Ref. Ares(2022)4797210 - 30/06/2022

HORIZON 2020

The EU Framework Programme for Research and Innovation

CURE System

Deliverable D4.3

DATE
30 June 2022

ISSUE
1.0

GRANT AGREEMENT
no 870337

DISSEMINATION LEVEL
PU

PROJECT WEB-SITE
http://cure-copernicus.eu/

AUTHORS
Johannes Schmid (GeoVille)
Samuel Carraro (GeoVille)
Mario Dohr (GeoVille)

CONTENTS

1 Introduction ... 3
1.1 Purpose of the document ... 3
2 System Components ... 4
2.1 API Gateway ... 4
2.2 Authentication and Authorization ... 5
2.3 Order Status Updates ... 5
2.4 Message Broker ... 6
2.4.1 Queueing System ... 7
2.4.2 Queue listener ... 7
2.5 Scheduler ... 7
2.5.1 DAG ... 7
2.5.2 Operator ... 8
2.5.3 Worker ... 8
2.5.4 Parallelism ... 8
2.5.5 Monitoring ... 8
2.6 Logging ... 8
2.7 Database ... 9
2.8 Container virtualization ... 10
2.9 Monitoring ... 11
3 Modules ... 12
3.1 Logging module ... 12
3.2 Database module ... 12
3.3 RabbitMQ module ... 12
3.4 Request validation module ... 12
4 Testing ... 14
4.1 Component Tests ... 14
4.2 Integration Tests ... 15
4.3 System Tests ... 16
5 Conclusion ... 16

Figure 2-1 General Workflow 4
Figure 2-2: Overview of order status updating workflow 6
Figure 2-3 Graph Example of DAG 8
Figure 2-4 Data Model 10

List of Acronyms

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
DAG	Directed Acyclic Graph
DIAS	Data and Information Access Services
GIS	Geographic Information Systems
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
OGC	Open Geospatial Consortium
OS	Operating System
REST	Representational State Transfer
REGEX	REGular EXpression
SQL	Structured Query Language
UI	User Interface
WP	Work Package

1 INTRODUCTION

1.1 Purpose of the document

Deliverable 4.3 is an essential delivery of Task 4.4 Integration of Services in WP4 Cure System Development.

CURE System will describe the implementation of architectural design as described in D4.2 in detail based on the usage of DIAS as a Service. That comprises the set-up of all System Components as well as the software design including the interaction between those components.

2 System Components

The system is based on a distributed microservice architecture. Figure 2-1 General Workflow shows the general workflow with the main system components (microservices) and how they link together. Besides the description of the workflow on the right-hand side, the Figure also displays the modules and software stack used by each component.

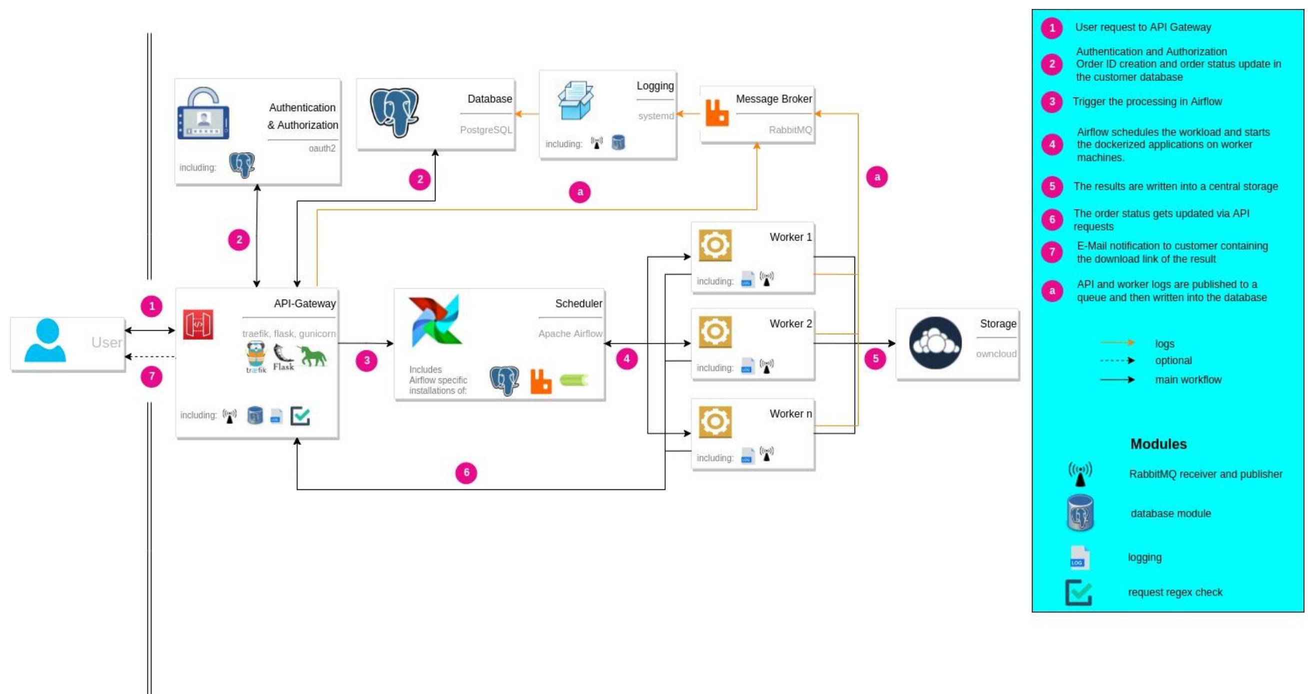

Figure 2-1 General Workflow

2.1 API Gateway

An API gateway sits between a client and a collection of backend services and serves as the entry point to the microservice infrastructure. An API gateway accepts all application programming interface (API) calls, aggregates the various services required to fulfil them, and returns the appropriate result. The RESTful API provides all endpoints which are required to interact with the system. The endpoints are divided into namespaces or sections to group common operations (e.g.: geo-services, custom-relation-management services, etc.). The API is documented with the help of the OpenAPI specification. The OpenAPI Specification (OAS) defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. An OpenAPI document that conforms to the OpenAPI Specification is

itself a JSON object, which may be represented either in JSON or YAML format. An OpenAPI definition can then be used by documentation generation tools to display the API, code generation tools to generate servers and clients in various programming languages, testing tools, and many other use cases. Moreover, the UI generated by these tools supports a straightforward workflow to run, debug and test all API endpoints. Please visit https://services.geoville.com/cure/v1/ for more details.

2.2 Authentication and Authorization

User authentication, authorization and management are based on the OAuth2 framework. It is used to exchange data between client and server through authorization. The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf. The Authorization Server provides several endpoints for authorization, issuing tokens, refreshing tokens and revoking tokens. When the resource owner (user)authenticates, this server issues an access token to the client. The resource owner is the user who is using a service. A resource owner can log in to a website with a username/email and password, or by other methods. A client is an application making protected resource requests on behalf of the resource owner and with its authorization. Any application that uses OAuth 2.0 to access the CURE API must have authorization credentials that identify the application to the OAuth 2.0 server. Therefore, the authorization server comes with a PostgreSQL database for managing users, clients, access permissions and access tokens. The API of the authorization server provides a set of endpoints which are required to perform common authorization operations and flows. Amongst others, this includes for example:

- Creating OAuth clients
- User login
- Access token generation
- Token validation
- Scopes creation and management

Scopes define which services a user has access to.

2.3 Order Status Updates

The status of an asynchronous order changes in the course of processing it. Possible order states can be the following:

- RECEIVED: the order has succeeded the validation and was accepted
- QUEUED: the order has been sent to the service queue
- RUNNING: the order started processing
- INVALID: there is no satellite data available for the requested date(s)
- SUCCESS: the order finished successfully
- FAILED: the order failed during processing
- ABORTED: the order was canceled manually by the user

The states are being updated by the API (RECEIVED, QUEUED and ABORTED) and by Airflow (RUNNING, INVALID, FAILED, SUCCESS). The overall workflow is illustraded in Figure 2-2.

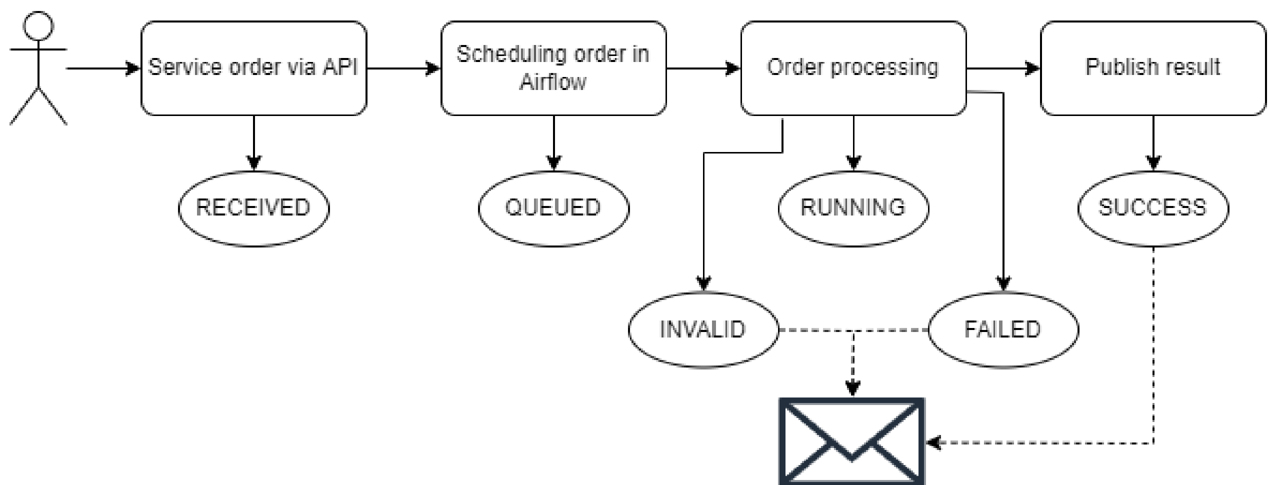

Figure 2-2: Overview of order status updating workflow

Updating a status is done by the API gateway, which offers respective routes for Airflow to access. For specific status updates such as “SUCCESS”, “INVALID” and “FAILED” an e-mail notification is sent to the user, if the optional notification parameter in the service ordering payload is set to “true”. By default no email is sent.

2.4 Message Broker

As already mentioned, the system infrastructure is based on a microservice architecture. The communication between these microservices is implemented by a message broker. Specifically, the system uses the open-source message broker RabbitMQ together with a Python module that listens to the queues and triggers the respective services in the scheduler.

2.4.1 Queueing System

RabbitMQ is an open-source message broker that implements the Advanced Message Queuing Protocol (AMQP). Basically, the message broker software has two functions, to "publish" and to "receive" messages. By using the Python module pika, an AMQP connection to RabbitMQ can be created to send requests to and receive requests from the server. To ensure that the messages are secure - as they might include sensitive information - Cryptography is used to encrypt the messages.

The main advantages of using RabbitMQ are the high scalability, the ability to run on most operating systems and cloud environments and the regular updates from a large developer community.

2.4.2 Queue listener

Service and System logs that were published to a RabbitMQ queue need to be received and written to the respective logging database table. This is done by a system service. Both functions, listening to the queue and storing the logs inside a database happen on different threads.

For efficient usage of RabbitMQ's functionalities from within our system, we created a Python module that simplifies accessing queues from our code. Details about the module can be found in section 3.3.

Besides the usage of RabbitMQ for handling logs, a separate full installation of RabbitMQ gets used by Apache Airflow to schedule the different services.

2.5 Scheduler

The scheduling system is one of the most important components of the system infrastructure. It allows the creation, execution and monitoring of multiple parallel workflows and tasks. The scheduler is based on the Apache Airflow workflow management platform. Airflow is an open-source platform based on Python that is designed under the principle of "configuration as code".

The following sub-sections address some of the most important Apache Airflow components.

2.5.1 DAG

Apache Airflow supports the creation of workflows. A workflow is formulated as a Directed Acyclic Graph (DAG). Usually, such a DAG is a collection of tasks, and each DAG is represented by a Python script.

In Figure 2-3 Graph Example of DAG an example DAG is visually presented. Each rectangle represents a task which can be either a Python function, a Bash command or a Docker container.

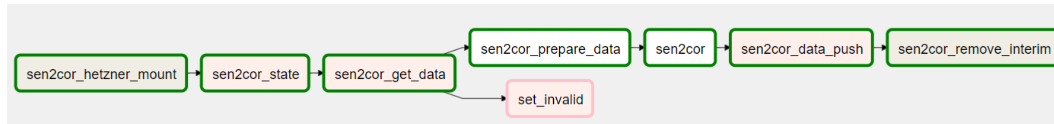

Figure 2-3 Graph Example of DAG

2.5.2 Operator

A DAG consists of several tasks which are also called operators. Airflow supports various types of operators. Common operators which are used in the current installation are listed below:

- Python Operator: Executes Python callable and commands.
- Bash operator: Executes commands in a Bash shell.
- Docker operator: Executes a command inside a docker container.
-

2.5.3 Worker

One advantage of Apache Airflow is the support of distributed system architectures. The system includes several so-called workers, which run on systems known as worker nodes. Jobs can be assigned by the main Airflow application to worker instances by using message protocols. Put simply, each production step can run on a separate virtual machine and is managed by the airflow scheduler machine.

2.5.4 Parallelism

Apache Airflow provides powerful tools and configuration options to run workflows (DAGs) and even single tasks (operators) in parallel. This helps to manage huge workloads.

2.5.5 Monitoring

Airflow provides a powerful monitoring tool which helps to monitor, start, delete and debug operators and DAGs.

2.6 Logging

The logging functionality of the system aims to provide a simple, yet flexible way to log events in a central database from each system component. It is based on:

1. a mechanism that receives logging messages and sends them into a queue
2. a service that consumes the log messages and stores them.

The receiving mechanism offers two options to be used for logging:

- A Python module which provides a command for logging. For details see section 3.1.
- A REST endpoint which allows logging via HTTP request, which provides a lot of flexibility because any client capable of sending HTTP requests can use it.

Generally, both logging mechanisms send messages into a queue.

The consumption mechanism is a standalone program (GeoVille_MS_Logging_Saver) which reads the log messages in batches from the queue and persists the logs in a relational database.

In order to support traceability and debugging, the logging module provides different log-levels:

- INFO: Confirmation that things are working as expected.
- WARNING: Indicates that something unexpected happened, but the software is still working as expected
- ERROR: Indicates a serious problem. The software has not been able to perform some function.

2.7 Database

Any modern backend solution needs a storage system to store data whilst processing particular tasks. As spatial information will be processed in the project, the tool chosen was a PostgreSQL database server with its extension PostGIS for spatial operations. PostgreSQL is a powerful, Open-Source object-relational database system with over 30 years of active development. PostgreSQL supports both SQL (relational) and JSON (non-relational) querying. PostgreSQL is a highly stable database and used as a primary database for many web applications as well as mobile and analytics applications. PostGIS is a spatial database extender for PostgreSQL object-relational database. It adds support for geographic objects allowing location queries to be run in SQL and provides geospatial databases for geographic information systems (GIS). PostGIS follows the Simple Features for SQL specification from the Open Geospatial Consortium (OGC). A relational database model is required to persist the processed information in a structured manner. The Figure 2-4 Data Model below shows the data model used by the CURE API.

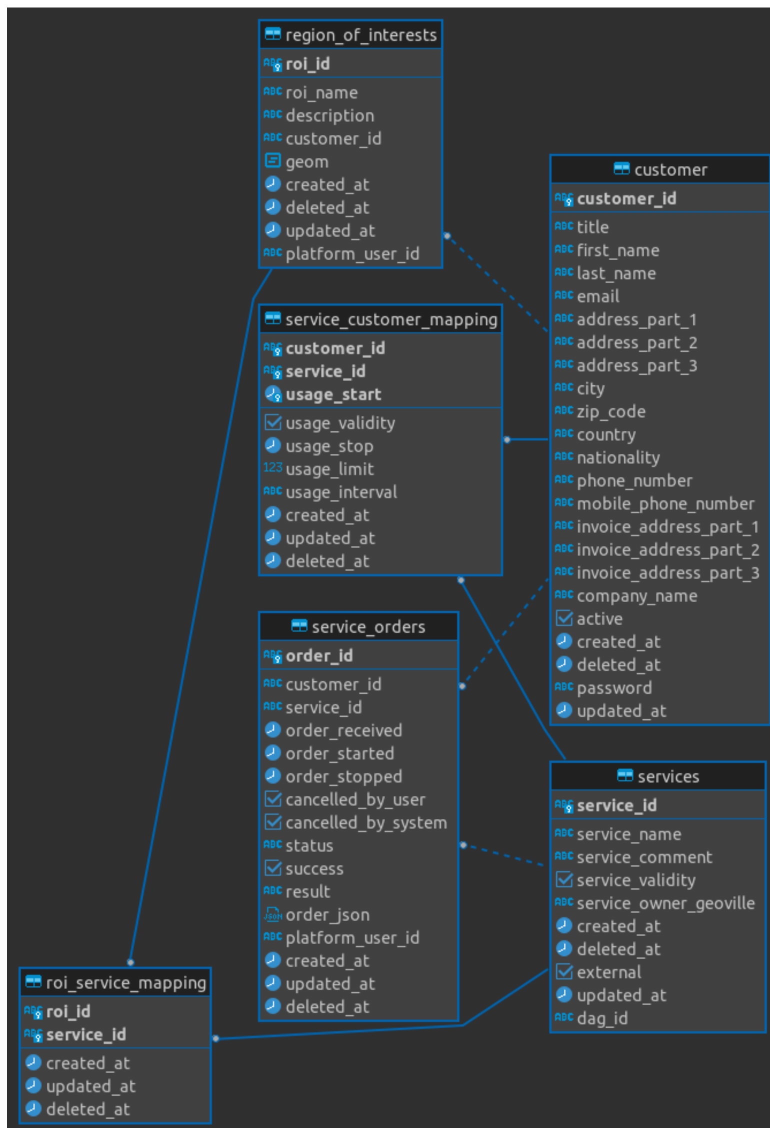

Figure 2-4 Data Model

2.8 Container virtualization

For OS-level virtualization the system uses Docker. As already mentioned, Apache Airflow provides an operator for efficiently running Docker containers. However, not just the single service tasks have been dockerized, but also the Airflow installation and configuration of each worker node.

Besides the Airflow components, the API gateway, the authentication and authorization as well as the central PostgreSQL database have also been dockerized.

The main reason why these system components have been dockerized is to enable simple reusability and scaling. When a system component needs to be migrated to another virtual machine or when another Airflow worker needs to be deployed, the Docker image can be

reused instead of spending hours on installation and configuration. This can be done by either pulling the Docker image from a private or public Docker registry or by building the Docker image locally.

Another big advantage of Docker is the independence from system updates. While the upgrade of a program or module by a system update could cause dependency problems if the system component runs as a daemonized system service, the installation and execution within a Docker container is not affected.

2.9 Monitoring

The monitoring system consists of three parts. Prometheus is a monitoring solution for storing time series data like system metrics. Grafana allows for the visualisation of the data stored in Prometheus. The Prometheus Alert manager is the third part of the monitoring system. It handles any alerts sent by the Prometheus server.

3 MODULES

3.1 Logging module

The logging module is a Python module which simplifies logging from within Python code. It takes the log messages, including the log level, and sends them to the logging queue (RabbitMQ). This module helps to simplify the code, as it does not require communication with the logging API via HTTP. To send messages to the queue, the RabbitMQ module described in section 3.3 is used.

3.2 Database module

This module abstracts a PostgreSQL database connector and provides several functions to read from and write into tables. The provided methods are:

- read one row from a query
- read all rows from a query
- read many (a specific number of) rows from a query
- execute commands such as insert, update, create, drop, delete, etc.

3.3 RabbitMQ module

This Python module provides the basic implementation to retrieve messages from and publish messages to RabbitMQ queues. It consists of the classes BaseReceiver and Publisher. The Publisher makes it very simple to send a message to a queue, as can be identified by its name. A message can be everything from a string to a number to a more complex object like a dictionary. The BaseReceiver on the other hand can listen to a queue and retrieve messages whenever there are some in the queue.

3.4 Request validation module

Before a service request can be queued and sent to the scheduler, it must be validated. If the validation fails, an error is instantly returned to the user. Hence, the user will know right away that invalid input parameters were provided or that parameters are missing. If no validation was done, the system would attempt to process the service request normally and it could take minutes or even hours before the code realizes that the input was incorrect.

The validation consists of several parts such as checking whether the user or a processing unit exists. If dates are included, they get checked to ensure that they lie within an acceptable range (from the satellite start until the current date) and that the end date is later than the start date. Beside those basic content checks, a REGEX check is also performed. REGEX stands for REGular EXpression and can be described as a search pattern. This pattern can be used by a search algorithm to search a text for numbers, letters or specific characters.

It was developed in theoretical computer science as well as formal language theory and is often used for input validation.

By executing the following command from the request validation module, a request payload can be validated:

```
check_message({"servicename": "test_service", "unit": "123",
"begin": "2019-06-30", "end": "2019-12-31"})
```

In this example, a request with four payload parameters (servicename, unit, begin and end) gets checked. The result is a boolean that states whether the request is valid (True) or invalid (False).

For example, the payload parameter “begin” should include a date. However, it has to be in the specific format “Year-Month-Day" (e.g. 2019-06-30). The REGEX for this would be

```
([12]\d{3}-(0[1-9]|1[0-2])-(0[1-9]|[12]\d|3[01]))
```

If the user does not provide the parameter “begin” or if the user provides an incorrectly formatted date such as “30.06.2019”, the validation fails and the request will not be processed. The user will receive an appropriate explanation as to why the request has failed, either because the parameter was missing or because it has the wrong format.

Note that this module expects that the service and its REGEX check rules are inserted in the "message_checker" table of the "postgres" database.

After extracting the list of parameters and regular expressions for the requested service from the database by using the module described in chapter 4.2, the parameters of the request and the database extraction will be compared. If a required parameter is missing, an error will be returned. In case of a success, the parameter values (e.g. the date of the parameter “begin”) will be validated using the regular expression. This part of the code mainly uses the Python module re[1].

[1] https://docs.python.org/3/library/re.html

4 Testing

Since the processing of CURE is needed to be on demand, integration tests are run before every new change or update to the process chain. This ensures that the processing of CURE delivers correct results after every update. Furthermore, to ensure that the system is completely up and running system tests and its component tests are performed every time the system is updated. To ensure it is running smoothly while processing the system is monitored and in case of an unexpected event the IT-Team of GeoVille is alerted by the monitoring system.

4.1 Component Tests

The 4 main Component of the processing system are the API, and three components in Airflow (Scheduler, Webserver and Worker). Each one of them needs to be tested separately, before a full system test can be performed.

Test case IDs	Test 1
Purpose	Test if API is up and running
Test items	API
Startup condition	Before System tests
Expected result	API is up and running and can retrieve user inputs

Test case IDs	Test 2
Purpose	Test if Airflow Scheduler schedules jobs
Test items	Airflow scheduler
Startup condition	Before System tests
Expected result	Airflow Scheduler schedules jobs and orchestrates them to the workers

Test case IDs	Test 3
Purpose	Test if Airflow Webserver schedules jobs
Test items	Airflow Webserver
Startup condition	Before System tests
Expected result	Airflow Webserver shows all DAGs and its states and executions

Test case IDs	Test 4
Purpose	Test if Airflow Worker receives Jobs and executes them
Test items	Airflow Worker
Startup condition	Before System tests
Expected result	Airflow Worker successfully executed Job and reports back the result to the Airflow Scheduler

4.2 Integration Tests

After every update to the items in or the process chain of a CURE App itself in Airflow an integration test is performed to ensure the functionality. A given set of parameters of a test scenario is used as input. After the execution of the process chain the expected output is compared to the output generated by the process chain. If no errors or differences in the outputs are detected the integration test was successful.

Test Cases

Test case IDs	Test 1
Purpose	Test of the route of CURE app
Test items	Route of CURE app in API
Startup condition	After new Integration or Update
Expected result	Execution of route triggers DAG

Test case IDs	Test 2
Purpose	Test the Process chain of CURE app in Airflow
Test items	DAG execution in Airflow
Startup condition	After new Integration or Update
Expected result	Execution of DAG producing a valid result

4.3 System Tests

Since the system can be split into two parts API and Airflow. The API handles the request of the user and triggers the process chain with the given parameters in Airflow. Airflow schedules and orchestrates the tasks in the process chain. After a successful Execution the State is expected to be "SUCCESS" and the result should be a link to a S3 bucket to download the result.

Test case IDs	Test 1		
Purpose	Test of the System to run CURE applications		
Test items	API, Airflow		
Startup condition	After update of System		
Requirements	API up and running, Airflow Scheduler, Webserver and Worker up and running, Postgresql and RabbitMQ up and running		
Scenario steps	Action	Expected Output	Comments
1	Correct input	Trigger Dag in Airflow and return RUN_ID. In Airflow DAG gets executed and updated the state of the given RUN_ID after successful execution.	
2	Wrong input	Return correct error message to user	

5 CONCLUSION

This document presents the current version of the CURE System, its components and functionalities. With this system a stable execution of the in WP3 developed apps can be executed in a cloud environment and, together with the in Task 4.5 developed CURE web-portal, it is possible to demonstrate the capacity and potential of these applications.