use Cases . . . . .	6
2.3	Architektur . . . . .	7
2.3.1	Webserver . . . . .	7
2.3.2	Scheduler . . . . .	8
2.3.3	Executer . . . . .	8
2.3.4	Konzepte . . . . .	10
2.4	Kubernetes . . . . .	10
2.4.1	Airflow on Kubernetes . . . . .	11
2.4.2	Kubernetes Worker . . . . .	11
<b>3</b>	<b>Demoapplikation CoronaBot</b>	<b>12</b>
3.1	Verzeichnisstruktur . . . . .	12
3.2	DAG-Entwicklung . . . . .	14
3.3	DAG Management . . . . .	14
<b>4</b>	<b>Fazit</b>	<b>19</b>

# 1 Einleitung

Apache Airflow ist eine Software Plattform um effizient Workflows zu managen. Entwickelt wurde sie 2014 von Maxime Beauchemin bei Airbnb und als Open Source Projekt auf GitHub veröffentlicht. Bereits im Jahr 2016 wurde das Projekt in das Apache Software Foundation Incubator Programm aufgenommen und unter der Apache License Version 2.0 geführt. Des weiteren ist Airflow im Jahr 2019 in die Riege der Apache Top-Level Projekte aufgestiegen. Zum Managen der Workflows gehört die Erstellung, Verwaltung und die Überwachung einzelner sowie zusammenhängender Arbeitsabläufe. Die Workflows werden dabei als azyklische gerichtete Graphen über ein Python Skript definiert und können über eine webbasierte Oberfläche verwaltet und überwacht werden. [Aird]

## 2 Apache Airflow

Im folgenden wird nun ein kleiner Einblick in die Apache Airflow Welt gegeben. Hierzu wird in 2.1 die grundlegende Funktionsweise aufgezeigt und wichtige Begriffe definiert. Um die vielseitige Einsetzbarkeit von Apache Airflow zu verstehen wird in 2.2 ein kleiner Ausschnitt zu möglichen Anwendungsgebieten beschrieben. Weiterhin wird in 2.3 ein Einblick in die tiefere Architektur und Konzeptionierung der Software gewährt. Das mittlerweile breitgefächerte Ecosystem an Tools und Services wird in 2.4 beleuchtet.

### 2.1 Funktionsweise

Um die Funktionsweise von Apache Airflow zu verstehen, werden im folgenden mehrere Begrifflichkeiten definiert. Zu diesen Begriffen gehören **Workflow**, **Directed Acyclic Graph (DAG)**, **Operator**, **Task**, **Scheduler**, **Executor** und **Worker**.

Die wichtigste Definition stellt das Wort **Workflow** dar. Im Kontext von Apache Airflow wird damit ein in Python programmierter Arbeitsablauf beschrieben. Dieser wird sowohl zeitlich als auch eventbasiert gesteuert. Weiterhin werden diese Workflows in der Airflow Umgebung durch **DAGs** repräsentiert. Dabei können wie in Abbildung 1 zu sehen, die Knoten als Tasks und die Verbindungen als Abhängigkeiten zwischen den Aufgaben gesehen werden. Um eine lose Kopplung zu den auszuführenden Aufgaben zu erreichen, ist ein DAG nur für die zeitlich korrekte Ausführung, die Einhaltung der spezifizierten Reihenfolge und die Bearbeitung unerwarteter Probleme verantwortlich. Dies vereinfacht die Einbringung von Änderungen in die Workflows sowie das Handling auftretender Probleme in den Tasks. Weiterhin können DAGs zusätzliche Sub-DAGs enthalten. In diesem Fall wird in Abbildung 1 von „section“ bzw. „inner\_section“ gesprochen. Start und Endpunkt können hier als fest definierte Trigger „start\_date“ vgl. Codebeispiel 3 Zeile 2 und das Ende eines komplett durchlaufenen Workflows verstanden werden.

**Hinweis:** Da es sich um azyklische Graphen handelt, werden von Apache Airflow keine Schleifen oder Mehrfachreferenzierungen in den programmierten DAGs gestattet.

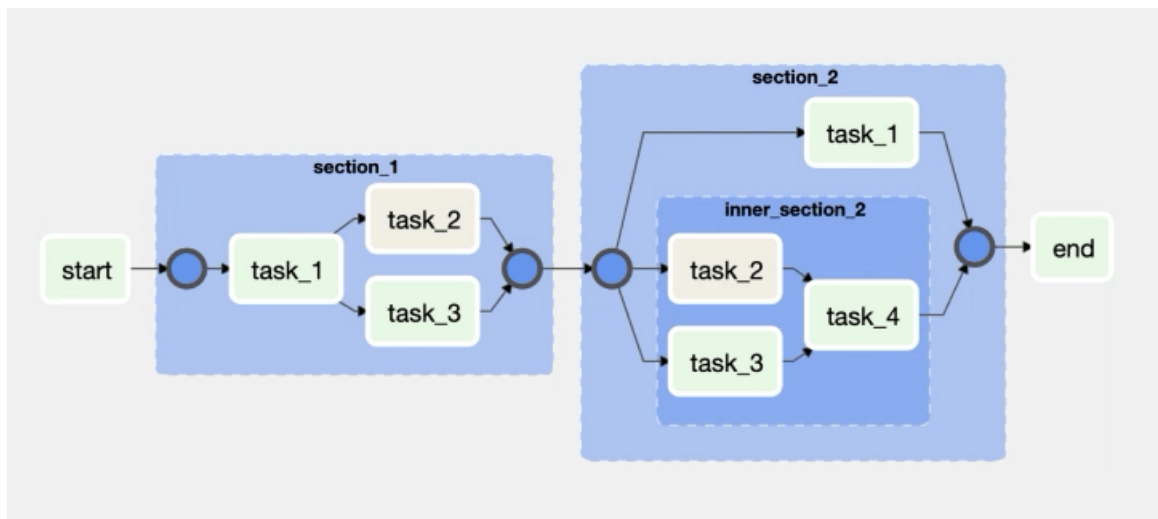


Abbildung 1: Beispiel Darstellung eines DAG mit weiteren Sub-DAGs (sections) [Aird]

Jeder einzelne DAG wird gemäß Codebeispiel 3 in Python implementiert und enthält eine **DAG-Definition**, **Operatoren** und die **Beziehungen** zwischen den Operatoren. [Goo] Diese Operatoren stellen dabei die Schnittstelle zu unterschiedlichen Programmiersprachen, Datenbanken und vielen weiteren nützlichen Systemen dar. Anzuführen wäre hier beispielsweise ein Python-Operator um Python Code oder ein Bash-Operator um Bash Befehle in einem Task auszuführen. Für die Demoapplikation wird unter Anderem auf einen Email-Operator zurückgegriffen. **Tasks** werden in diesem Zusammenhang als die Python Implementierung eines Operators bezeichnet. Die Beziehungen der einzelnen Task zueinander werden am Ende des DAG's festgelegt. In Apache Airflow werden diesbezüglich die Begriffe upstream und downstream verwendet. Im Codebeispiel 3 wäre task\_1 upstream des task\_2 bzw. task\_2 downstream von task\_1.

**Hinweis:** Airflow stellt des weiteren diverse Parameter und Makros über Jinja Templating zur Verfügung und ermöglicht über Hooks die Definition eigener Parameter, Makros und Vorlagen. [Airtg]

Listing 1: Beispielimplementierung eines simplen DAG

```

1 # DAG Definition
2 with DAG('my_dag', start_date=datetime(2021, 1, 1)) as dag:
3     # Task 1 as Bash Operator printing current date
4     task_1 = BashOperator(
5         task_id='print_date',
6         bash_command='date',
7         dag=dag,)
8
9     # Task 2 as Python Operator sleeping 5s
10    task_2 = PythonOperator(
11        task_id='sleep',
12        python_callable=lambda: time.sleep(5),
13        dag=dag,)
14
15    task_1 >> task_2 # Run Task 2 after Task 1

```

Die Initiale Ausführung eines DAG wird über Airflow WebUI oder über die Ausführung des scheduler Kommandos im CLI angestoßen. Anschließend ist der Airflow **Scheduler** für die zeitlich korrekte Ausführung und Überwachung der einzelnen Tasks verantwortlich. Im Codebeispiel 3 wird „task\_1“ als initialer Task abgearbeitet. Zur Ausführung startet der Scheduler einen Subprozess synchron zu allen DAGs die im angegeben DAG Ordner definiert sind. Anschließend werden minütlich (default) die DAG-Parsing Ergebnisse und weitere zur Ausführung stehende Tasks überprüft. Zusätzlich anstehende Aufgaben kommen dann zur Ausführung, sobald ihre im DAG definierten Abhängigkeiten zu anderen Tasks komplett erfüllt sind. [Airf] In Standardfall bedeutet dies, dass jede vorangegangene Aufgabe erfolgreich beendet werden muss (all\_success). Diese Trigger Regeln können jedoch auch auf die Fälle „all\_failed“, „one\_failed“ oder „one\_success“ angepasst werden.

Der Scheduler kann dabei Tasks sequenziell oder parallel starten. Zu diesem Zweck werden die Tasks über den Scheduler an den **Executor** übergeben, und für die Bearbeitung in eine Queue eingereiht vgl. Abbildung 2. Für jeden Schritt, den ein Task dabei durchläuft, wird ihm ein Status in Form der aktuellen „Task Stage“ zugewiesen. Die aktuellen Schritte eines jeden Tasks können zeitgleich über die Airflow WebUI grafisch nachvollzogen werden. Am Ende der Queue werden die Aufgaben durch **Worker** Nodes ausgeführt und als „Running“ markiert. Nach Abarbeitung der beispielsweise in Python beschriebenen Befehle, vergibt der Worker einen entsprechenden Status an den Task und reicht ihn an den Scheduler zurück. Dieser entscheidet anschließend, welcher Task als nächstes zur Bearbeitung in die Queue eingereiht werden soll.

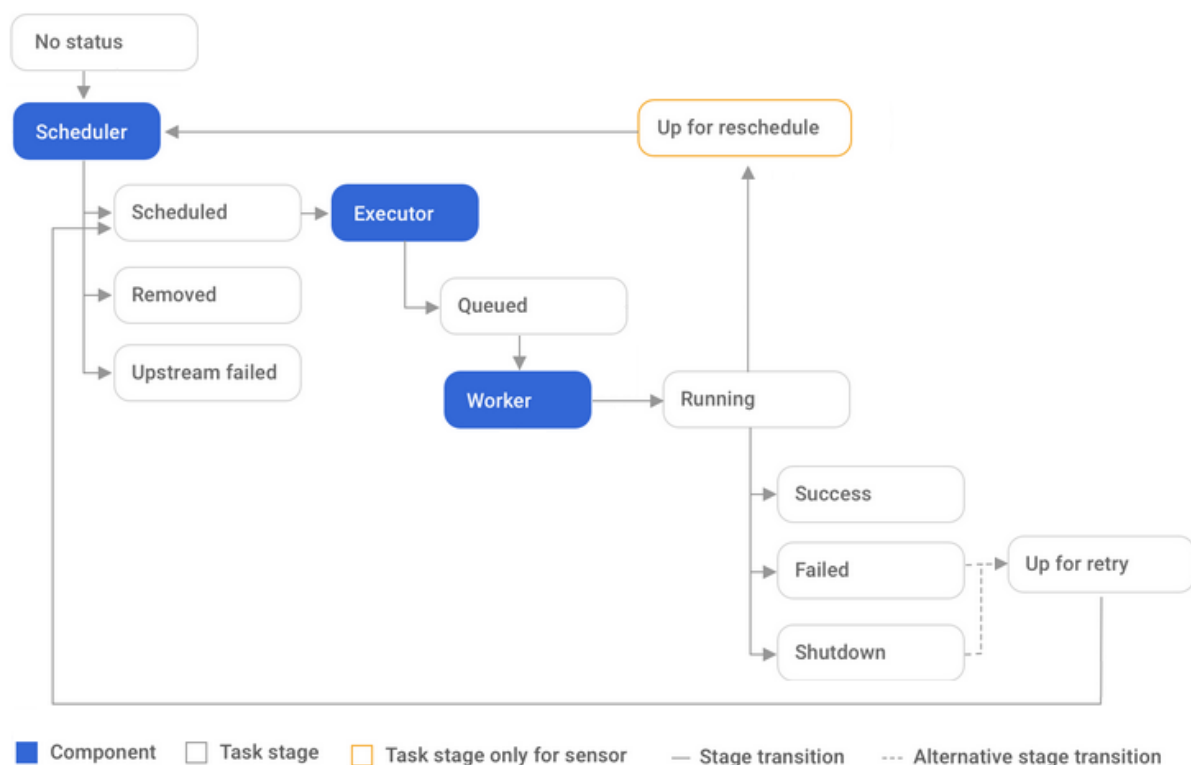


Abbildung 2: Beispiel Darstellung eines Task Lifecycle [Aire] (Bild verändert)

## 2.2 Use Cases

Dieses Kapitel beschreibt die Möglichkeiten um Apache Airflow einzusetzen und zeigt am Beispiel von zwei Unternehmen, wie es als Ökosystem genutzt werden kann.

Apache Airflow wird von einer wachsenden Zahl an einflussreichen Unternehmen und Plattformen verwendet. Darunter sind unter anderen Adobe, Airbnb, sift, Slack, Big Fish und 9gag [Airi]. Airflow bietet eine Möglichkeit, große Mengen an Aufgaben zu planen und autonom auszuführen, aber dabei dennoch Abhängigkeiten zwischen einzelnen Tasks aufrechtzuerhalten. Im Fehlerfall können diese wiederholt oder davon abhängige Aufgaben nicht ausgeführt werden. Der Blick auf die Gesamtheit der Tasks, anstatt auf einzelne, ermöglicht es außerdem, diese verteilt zu verarbeiten, etwa in einem Kubernetes-Cluster.

Apache Airflow bietet viele Vorteile. Zum Beispiel können komplexe und dynamische Workflows abgebildet werden. Es bietet eine graphische Oberfläche zum Planen und Managen der Workflows, sowie zur Überwachung. Der Status eines Workflows ist während des Ablaufs jederzeit nachvollziehbar. Durch den modularen Aufbau entsteht eine hohe Skalierbarkeit. Außerdem existieren nicht nur viele Operatoren für Airflow. Es können auch eigene Operatoren nach Bedarf entwickelt werden.

Häufig kommt Airflow zum Einsatz um Extract, Transform, Load (ETL)-Prozesse abzubilden. Dabei werden Daten aus einer oder mehreren Quellen Daten erhoben. Diese werden in ein gewünschtes Format gebracht und anschließend in einer Datenbank abgelegt. Weiterhin ist ein oft genannter Einsatzbereich das Verwalten von Machine Learning Pipelines.

Vergleicht man die Möglichkeiten mit herkömmlichen Programmen zum Planen von Aufgaben wie dem Programm cron, so lassen sich auch hier einzelne Workflows leicht abbilden und planen. Die Möglichkeiten sind jedoch begrenzt. Aufgaben, die von cron geplant werden, sind immer abhängig von der Zeit. Dabei sind die Tasks entweder an einen Kernprozess gebunden oder müssen aufgeteilt werden. Mit mehreren Einzelaufgaben können in cron jedoch keine Abhängigkeiten zwischen den Prozessen beachtet werden. Weiterhin gibt es keine Möglichkeiten zum Überwachen und entsprechende Vorkehrungen müssen in jedem ausgeführten Programm einzeln getroffen werden, was zu sehr inhomogenen Resultaten führt. Zuletzt bietet cron auch nur die Möglichkeit alle Tasks auf einem einzelnen System auszuführen.

Apache Airflow wird zum Beispiel von Sift verwendet [Airb]. Diese Plattform bietet Services an, die verdächtige Aktivitäten auf Webplattformen erkennen und Betreiber sowie Nutzer davor schützen sollen. Dafür wird Machine Learning verwendet und die Trainingsdaten werden durch eine dafür erstellte Pipeline vorbereitet. Diese wiederum besteht aus mehreren hundert Wiederholungen von MapReduce, wobei komplexe Anforderungen zwischen den einzelnen Schritten vorliegen. Viele der Arbeitsschritte sind dabei unabhängig und können parallel ausgeführt werden. Manche müssen jedoch auf Ergebnisse aus vorherigen Abläufen warten. Apache Airflow bietet eine Möglichkeit die gesamten Arbeitsschritte sowie deren Abhängigkeiten zentral zu verwalten. Die Software kommt jedoch nicht nur zum Realisieren der Trainingspipeline zum Einsatz, sondern hilft auch komplexe ETL-Pipelines abzubilden, Backups der Daten zu erstellen und die Daten in ein geeignetes Format für Experimente zu bringen. Hinzu kommt, dass Airflow ein breitgefächertes Toolset unterstützt. Es kommen hier nicht nur Python- oder Bash-Skripte zum Einsatz, sondern auch Java-Programme oder Jupyter notebooks.

Für Big Fish hingegen, stellte die Verwaltung von ETL-Anwendungen eine große Herausforderung dar [Aira]. Airflow hat gegenüber den hauseigen Softwarelösungen zum Verwalten von ETL-Workflows die Vorteile, dass es eine Web UI besitzt und das verspätete Eintreffen von Daten gut handhabt, sowie fertige und ausgereifte Programmlogik im Fehlerfall oder für erneute Versuche

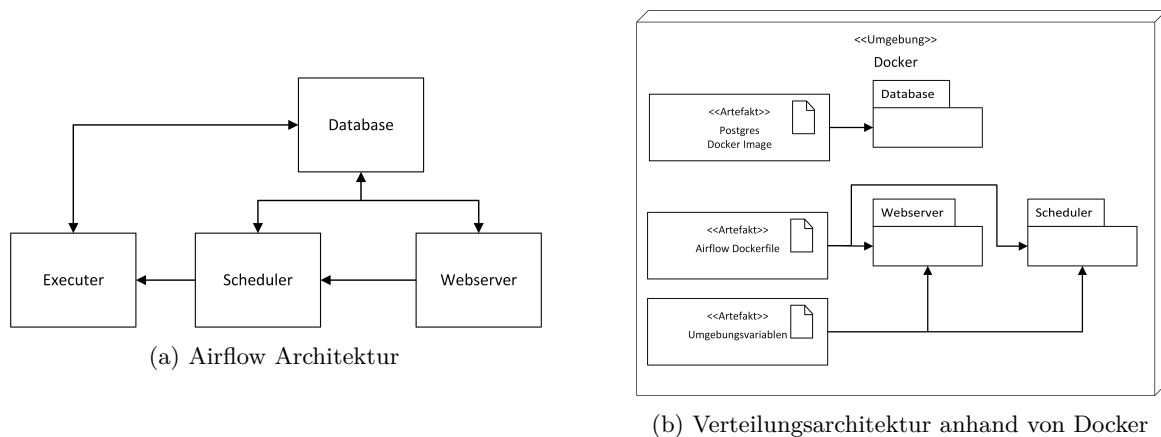


Abbildung 3: Visualisierungen der Airflow Architektur

beim Fehlerfall besitzt. Die Web UI erleichtert vor allem die Verwaltung der Workflows um ein Vielfaches. Vor allem das Vorhandensein von vielen Connectoren zum Beispiel für Hive, MySQL und Google Cloud API oder die Möglichkeit eigene Connectoren zu erstellen bieten für Big Fish einen großen Mehrwert.

## 2.3 Architektur

In diesem Kapitel wird die unterliegende Architektur von Airflow beschrieben. Diese kann in vier Komponenten aufgeteilt werden und wird in Abbildung 3a dargestellt. Der *Webserver* dient als eine Schnittstelle zum Benutzer. Der *Scheduler* kontrolliert welcher DAG als nächstes angestoßen wird und welcher Task als nächstes ausgeführt werden soll. Der *Executer* ist für die Ausführung der eigentlichen Tasks zuständig. Auf die verschiedenen Komponenten wird nun in den folgenden Unterkapiteln näher eingegangen. Jede der Komponenten kommuniziert mit der *Datenbank*, die zur Verwaltung der Metadaten von DAGs und Tasks verwendet wird [Airh]. Eine einfache Art der Installation liefert Docker. In der Abbildung 3b wird die Verteilungsarchitektur des Aufbaus gezeigt. Innerhalb einer Docker Umgebung werden drei Container gestartet, die in demselben Netzwerk miteinander kommunizieren. Die Datenbank wird anhand des offiziellen Postgres Docker Image instanziiert. Der Webserver und Scheduler werden durch ein angepasstes Docker Image gestartet. Dieses basiert auf dem offiziellen Repository von Airflow, aber enthält zusätzliche Abhängigkeiten, die zur Verbindung mit Postgres und zur Ausführung der Demoapplikation in Kapitel 3 nötig sind. In Kapitel 2.3.4 wird die Modularität und Erweiterbarkeit näher beschrieben. Zusätzlich werden über eine Datei die Umgebungsvariablen gesetzt. Diese dienen zur Konfiguration von Airflow und setzt zusätzlich den gemeinsamen Schlüssel zur Entschlüsselung der Zugangsdaten von Verbindungen.

### 2.3.1 Webserver

Der Webserver bietet Funktionen zur Überwachung und Kontrolle der Tasks an. Die Oberfläche wird durch eine *Flask* Anwendung realisiert, die auf einem *Gunicorn* Server ausgeführt wird [Airh]. In der Abbildung 4 wird der Aufbau des Webserver dargestellt. Gunicorn und Flask kommunizieren über das Web Server Gateway Interface (WSGI) [Pyt] [Airh]. Airflow verwendet für Gunicorn 4 synchrone *Worker* [Airh]. Der Worker arbeitet Anfragen synchron hintereinander

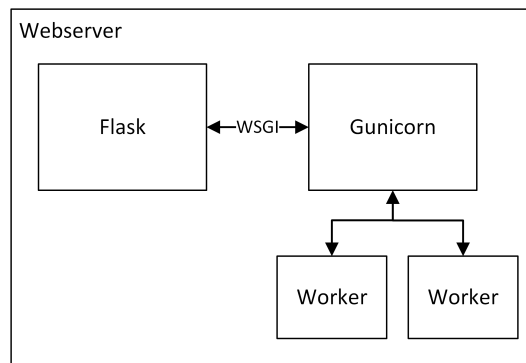


Abbildung 4: Aufbau des Webserver nach [Airh]

ab [Gun]. Falls ein Fehler in einem der Worker auftritt, wird deswegen maximal eine Anfrage beeinflusst [Gun]. Laut [Gun] sollte die Anzahl Worker nicht an die Anzahl der Benutzer gekoppelt werden. 4 bis 12 Worker sollten dabei reichen, 100 bis 1000 Anfragen pro Sekunde zu bearbeiten [Gun]. Eine gute Richtlinie zur Bestimmung der Anzahl der Worker kann durch  $(2 \times \#Prozessorkerne) + 1$  berechnet werden [Gun]. Die Formel nimmt an, dass pro Prozessor ein Worker von einem Socket liest oder schreibt und der andere die Anfrage bearbeitet [Gun]. Wie in der Abbildung 3a zu sehen ist, kommuniziert der Webserver direkt mit der Datenbank.

### 2.3.2 Scheduler

Der Scheduler überprüft die angelegten DAGs und erstellt Task Instanzen, sobald die Bedingungen erfüllt sind [Aird]. Da alle Metadaten der DAGs und Tasks über die Datenbank kontrolliert werden, kann mehr als ein Scheduler auf einmal ausgeführt werden [Aird]. Die Entwickler von Apache Airflow haben sich gegen eine direkte Kommunikation oder einen Konsensus Algorithmus (z.B.: Apache Zookeeper) zwischen den Schemulern entschieden, um den Betrieb möglichst übersichtlich zu halten [Aird]. Um diese Architekturentscheidung zu ermöglichen, muss die gewählte Datenbank zeilenbasierte Sperrungen wie *SKIP LOCKED* oder *NOWAIT* unterstützen [Aird]. Dies wird bei dem Statusübergang einer Task Instanz von *Scheduled* zu *Queued* benötigt [Aird]. Falls die gewählte Datenbank diese Funktion nicht unterstützt, blockiert der weitere Scheduler und führt eine andere Aufgabe aus [Airh]. Um dies zu bewerkstelligen wird alle 5 Sekunden durch einen *Heartbeat* folgender Ablauf ausgeführt [Airh]. Als erstes wird in dem angegebenen Ordner nach neuen oder geänderten DAGs gesucht und diese analysiert [Airh]. Darauf wird nach dem nächsten auszuführenden Task gesucht [Airh]. Dabei wird der Status zu *Scheduled* geändert und bei dem Executer in die Warteschlange eingereiht [Airh]. Als letztes wird der gewählte Executer durch einen Heartbeat benachrichtigt, dass neue Tasks zur Ausführung bereitstehen [Aird].

### 2.3.3 Executer

Der *Executer* ist für die Ausführung der Task Instanzen zuständig [Aird]. Dafür werden *Worker* erstellt, die durch den Konsolen Befehl `airflow tasks run <dag_id> <task_id> <options>` eine Task Instanz ausführen [Airh]. Diese werden durch den Scheduler ausgewählt und zur Ausführung markiert [Aird]. In Abbildung 5 wird der Aufbau von verschiedenen Executern gezeigt. Sowohl der Scheduler als auch der Executer werden auf demselben Server ausgeführt [Airh]. Dies ist unabhängig davon, ob es sich um einen Local Executer oder Kubernetes



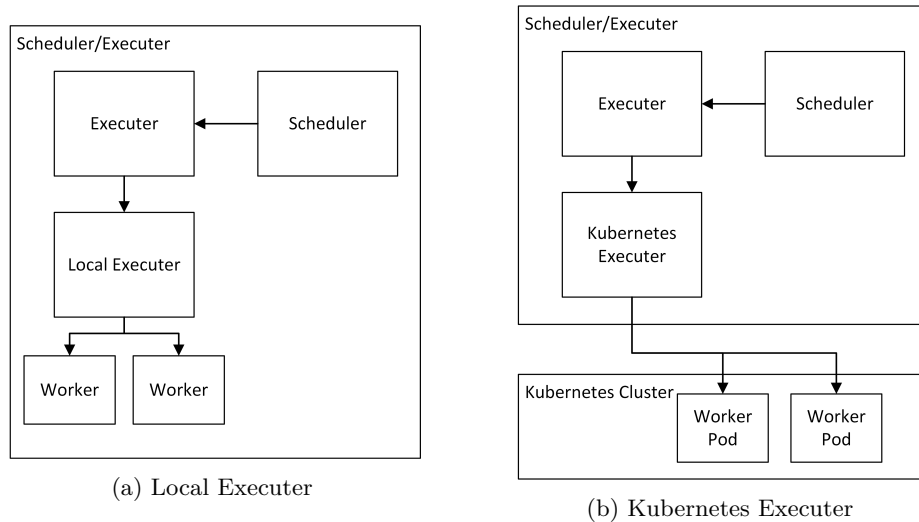


Abbildung 5: Architektur von unterschiedlichen Executern

Executor handelt.

Der *Local Executor* führt die Worker lokal auf demselben Server aus [Airh] und wird in Abbildung 5a dargestellt. Die Demoapplikation greift aus diesem Grund auf einen *Local Executor* zurück. Standardmäßig wird der Worker als Fork erstellt, aber kann auch als neuer Subprozess ausgeführt werden [Aird]. Ein Fork ist jedoch schneller als die Erstellung eines Subprozesses [Aird]. Um den Server nicht durch eine hohe Anzahl an Workern zu überlasten, kann die Anzahl der parallel ausgeführten Task Instanzen begrenzt werden. Dies wird in Airflow als *Unlimited Parallelism* oder *Limited Parallelism* bezeichnet.

Bei der *Limited Parallelism* wird eine Queue verwendet, um Tasks zur Ausführung einzureihen [Aird]. Sobald ein neuer Task in die Queue hinzugefügt wird oder die maximale Anzahl an Task Instanzen nicht überschritten ist, wird ein neuer Task ausgeführt [Airh]. Falls eine Beschränkung von 1 besteht, kann dies auch als *Sequential Executor* betrachtet werden, da nur ein Task nach dem anderen bearbeitet wird [Aird].

Durch *Unlimited Parallelism* wird jeder Task sofort an einen neuen Worker übergeben [Airh].

Die Abbildung 5b zeigt den Kubernetes Executor. Der Executor selbst läuft auf demselben Server wie der Scheduler, jedoch werden die Worker in einem Kubernetes Cluster ausgeführt [Aird]. Ein *Cluster* besteht aus *Nodes*, welche als Arbeitsmaschine beschrieben werden kann [Kubb]. Auf einem Node werden *Pods* ausgeführt [Kubb], die in diesem Fall einem Container entsprechen [Airh]. Somit ist der Executor nicht mehr an die Ressourcen des Scheduler/Executor Servers gebunden, sondern kann theoretisch unendlich viele Worker starten.

In der Abbildung 6 wird die Architektur des Celery Executors gezeigt. Celery ermöglicht durch eine *Task Queue* einen Arbeitsauftrag auf einem *Worker* auszuführen [Cel]. Dabei kann ein Worker ein Server oder eine virtuelle Maschine sein [Cel]. Die Bearbeitung von Tasks in einer Queue durch Worker entspricht dem Ansatz des Local Executors mit Limited Parallelism, mit der Ausnahme, dass die Worker nicht auf derselben Maschine ausgeführt werden. Diese erstellen auch einen Fork oder neuen Subprozess [Airh]. Das heißt aber auch, dass die benötigten Abhängigkeiten auf den Workern vorhanden sein müssen [Aird]. Wie in der Abbildung 6 zu

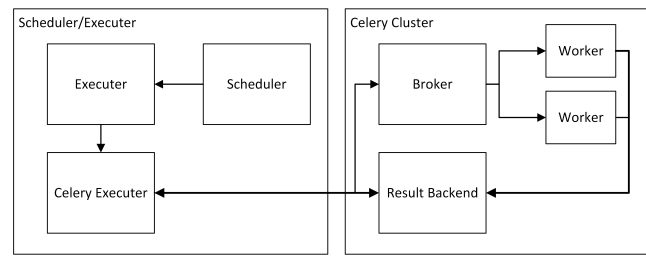


Abbildung 6: Architektur des Celery Executors

sehen ist, wird zusätzlich ein *Broker* und *Result Backend* benötigt. Der Broker implementiert die Task Queue und kann entweder durch *RabbitMQ* oder *Redis* realisiert werden [Cel]. Die Worker nehmen eigenständig neue Tasks aus der Queue und bearbeiten diese [Cel]. Das Result Backend speichert die Ergebnisse der Worker und werden durch den Celery Executor abgerufen und ändert entsprechend den Status der Task Instanz in der Airflow Datenbank [Aird].

Zusätzlich werden noch Executer für *Dask* und eine Kombination zwischen *Celery* und *Kubernetes* unterstützt [Aird], die in dieser Arbeit nicht näher betrachtet werden.

### 2.3.4 Konzepte

Eines der Ziele von Airflow ist die Erweiterbarkeit [Airh]. Durch Plugins können neue *Views* für den Webserver oder Macros erstellt werden [Aird]. Dadurch können Anwender zusätzliche Funktionen für einen Use Case ergänzen [Aird]. Eine andere Art der Erweiterung bieten Provider. Diese Funktion wurde mit der Erscheinung von Version 2.0 eingeführt und ermöglicht das Hinzufügen oder die Erweiterungen von bestehenden Operators, Hooks oder Sensoren [Aird]. Apache Airflow soll nicht mehr ein Monolith sein, in dem alle Module enthalten sind, sondern aus Bausteinen bestehen [Airc]. Dadurch wird eine effizientere Umgebung geschaffen [Airc] und Provider Pakete können öfter Änderungen veröffentlichen [Pot20]. Bei dem Start von Airflow lädt der *Provider Manager* alle *airflow.provider* Pakete [Airh]. Um *airflow.providers.\** Pakete an unterschiedlichen Orten zu installieren, wird *airflow* als *namespace package* definiert [Airh].

## 2.4 Kubernetes

Durch die Implementierung von Airflow in Python, wird für die Laufzeitumgebung von Airflow eine Python Umgebung benötigt. In der aktuellen Stable Version wird mindestens Python 3.6 vorausgesetzt. Zum Python Support wird folgendes dokumentiert:

- Sobald eine Version von den Python Entwicklern nicht weiter entwickelt wird, hört der Support bei Airflow für diese Version auf
- Default Version von Python ist die älteste angegebene Version. Zum Beispiel wird für Airflow 2.0.0 Python 3.6, 3.7 und 3.8 angegeben. Damit wird die Python Version 3.6 als Standard von Airflow vorausgesetzt.
- Eine neue Version von Python wird nach dessen Release unterstützt, sobald es in der CI Pipeline von Airflow läuft.

Damit ist Airflow grundsätzlich auf jedem System lauffähig, das eine Python Umgebung in der vorausgesetzten Version bereitstellen kann. Kubernetes ist ein Open-Source System zur

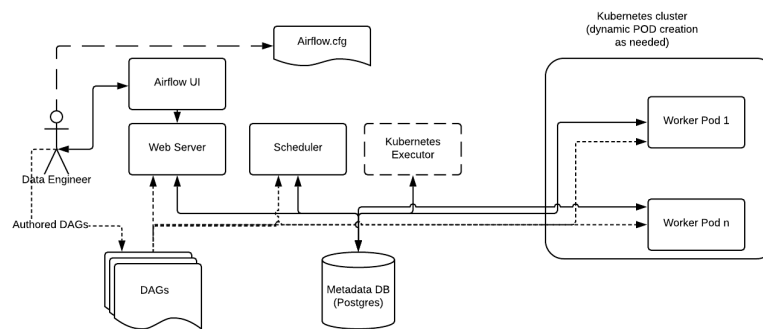


Abbildung 7: Ablauf Kubernetes Worker

Automatisierung, Bereitstellung und Skalierung von Container Anwendungen, eine sogenannte Container-Orchestrierung[Kubb]. In Kubernetes werden Nodes angelegt, diese können sowohl virtuelle, als auch physikalische Maschinen sein. Dabei verwaltet Kubernetes auf diesen Rechnern vordefinierte Container. Damit bietet es sich an, Airflow innerhalb eines Kubernetes Deployment zu betreiben. Für die Zusammenarbeit zwischen Kubernetes und Airflow gibt es zwei unterschiedliche Szenarios, die auch kombiniert werden können. Im folgendem werden beide Szenarien beschrieben.

#### 2.4.1 Airflow on Kubernetes

Die erste Möglichkeit ist es, Airflow selber in ein Kubernetes Cluster einzubinden. Hierfür wurde von Google ein spezieller Airflow Operator in Kubernetes implementiert. Der Begriff kann leicht mit den bekannten Operatoren aus der Airflow Terminologie verwechselt werden. In diesem Kontext bedeutet dieser Begriff allerdings einen Operator aus Kubernetes und beschreibt ein Stück Software, das innerhalb eines Kubernetes Clusters läuft und für die Automatisierung der Aufgaben zuständig ist. Da in Kubernetes grundsätzlich zustandslose Anwendungen [Red] verwaltet werden können, Airflow allerdings eine Datenbank benötigt, muss die Datenbank dadurch auf einem anderem Server laufen als Kubernetes.

#### 2.4.2 Kubernetes Worker

Die Integration von Kubernetes innerhalb des Airflow Frameworks bietet das zweite Szenario für die Zusammenarbeit. In diesem Szenario wird die Anforderung Skalierbarkeit der Tasks erfüllt. Hierfür wurde in Airflow ein Kubernetes Executor implementiert. Der Kubernetes Executor Operator startet jeden Task in einem vom Benutzer vordefiniertem Kubernetes Pod. Der Kubernetes Operator läuft dafür im Kontext vom Airflow Scheduler. Da Kubernetes über einen eigenen nativen Weg verfügt Tasks auszuführen, wurde in Airflow nur die Delegation der Tasks an Kubernetes implementiert. Alles, was dieser dafür benötigt, ist Zugriff auf die API des Kubernetes Cluster. Die Pods werden in einer Datei definiert und an die Kubernetes API übergeben. Dabei kann jeder Kubernetes Pod eine andere, auf die Aufgabe spezialisierte Konfiguration enthalten. Anschließend wird der Task durch Kubernetes verwaltet. Airflow selber erhält nur das Ergebnis der Berechnung zurück. Der Ablauf ist in der Abbildung 7 dargestellt.

Airflow hatte vor der Implementierung des Kubernetes Executors bereits ähnliche Konzepte, wie den Celery Executor. Diese benötigten allerdings statisch Konfigurierte und allzeit

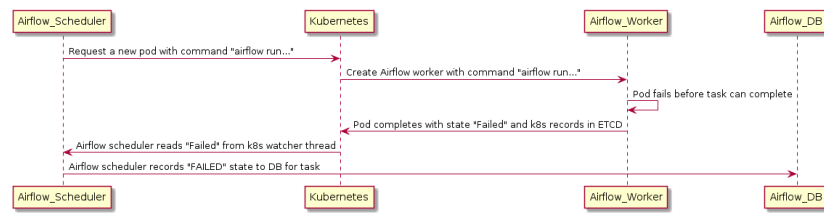


Abbildung 8: Fehlgeschlagener Task im Kubernetes Cluster

bereitete Worker. Da die Pods von Kubernetes erst bei Bedarf hochgefahren und anschließend wieder gelöscht werden, ist der positive Effekt der Benutzung von Kubernetes eine bessere Skalierbarkeit. Auf der anderen Seite aber auch unter Umständen niedrigere Latenzzeiten, da die Pods selber auch eine gewisse Startzeit haben. [Kuba] Bei diesem Konzept könnte das Problem auftreten, dass die Worker ihre Arbeit nicht vollständig abschließen können. Hierfür bekommen die Worker dann einen Zugriff auf die Datenbank und schreiben regelmäßig ihren Status in die Datenbank. Falls einer der Worker den Task nicht beenden kann, bekommt der Airflow Scheduler von der Kubernetes API ein *Failed* zurück. Anschließend wird der Airflow Scheduler den Status über die Datenbank abfragen. Dieser Ablauf ist in Abbildung 8 dargestellt.

### 3 Demoapplikation CoronaBot

Die Apache Airflow Demoapplikation umfasst einen Workflow, der tagesaktuelle Informationen zur Entwicklung der Coronafallzahlen für einen spezifischen Landkreis vom NPGeo Corona Hub 2020 über die entsprechende API liest. Diese Daten werden abgelegt, grafisch in einem Jinja-Template aufbereitet und als Email sowie über einen Telegrambot verteilt. Konkret abgefragt werden:

- Fälle der letzten 7 Tage / 100.000 Einwohner
- Genesene
- Fälle
- Todesfälle

Der sogenannte Coronabot besteht aus den drei Teilen Webserver, Scheduler und Postgres-Datenbank, welche jeweils auf einem Docker-Image basieren. Die Verwaltung der Images erfolgt über Docker-Compose. Das Postgres-Image ist dabei grundsätzlich unabhängig von Airflow und stellt lediglich eine Datenbank zur Verfügung. Der Webserver basiert auf dem Python-Server Gunicorn und bietet eine grafische Oberfläche zur Verwaltung und Steuerung der DAGs durch den Nutzer, sowie der Airflow-Instanz insgesamt.

Eingesetzt wird Airflow in der aktuellen Version 2.0.0 von Dezember 2020. Die Entwicklung der Airflow-Pipeline erfolgte in Python 3.8, da diese zum Zeitpunkt Dezember 2020 die aktuellste von Airflow unterstützte Version darstellt.

#### 3.1 Verzeichnisstruktur

Die Verzeichnisstruktur unterhalb des Homeverzeichnisses `/airflow/` besteht aus je einem dags-Verzeichnis, einem log-Verzeichnis und einem scripts-Verzeichnis, wie in Abbildung 9

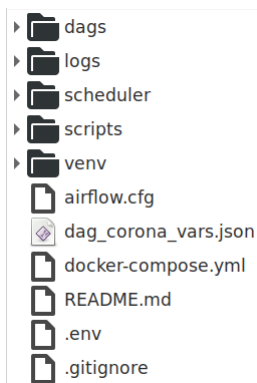


Abbildung 9: Verzeichnisstruktur CoronaBot

abgebildet. Dies entspricht der Standardverzeichnisstruktur von Airflow, sodass keine weiteren Konfigurationsschritte nötig sind. Zusätzlich gibt es ein scheduler-Verzeichnis, in welchem ein Dockerfile abgelegt ist, da der Scheduler beim starten externe Komponenten, wie den postgres-Operator und den telegram-Operator, aufgrund der Auslagerung der Operatoren aus dem Airflowkern, nachladen muss. Im Script-Verzeichnis befindet sich ein simples BASH-EntryPointScript, welches von Docker-Compose genutzt wird, um die einzelnen Komponenten zu starten und zu initialisieren. Der Ablauf ist in Listing 3.1 abgebildet. Die Kommandos werden von der Airflow-CLI bereitgestellt.

Listing 2: entrypoint.sh

---

```
1 #!/usr/bin/env bash
2 airflow db init
3 airflow webserver
```

---

Zunächst muss hierzu die Datenbank initialisiert werden. Anschließend wird der Webserver gestartet. Über die Airflow-CLI ist es auch möglich DAGs, bzw. Tasks mittels ‘airflow test <...>’ testweise ausführen, ohne auf den Scheduler zurückgreifen zu müssen. In einem env-File ist unter anderem der Connectionstring zur Postgres-Datenbank abgelegt, ein Local Executor als Core-Executor definiert, sowie eine Reihe weiterer Konfigurationen. Diese Einstellungen könnten grundsätzlich, wie bei den SMTP-Daten für den Emailversand, direkt in der Airflowkonfigurationsdatei ‘airflow.cfg’ eingetragen werden. Zu Demozwecken wurden allerdings beide Möglichkeiten verwendet. Dabei wird .env höher priorisiert als airflow.cfg. Der Local Executor führt die Tasks auf dem selben System aus, auf dem Airflow läuft. Als Treiber für die Postgresverbindung wird, entsprechend der Airflowdokumentation, ‘psycopg2’ verwendet. Für Demozwecke ist außerdem die Konfigurationsdatei in der Weboberfläche sichtbar geschaltet. Für den Telegram-Bot gibt es, zur Verschlüsselung, einen Fernent-Key sowie ebenfalls einen Connectionstring. Die Datei ist in Listing 3.1 abgebildet.

Listing 3: .env

---

```
1 AIRFLOW__CORE__SQL_ALCHEMY_CONN=postgres://airflow:airflow@postgres/
  airflow
2 AIRFLOW__CORE__EXECUTOR=LocalExecutor
3 AIRFLOW__CORE__FERNET_KEY=q_1P6Sx4YmZB_NACOXCI1B8iH8fJvy8qGgqHzN8H118=
4 AIRFLOW__WEBSERVER__EXPOSE_CONFIG=True
5
6 AIRFLOW_CONN_TELEGRAM_DEFAULT=http://1573377014%3AAAHX_eM4mYzc8pkqr1qagpuhfT-
  kkGvFkrg@
```

---

## 3.2 DAG-Entwicklung

Der Scheduler lädt automatisch alle DAGs, welche sich im entsprechenden Verzeichnis befinden. Zur Entwicklung eines neuen DAGs wird eine Pythondatei `'corona_dag.py'` angelegt. Die einzelnen Schritte der Demoapplikation, die sogenannten Tasks, werden hierzu in einem DAG-Objekt gesammelt, welches die Beziehungen und Abhängigkeiten der Tasks voneinander strukturiert. Diese Struktur wird in der Weboberfläche aufgegriffen, in der jeder Task als Node dargestellt wird. Der Scheduler verwendet dabei an einem DAG angehängte Argumente, wie die Häufigkeit der Ausführung, Startzeitpunkt, Email-Adresse für Fehlerbenachrichtigungen, etc. Tasks selbst sind sogenannte 'Unit of works' und die Implementierung eines Operators. Für die Demoapplikation wurden folgende Operatoren implementiert:

- PostgresOperator zur Datenbankinteraktion
- PythonOperator zur Abfrage der API, als Wrapper für den Mailversand, sowie zum Versenden der Telegramnachricht
- EmailOperator zum Versenden der Email

Die Implementierung von Hooks und Operatoren läuft dabei über Callables ab. Zunächst wird über die PostgresOperatoren `'create_table_api_data'` und `'create_table_html_template'` je eine Tabelle für die Rückgabewerte des API-Aufrufs, sowie für das gefüllte Template der Email in der Datenbank erzeugt. Anschließend fragt der PythonOperator `'fetch_api_data_into_db'` die Coronaapi ab und schreibt die Daten in die zugehörige Tabelle. Die URL, sowie die Parameter der Abfrage sind in Variablen gespeichert. Diese können entweder manuell in der Weboberfläche erstellt und editiert werden, oder im JSON-Format importiert oder exportiert werden. So lässt sich beispielsweise der abzufragende Landkreis dynamisch ändern. Variablen können auch genutzt werden um Informationen zwischen Tasks, unter Umgehung der langsamen Datenbank, auszutauschen. Der PythonOperator `'render_html_message'` ließt die Daten wieder aus, schreibt sie in das HTML-Template und legt dieses in der entsprechenden Tabelle ab. Die Kommunikation zwischen diesen beiden Tasks erfolgt somit über die Datenbank. Airflow bietet mit der 'cross-communication', XCom, einen key-value-Speicher für die schnelle Kommunikation zwischen Tasks an. Dieser Mechanismus wird für die Weitergabe des HTML-Templates an den `'send_email'` Task, sowie die Weitergabe der Fallzahlen für die Telegramnachricht verwendet. Hierzu schreibt der `'render_html_message'` Task die Informationen in die XCom-Datenbank. Der Email-Task pulled das HTML-Template aus der Datenbank, ließt die Email-Empfänger aus der entsprechenden Variable aus und sendet ihnen die Nachricht zu. Parallel greift der Telegram-Task auf die Fallzahlen aus der XCom-Datenbank zu, ließt die Chat-ID und seinen Token aus den entsprechenden Variablen und postet die Nachricht in den Telegramchat. Die Parallelität ist in Abbildung 14 zu erkennen.

## 3.3 DAG Management

Nach erfolgreichem hochfahren ist die Weboberfläche über `'http://127.0.0.1:8080/home'`, siehe Abbildung 10, erreichbar. Zum Login wird ein Admin-User zunächst manuell über die CLI angelegt. Die manuellen Schritte, die im folgenden zur Demonstration beschrieben sind, lassen sich auch programmatisch umsetzen. Abbildung 11 zeigt die Oberfläche zum Verwalten der DAGs, welche hier aktiviert, gestartet, gelöscht oder aktualisiert werden können. Es ist auch möglich, den aktuellen Status, sowie bereits durchgeführte Runs einzusehen. DAGs lassen sich aus der Übersicht löschen, sobald die Seite neu geladen wird, erscheint der Eintrag

Sign In


Enter your login and password below:

Username:

Password:

Sign In

Abbildung 10: Airflow Loginscreen


DAGs
Security
Browse
Admin
Docs

19:59 UTC
PP

Triggered corona\_api, it should start any moment now.

DAGs

DAG	Owner	Runs	Schedule	Last Run	Recent Tasks	Actions	Links
corona_api Demo	me	<div> <div> </div> <div> </div> <div> </div> </div>	15 08 ***	2021-01-18, 19:59:08	<div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> </div>	<div> <div> </div> <div> </div> <div> </div> </div>	
example_dag	airflow	<div> <div> </div> <div> </div> <div> </div> </div>	1 day, 0:00:00	2021-01-17, 09:58:02	<div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> </div>	<div> <div> </div> <div> </div> <div> </div> </div>	
example_telegram example	airflow	<div> <div> </div> <div> </div> <div> </div> </div>	1 day, 0:00:00	2021-01-18, 17:46:01	<div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> </div>	<div> <div> </div> <div> </div> <div> </div> </div>	
telegram_echo_bot	airflow	<div> <div> </div> <div> </div> <div> </div> </div>	1 day, 0:00:00	2021-01-17, 09:58:08	<div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> </div>	<div> <div> </div> <div> </div> <div> </div> </div>	
telegram_echo_bot_2	airflow	<div> <div> </div> <div> </div> <div> </div> </div>	1 day, 0:00:00	2021-01-18, 17:47:15	<div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> </div>	<div> <div> </div> <div> </div> <div> </div> </div>	
telegram_test example	airflow	<div> <div> </div> <div> </div> <div> </div> </div>	1 day, 0:00:00	2021-01-18, 17:37:53	<div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> <div> </div> </div>	<div> <div> </div> <div> </div> <div> </div> </div>	

Abbildung 11: Airflow DAG Verwaltung

aber wieder. Um einen DAG endgültig zu löschen, muss er aus dem dag-Verzeichnis entfernt werden. Zur Übersichtlichkeit lassen sich den DAGs Tags zuweisen. Der Demotask ist jeden Tag für 08:15 Uhr eingeplant, die Information kann in Crontab-Syntax mitgegeben werden. Bevor die Demoapplikation ausgeführt werden kann, müssen dynamische Werte, wie die Parameter der API über Variablen aus einer JSON-Datei importiert werden, siehe Abbildung 12. Diese Variablen können zur Laufzeit von Airflow oder auch manuell durch den Benutzer erstellt oder editiert werden. Diese Einträge lassen sich über die GUI in eine JSON-Datei exportieren. Airflow bietet bereits out-of-the-box eine Reihe von Datenbankverbindungen an. Für die Demoapplikation wird eine Verbindung zu einer Postgresdatenbank aufgebaut. Die Verbindungsdaten müssen hierzu in Airflow hinterlegt werden. Abbildung 13 zeigt die erstellte Verbindung mit den Informationen, welche dem Postgrescontainer im Docker-Compose.yml mitgegeben wurden. Nach dem Starten eines DAGs lassen sich in Airflow detaillierte Informationen zum Laufzeitverhalten anzeigen. Abbildung 14 zeigt als Graph-View die einzelnen Tasks, ihre Abhängigkeiten, ob sie parallel oder sequentiell ausgeführt werden, sowie den aktuellen Status ihrer Ausführung. Über die Menüleiste lassen sich bei Bedarf weitere Informationen abrufen. Falls ein Tasklauf fehlschlägt, lässt sich über einen Klick auf den Status über ein

Abbildung 12 shows the 'List Variable' page in the Airflow web interface. The page has a search bar and a table of variables. The table has columns for 'Key', 'Val', and 'Is Encrypted'. There are 9 records in the table.

Key	Val	Is Encrypted
api_base_url_string	https://services7.arcgis.com/mOBPykOjAyBO2ZKkI/arcgis/rest/services/RKI_Landkreisdaten/FeatureServer/0/	True
api_service_output_format	json	True
api_service_query_county	'SK Rosenheim'	True
api_service_query_out_fields_list	cases7_per_100k, county, recovered, cases, deaths	True
api_service_query_return_geometry	false	True
api_service_query_return_outSR	4326	True
bot_token	1594779613:AAGdZdVPpfh_1BQyTxgUUHpA6maEuQSSbll	True
chat_id	-421435563	True
email_receiver	christian.pritzl@outlook.de, corona.bot@outlook.de	True

Abbildung 12: JSON-Datei mit Airflowvariablen zum dynamischen Laden

Abbildung 13 shows the 'List Connection' page in the Airflow web interface. The page has a search bar and a table of connections. The table has columns for 'Conn Id', 'Conn Type', 'Description', 'Host', 'Port', 'Is Encrypted', and 'Is Extra Encrypted'. There is 1 record in the table.

Conn Id	Conn Type	Description	Host	Port	Is Encrypted	Is Extra Encrypted
api_postgres_connection	postgres	CoronaBot Demo	postgres		True	False

Abbildung 13: Verbindungsdaten zur Datenbank



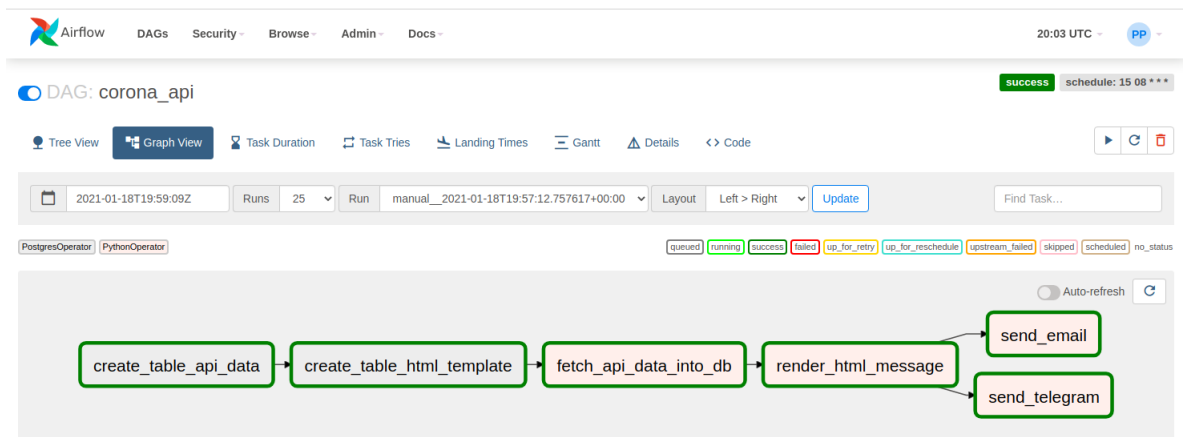
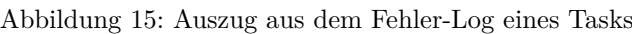


Abbildung 14: Graph-View des Corona-DAGs

Unter dem Menü das Log der aktuellen Ausführung einsehen, siehe Abbildung 15. Hier kann der Stacktrace der Taskausführung eingesehen werden. Über eine logging-Schnittstelle lassen sich bei der Programmierung des Tasks auch eigene Informationen in das Log schreiben. In diesem Untermenü ist es möglich, den gewählten Task erneut anzustoßen. Nach erfolgreicher Ausführung erhalten alle hinterlegten Email-Adressen die in Abbildung 16 gezeigte Email, sowie alle Chatmitglieder die, in Abbildung 17 gezeigte, Chatnachricht zum täglichen Corona-Update.



## 4 Fazit

Apache Airflow sticht vor allem durch die Möglichkeit, dynamische und komplexe Workflows abzubilden und zentral zu verwalten hervor. Besonders die Abbildung von Abhängigkeiten der einzelnen Aufgaben untereinander und das Einhalten dieser, machen Airflow zu einem mächtigen Werkzeug. Mit der Möglichkeit die Tasks über eine Web UI zu überwachen, zu managen und zu planen zeichnet sich die Software auch durch hohe Nutzerfreundlichkeit aus.

Durch die gewählte Architektur wird Skalierbarkeit und Redundanz der Anwendung gewährleistet. Auch die Modularität und einfache Erweiterbarkeit durch Plugins und Provider bietet dem Benutzer die Möglichkeit, den Workflow an seine Anforderungen anzupassen.

Aus Entwicklersicht fällt die Arbeit mit Airflow positiv aus. Angenehm ist die aktuelle, gut formulierte Dokumentation mit verständlicher, übersichtlicher Gliederung. Dank funktionierender Beispiele, basierend auf aktuellen Technologien wie Docker, fällt der erste Einstieg in die Entwicklung leicht. Auch die Community um das Produkt beteiligt sich rege in Forumsdiskussion und auf einschlägigen Plattformen. Die Entscheidung für Python als Programmiersprache kann aufgrund der geringen Einstiegshürde und der vielfältigen Bibliotheken, die diese Sprache mit sich bringt, ebenfalls als positiv eingestuft werden. Sobald die grundsätzlichen Konzepte verstanden wurden, können problemlos auch komplexere Anwendungsfälle, gegliedert in Einzelschritte, umgesetzt werden.

## Literatur

- [Aira] *Airflow - Use Cases - Big Fish Games*. <https://airflow.apache.org/use-cases/big-fish-games/>. last visit: 10 Jan 2021. 2021.
- [Airb] *Airflow - Use Cases - Sift*. <https://airflow.apache.org/use-cases/sift/>. last visit: 10 Jan 2021. 2021.
- [Airc] *Apache Airflow 2.0 is here!* <https://airflow.apache.org/blog/airflow-two-point-oh-is-here>. last visit: 4 Jan 2021. 2020.
- [Aird] *Apache Airflow Documentation*. <https://airflow.apache.org/docs/apache-airflow/stable>. last visit: 4 Jan 2021. 2021.
- [Aire] *Apache Ariflow Dokumentation Konzepte*. <https://airflow.apache.org/docs/apache-airflow/stable/concepts.html>. last visit: 10 Jan 2021. 2021.
- [Airf] *Apache Ariflow Dokumentation Scheduler*. <https://airflow.apache.org/docs/apache-airflow/stable/scheduler.html>. last visit: 10 Jan 2021. 2021.
- [Airg] *Apache Ariflow Tutorial*. <https://airflow.apache.org/docs/apache-airflow/stable/tutorial.html>. last visit: 10 Jan 2021. 2021.
- [Airh] *Github - Apache Airflow*. <https://github.com/apache/airflow>. last visit: 4 Jan 2021. 2021.
- [Airi] *Stackshare - Airflow*. <https://stackshare.io/airflow>. last visit: 10 Jan 2021. 2021.
- [Cel] *Celery - Distributed Task Queue*. <https://docs.celeryproject.org/en/stable>. last visit: 4 Jan 2021. 2020.
- [Goo] *Google Cloud Dokumentation*. <https://cloud.google.com/composer/docs/how-to/using/writing-dags?hl=de>. last visit: 12 Jan 2021. 2021.
- [Gun] *Gunicorn - WSGI server Documentation*. <https://docs.gunicorn.org/en/stable/index.html>. last visit: 4 Jan 2021. 2019.
- [Kuba] *Celery - Distributed Task Queue*. <https://docs.celeryproject.org/en/stable>. last visit: 21 Jan 2021. 2015.
- [Kubb] *Kubernetes Dokumentation*. <https://kubernetes.io/de/docs/home>. last visit: 4 Jan 2021. 2020.
- [Pot20] Jarek Potiuk. *Airflow 2.0 Providers*. <https://www.polidea.com/blog/airflow-2-providers>. last visit: 4 Jan 2021. 2020.
- [Pyt] *PEP 3333 - Python Web Server Gateway Interface*. <https://www.python.org/dev/peps/pep-3333>. last visit: 4 Jan 2021. 2010.
- [Red] *Was ist ein Kubernetes Operator?* <https://www.redhat.com/de/topics/containers/what-is-a-kubernetes-operator>. last visit: 21 Jan 2021. 2021.