Hochschule Neubrandenburg
University of Applied Sciences

Hochschule Neubrandenburg
Course of study: Geodesy and Geoinformatics

Prototype Development of an Automated Processing System for Space and Ground Based Data

Master's thesis
submitted by: *Paul David*

To obtain the academic degree
„Master of Engineering“ (M.Eng.)

First supervisor: Prof. Dr. Andreas Wehrenpfennig
Second supervisor: Dr. Mohammed Mainul Hoque

Editing period: 29.10.2019 to 27.03.2020

URN: **urn:nbn:de:gbv:519-thesis2019-0479-6**

Abstract

In the context of the establishment of the new Institute for Solar-Terrestrial Physics at DLR, this work deals with the development of an automatic, horizontally scalable system for the operational processing and postprocessing of ground- and space-based data and thus for the continuous provision of data sets that can be used in both basic and applied research. For this purpose, the use cases and the requirements on the system are first collected and analyzed. Existing systems and projects in the DLR environment are then examined in order to adapt and, if necessary, extend existing concepts. After that the system is planned in a conceptual design, which is used in the following chapters as a basis for the selection of suitable software and as a guideline for implementation.
The final implementation uses *Docker* and *Kubernetes* to create a distributed, container-based, modular system, which allows the execution of algorithms and programs implemented in almost any programming language. By using the software of the *Argo Project* these modules can be linked to complex processing chains and started in response to time- or file-based events.

Kurzfassung

Im Rahmen der Gründung des neuen Instituts für Solar-Terrestrische Physik am DLR beschäftigt sich diese Arbeit mit der Entwicklung eines automatischen, horizontal skalierbaren Systems zur operationellen Prozessierung sowie zur Nachprozessierung von boden- und weltraumgestützten Daten und somit zur kontinuierlichen Bereitstellung von Datensätzen, die sowohl in der angewandten Forschung als auch in der Grundlagenforschung zum Einsatz kommen. Dazu werden zunächst die Anwendungsfälle und die Anforderungen an das System gesammelt und analysiert. Anschliessend werden bestehende Systeme und Projekte im Umfeld des DLRs betrachtet, um bereits vorhandene und erprobte Konzepte zu adaptieren und gegebenenfalls zu erweitern. Daraufhin wird das System in einem konzeptuellen Entwurf geplant, welcher in den darauf folgenden Kapiteln als Grundlage zur Auswahl geeigneter Software und als Leitfaden zur Implementierung genutzt wird.
Die finale Implementierung nutzt *Docker* und *Kubernetes*, um so mit Hilfe eines Container-basierten, modularen Systems Algorithmen und Programme, implementiert in nahezu beliebiger Programmiersprache, auf einem verteilten Rechnernetz auszuführen. Diese Module können durch den Einsatz der Software des *Argo Project* zu komplexen Prozessierungsketten verbunden und als Reaktion auf zeitliche oder datei-basierte Events gestartet werden.

Contents

1 Introduction

The upper region of the Earth's atmosphere, which is ionized by the Sun's radiation, is called ionosphere. It is located between 60km and 1000km altitude and it's density is directly influenced by the Sun and the amount of enegery absorbed. The resulting fluctuations are attributed to space weather. Space weather refers to conditions on the Sun and in the solar wind, magnetosphere, ionosphere and thermosphere. It manifests itself in a variety of ways and has a range of effects on the Earth. The impact of the solar wind on the ionosphere affects satellites, aviation, telecommunications and navigation. For this reason, monitoring space weather is a crucial national task. (German Aerospace Center (DLR), 2019b; National Oceanic and Atmospheric Administration, 2019b; Kriegel, 2012; Noja, 2010)
The new Institute for Solar-Terrestrial Physics established in June 2019 at the DLR site in Neustrelitz will create the conditions for prompt, precise and reliable observation and forecasting of space weather. Those observations and forecasts will help to increase resilience of vulnerable infrastructure components. Therefore the new institute focusses on the state and the dynamics of the Ionosphere-Thermosphere-Magnetosphere System (ITM) and it's driving by the Sun and by the lower and middle atmosphere. This includes both basic and applied research on space weather with the goal to protect national infrastructures and support affected industries through reliable observations and forecasts. (German Aerospace Center (DLR), 2019b; German Aerospace Center (DLR), 2019a)
The main objective of this work is to develop an operational processing system using state of the art technologies according to the software development guidelines of DLR. This system will be used to run various algorithms and reliably produce the data needed for both basic and applied research on this subject.

1.1 Overview of space and ground based data

To achieve this goal a lot of data can be taken into consideration including space and ground based observations.

1.1.1 Space Based Observations

The availability of numerous medium Earth orbit (MEO) satellites deployed by GPS, GLONASS, BeiDou systems and fulfillment of Galileo constellation, allows continuous monitoring of the ionosphere and neutral atmosphere of the Earth by tracking GNSS signals from low Earth oribiting (LEO) satellites.

In addition to the GNSS radio occultation measurements on board, navigation measurements can be used to derive the total electron content between GNSS and LEO satellites. This information provides an excellent database for subsequent data inversion and assimilation into 3D models of the ionospheric electron density distribution up to GNSS orbit heights. (Heise et al., 2002; Jakowski et al., 2002)
Dual-frequency altimeter missions such as TOPEX-Poseidon and Jason 1 and 2 are excellent sources of non-GNSS based TEC[1] data over the oceans. The data are used for the validation of global ionospheric maps of vertical TEC computed from ground based monitoring stations. As the altimeter measurements are limited to the ocean areas, these data are geographically complementary to GNSS data, which are mainly obtained over land. In the same way, the geodetic orbit determination and positioning system DORIS, which consists of a global network of terrestrial beacons, shall be considered as an independent data source for ionospheric monitoring and reconstruction. GNSS reflectometry is a new remote sensing technology that opens new scientific horizons in topography, weather forecast and climate research. Dual frequency signals allow estimating the TEC, which will support space weather monitoring in future.
In addition to indirect TEC measurements along signal paths, direct in situ measurements, especially of electron density, e.g. Langmuir probe on board CHAMP and SWARM, and energetic particles from space platforms, e.g. GOES, Galileo, shall be addressed to complement the data sets obtained from other sources. (Heise et al., 2002; Jakowski et al., 2002; Park et al., 2013; Pignalberi et al., 2016)

1.1.2 Ground Based Observations

With the modernization and completion of individual GNSS, the use of multi-constellation multi-frequency observations including new signals allows continuous monitoring of Earth's ionosphere using globally distributed sensor stations. In addition, the steadily growing number of GNSS receivers and associated networks essentially supports the establishment of high precision monitoring of ionospheric weather, including perturbation tracking and forecasts, which can be used in space weather services. Other ground based methods like Vertical Sounding (VS), Incoherent Scatter Radar (ISR), Very Low Frequency (VLF) or Radio Beacon (RB) measurements provide complementary data and can therefore be used to complete GNSS based data sets. VLF signals (3 - 30 kHz), which are usually transmitted for communica-

[1]Total electron content

tion with submarines, can effectively be used to detect solar flares in the X-band range, for example by the GOES satellites operated by NASA. Therefore, DLR's operational Global Ionosphere Flare Detection System (GIFDS), originally developed for flare monitoring and detection, has great potential to warn users in case of extreme flares with a delay of less than 1 minute. (Wenzel et al., 2016) Furthermore, GIFDS shall be used in the future to detect and monitor the precipitation of solar wind particles via magnetospheric coupling. The timely information on strength and expected dynamics of solar flare and particle precipitation activity is needed to ensure reliable terrestrial High Frequency (HF) communication, as these events cause increased signal absorption in the ionosphere up to black out. For example, in the case of transpolar flights, which usually use terrestrial RF communication, aircrafts can be warned in time to react accordingly, e.g. to change their route.
Vertical sounding measurements performed by ionosonde stations are being used to get information on the vertical electron density distribution. These data are therefore complementary to GNSS based sounding that provides a good horizontal resolution. In addition, ionospheric data from ISR facilities, in particular from the European Incoherent Scatter Scientific Association (EISCAT), will be used to study the underlying physics of complex phenomena, especially in relation to coupling processes from below and above. The world-class EISCAT facility provides an excellent data basis for testing and validating ionospheric models describing the polar ionosphere whose knowledge is a key issue for understanding the generation of propagation of ionospheric pertubations. The heating facilities can be used to study the impact on trans-ionospheric radio signals or to perform active experiments for better understanding the ionospheric impact on trans-ionospheric signal propagations. (Sato et al., 2018) EISCAT is currently building a next-generation radar that will enable 3D monitoring of the ionosphere. The new radar facility is capable of addressing new, significant science questions and can also be used to improve our understanding of space weather effects on technological systems like GNSS. The location of the radar within the auroral oval and at the edge of the stratospheric polar vortex is also ideal for studies of the long-term variability in the atmosphere and global change. (McCrea et al., 2015) The ground based observations group will be strongly involved with EISCAT to work at the forefront of science once the construction of EISCAT 3D is completed. Although RB measurements transmitted by several LEO satellites also provide TEC like GNSS measurements, RB data are able to provide a snapshot of the horizontal distribution of the ionospheric ionization along the satellite trace in the ionosphere, thus being also complementary to GNSS data. The geometry and the high spatial resolution make the data very attractive for regional tomography of the ionosphere and the detection

of travelling ionospheric disturbances.

2 Tasks / Objectives

This section is focussed on the analysis of system capabilities. It therefore describes the requirements and use cases of the new system.

2.1 Requirements analysis

The following is a list of requirements that will be used as a guidance during the development of the system and that describes features the new system must support.

Automated operational processing

To constantly process and produce data used for research, the first and most important requirement of the system is the automated operational processing of data. After the initial setup the system should run mostly unsupervised, manage incoming data and return and store the processing results. Therefore there have to be mechanisms that handle file transactions, exceptions, scheduling and garbage collection automatically. This instance of the system should also be strongly isolated to keep any external impacts from interfering with the system and guarantee a very high availability.

Postprocessing

In certain situations it is necessary to process a specified set of data or files again. In this process the configuration should be able to be changed, for instance to adapt the geospatial or temporal resolution. So in addition to the operational processing there has to be a postprocessing instance which gives the users the possibility to configure and trigger a workflow manually.

Module-based approach

The processing algorithms or programs of the systems should be encapsuled inside of modules. This approach aims to increase the modularity of the system and to enhance the maintainability of the system as well as the modules itself. It furthermore enables development of the modules with focus on the processing itself and leaves the communication between modules and file handling to the processing system.

Independent of programming language

The majority of algorithms will be implemented in Python, C, MATLAB, Fortran or Go. But to support modules written in an arbitrary programming language and to grant the users a free choice in terms of implementation, the system should be able to run modules regardless of the programming language. Therefore it has to be able to run isolated environments which provide all necessary dependencies, like interpreters, access to external libraries, configurations, etc.

File based processing

The system should be able to process files of nearly any type or extension. This includes handling of text and binary files, as well as handling archives, e.g. tarball or zip files.
Even though the concept is not part of this work, the system should optionally be able to be handle streams, since some of the modules, that are currently running in the operational processing environment, are based on stream-handling.

Matching of data

Many modules need data from different sources that are interdependent. The system should provide a mechanism to resolve and find files that match those dependencies. Therefore it should have a mechanism that allows the user to describe those dependencies.

Storage

The system should be able to store module input and output and logs temporarily over a defined period of time. This storage will be used for the file management of the operational processing environment. From this storage the processing results can then be moved or copied to a data archive for long-time storage. The concept of long-time archiving is not part of this work and will therefore not be explained in more detail.

Garbage collection

To avoid running out of memory or storage the system should provide a garbage collection mechanism, which is typically used to automatically clean up data or files that expired or are not used any more. This mechanism should automatically detect the expiration date of files based on their metadata or be configurable in terms of expiration date or duration and the type of resources that should be removed.

Error handling

The system should provide an exception handling system. This should not only be limited to the processing level but should also take care of system modules like for example the scheduling and the storage management. Furthermore the modules itself should be isolated, so that a failing process can not interfere with any routines of the system.

Distributed

The system should run in a distributed manner. This way the system should be able to compensate hardware and operating system failures and furthermore enable scale-out or horizontal scaling possibilities.

High availability

To guarantee a proper operational processing the system should provide a high degree of uptime or availability to reduce losses of data or processes. This not only includes error handling but also increasing availability of all system components, for example by clustering, sharding[2] or avoiding single points of failure, and reduce downtime through other impacts like crashes of processing hosts or simultaneous reboots of all hosts.

Isolation

The system should run inside of an isolated, dedicated network so the processes itself will not be influenced from any external impacts. This will also help to increase the systems availability and stability.

Monitoring and Logging

The system should provide a monitoring and an universal logging mechanism so that operators can control the function and the health status of the modules and the system itself.

[2]automatic splitting of data on several servers

The monitoring will give information about the system, like uptime, resource consumption (CPU, RAM, storage, etc.) and requests per second, which can be used in decision making about the scaling of the system.
The logging mechanism will provide a way to expose and analyze the logs of the processing jobs and the system itself to help finding potential issues or possibilities of improvement.

Simplicity and Transparency

The system should be easily configurable and should not expose implementation details to its users. For instance the user should not need to know the location (IP or hostname) of the systems components, like a database. It should therefore be encapsuled behind interfaces that simplify the users access.

Secure communication

The system should use secure communication protocols, for instance HTTPS to improve security and to prevent external attacks or leaks of sensitive data.

2.2 Users and roles

This section covers the use-cases of the processing system, which can mainly be categorized into two groups:

- Scientists
- Administrators

2.2.1 Scientist

The modules that are intended to be executed on the system shall be provided and implemented by the scientists. Additionally they should be able to integrate the modules in the system by setting up workflows and the modules dependencies. Once the module is integrated in the system and running correctly the scientists can review the produced results. They can then validate those to check if the module works the way it was intended to or export the results to use them for further processing.
Furthermore the scientists are able to use the system for postprocessing. To do that they have to provide configuration information used for the module

and start the process manually.
These use-cases are depicted in figure 1.

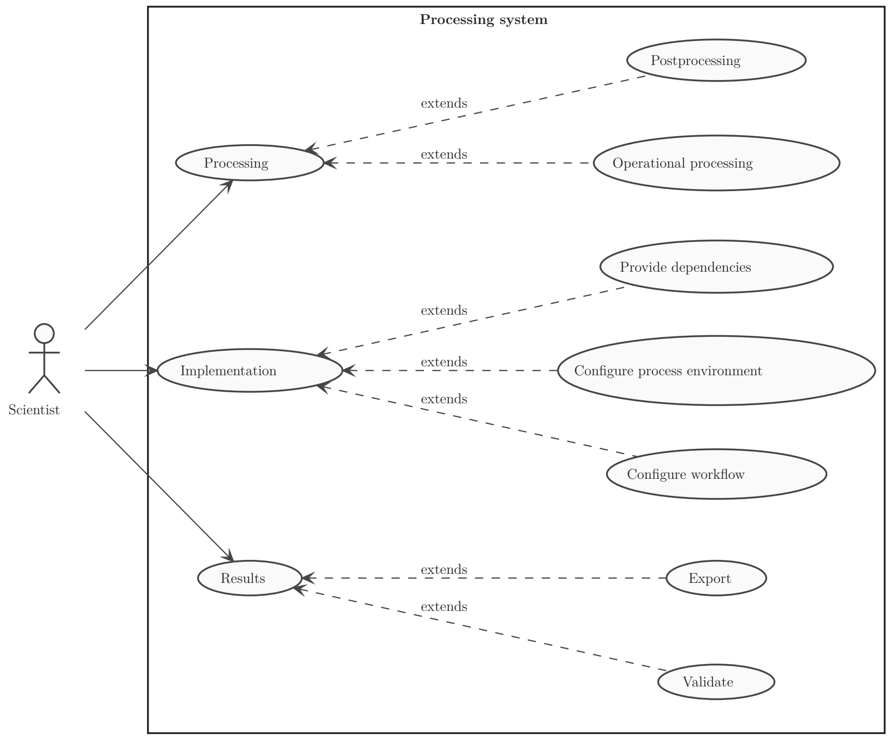


Figure 1: Use cases related to scientists

2.2.2 System administrator

The monitoring of the system shall be done by the administrators. This includes health monitoring of the system itself and all of the processes that are running on it. Additionally they can monitor the performance, which includes resource (e.g. CPU or RAM) utilization and process durations, so they can decide on scaling. In case of internal modules and processes failing they can also check logs to identify and solve problems.
The administrators are also responsible for setting up and maintaining the system, which includes management of hardware, hosts, virtual machines and software installations.
These use-cases can be seen in figure 2.

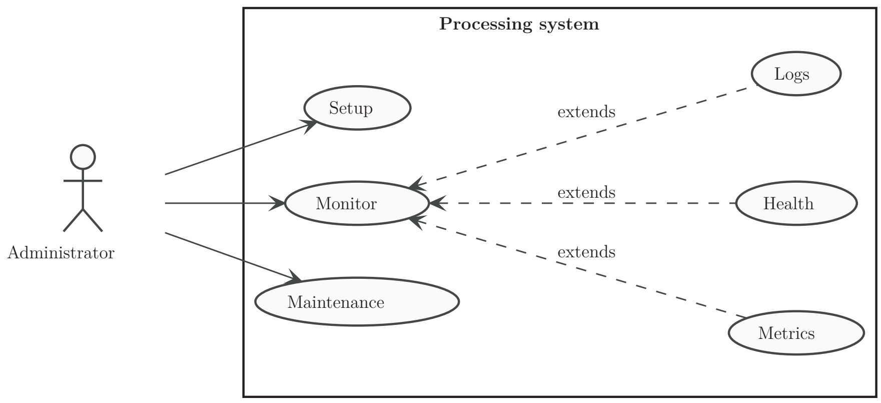


Figure 2: Use cases related to administrators

2.3 Storage requirements

A precise prediction of the amount of storage consumed in the system is not trivial, since the system must also be capable to process data from future missions. To give an idea of how the storage consumption could be predicted, the amount of data of two product types was predicted.
The first type are scintillation[3] products. The measuring stations produce around 25 Megabyte of raw data each day. Since there are currently eight stations involved, the amount of raw data for this product type amounts to 200 Megabytes per day. The processing results of this product are a series of plots with a size of around 330 Kilobytes per plot. The amount of plots depends on the configuration provided by the user. Since there are currently twelve plots configured, the output of the module amounts to about 4 Megabytes per station per day. This adds up to a total of 232 Megabytes per day for the scintillation product type.
Another module is the NPSM[4]. It depends on the date and time and the F10.7[5] value corresponding to that. Since the F10.7 values or tables will not be stored in the system, they are not relevant in this matter. Currently the output of the NPSM is about 9 Megabytes in size and gets computed every hour, which leads to a total of around 216 Megabytes per day.
In respect to the given values an average size of around 220 Megabytes per

[3]Diffraction and forward-scattering of trans-ionospheric radio signals caused by electron density irregularities and equatorial plasma bubbles (Basu et al., 1981; Kriegel et al., 2017)

[4]Neustrelitz Plasmasphere Model

[5]Solar radio flux at 10.7 cm (2800 MHz); can be used as indicator of solar activity, which correlates with the number of sunspots and UltraViolet and visible solar irradiance records (National Oceanic and Atmospheric Administration, 2019a)

product per day will be presumed. A system processing ten product types in a similiar scale would therefore produce around 2.2 Gigabytes of data per day, around 800 Gigabytes of data per year and around 8 Terabytes of data for a mission with a lifespan of 10 years.
Files will also be transferred to an archive storage. Thus not all files have to be kept inside the processing system, which will also help decrease the required amount of storage for the processing system.
While postprocessing data the incoming and outgoing traffic on the storage will be higher than in the operational environment since all the processing happens on a sequential basis instead of regularly scheduled processes. Therefore the storage instances for the operational and the postprocessing environments should be separated.

3 Existing systems

This section covers existing processing systems to get an overview over used architectures, strategies and best practices. It therefore focusses similiar projects or systems in the environment of DLR, including the *CHAMP processing system* (Wehrenpfennig, 2002), *BACARDI* (Stoffers et al., 2019) and the DIMS (Böttcher et al., 2001).

3.1 CHAMP Processing System

The CHAMP processing system was designed for automatic processing of space based radio occultation and topside navigation data measurements onboad CHAMP combined with ground based data files, but was also used for processing for data provided by the GRACE satellite mission. Data is entirely provided in form of files which are obtained by the system via FTP through the Intra- and Internet.

3.1.1 Architecture

The architecture of CHAMP processing system is displayed in figure 3. The system consists of five components:

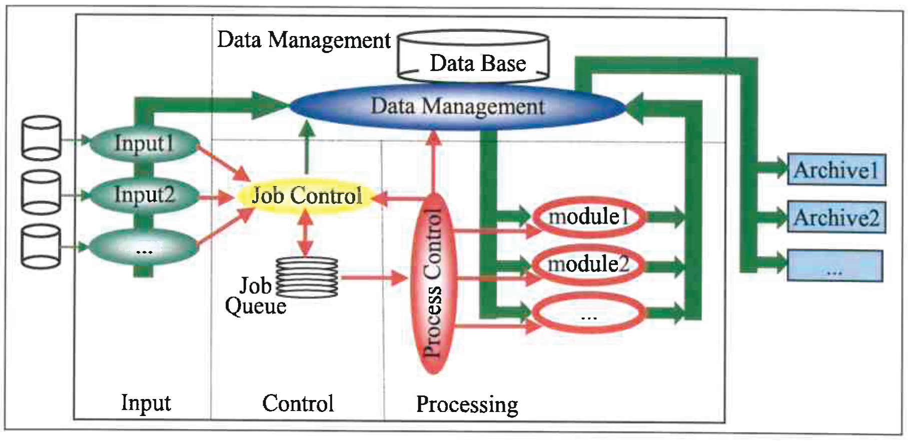


Figure 3: System architecture (Wehrenpfennig, 2002)

Data management Temporary data storage, control of data output of the system

Input Reception of input data for the processing

Job Control Combination of input data for task (job) management

Process Control Spawns and monitors processes

Supervising Command input (Operator); output of system status information

The jobs in this system are triggered by the arrival of new input data or time events. The application modules are started by the system when all input files for a processing step are available. Therefore the dependencies of a job are described in a relation file.
Jobs whose dependencies are resolved and thus all needed input files are available are delivered to a job queue which submits them to the processing system where the jobs are executed.

3.1.2 Naming conventions

The foundation for the automatic assignment of input files to related jobs are the systems naming conventions for files and jobs. By using matching patterns described in the *relations.env* file the job control system looks for all jobs in a related job queue to find jobs matching the rules.
Product names must be unique for a products and jobs. The order of the attributes in the file name is significant.
The first term of the name is the product type. It is followed by additional information, for instance subtypes, separated by a plus sign.
Furthermore the file names will contain time information separated by an underscore.
After that an arbitrary number of additional attributes can follow.
Those naming conventions are shown in table 1.

<type>	<info>	YYYY	DOY	HH	...
Product type	Subtype	Year	Day of year	Hour	Add. arguments

Table 1: Naming conventions (Wehrenpfennig, 2002)

3.1.3 Configuration and Initialisation files

The configuration and initialisation of the processes and the system in general is realised by using *.env* and *.ini* files. An *.env* or *.ini* file contains key-value-pairs closed by a semicolon sign. The following files will be automaticly generated and used as default. They can be adjusted to fit the needs of the process.

global.env Configuration file for most of the system modules containing global parameters like paths and names

local.env Contains switches / parameters used for applications

local.ini Contains parameters passed by the controlling system to an application

An example of a configuration is shown in listing 1.

```
[ INPUT ]
  INFILE$0 = INPUT/gps_2001_23;
  INFILE$1 = INPUT/leo_2001_23_06;
  INFILE$2 = INPUT/HK_2001_23_06;

[ OUPUT ]
  # name of the output directory
  OUTDIR = OUTPUT;
  # base names of products
  PRODUCT$0 = OUTPUT/CH-AI-1-MRR;
  PRODUCT$1 = OUTPUT/CH-AI-1-LRR;
  # full name of a product, if it can't be derived from the job
   ↪  that has to be processed
  OUTFILE$0 = HRR+2001_23_05;

[ ATTRIBUTE ]
  # list of attributes passed by the controlling system
  ATTR$0 = RAPID; # type
  ATTR$1 = 2000;  # year
  ATTR$2 = 00;    # doy
  ATTR$3 = 01;    # hour
  ATTR$4 = 126;   # measurement id
```

Listing 1: Example *.ini* file (Wehrenpfennig, 2002)

3.1.4 Definition of data dependencies

The relations or the dependencies between files are described in an *relations.env* file. Those files also use the syntax described in 3.1.3. An example for a *relations.env* file is shown in listing 2.

```
# input dependencies of jobs and assignment description
# match pattern example
#   '*_=_!_[0-3]_{-0+0}[0-365]_{0+23}[0-23]
# <all>_<exact-num>_<exact-string>_<range>_<offset-range>

# begin of section of jobtype LRGEN
[ LRGEN ]
  # attributes of the job:
  # <subtype>_<YYYY>_<DOY>_<HH>;
  # list of data types to receive with job attribute match
   ↪  pattern
  INTYPE$0 = iCH-AI-1-LR *_=_=_=;
  # if OVERWR is set, new receive input data may overwrite
   ↪  older data
  OVERWR$0 = ;

# begin of section of jobtype PROCC
[ PROCC ]
  # attributes of the job:
  # <subtype>_<YYYY>_<DOY>_<HH>;
  # list of data types to receive with job attribute match
   ↪  pattern
  INTYPE$0 = PSO-LEO *_=_{-1+2};
  INTYPE$1 = PSO-GPS *_=_=;
  INTYPE$2 = CH-AI-1-MR *_=_=;
  # if OVERWR is set, new receive input data may overwrite
   ↪  older data
  OVERWR$0 = ;
  OVERWR$1 = ;
  OVERWR$2 = ;
```

Listing 2: Example *relations.env* file

3.2 BACARDI

The Backbone Catalogue of Relational Debris Information or short BACARDI is a system developed by the German Space Operation Center (GSOC) and DLR Simulation and Software Technology to track cooperative and uncooperative orbital objects.
The main goal of this project is to provide a unified database containing orbit information of active satellites and space debris in Earth's orbit. In this context the focus lies on the development of an orbit database (High precision and integrity), an independent path determination from sensor data and the support of mission operations at GSOC in collision avoidance. Additionally the development and optimization of algorithms, like observation correlation, orbit propagation and determination, and detection and forecast of maneuvers, fragmentations an re-entries, is targeted.

3.2.1 Architecture

The system's architecture is depicted in figure 4.

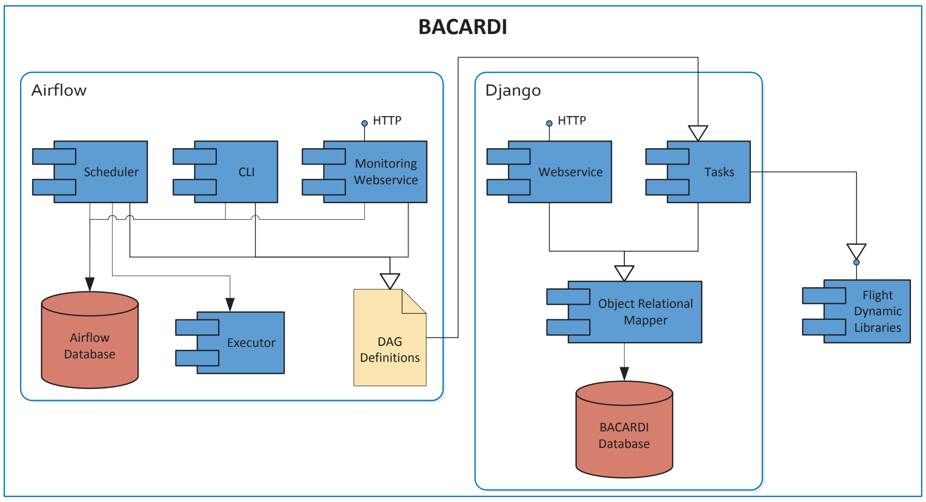


Figure 4: System architecture (Stoffers et al., 2019)

The system consists of three main components, which are loosely coupled to ensure horizontal scaling possibilities. Those components are:

- Django
- Airflow
- Flight Dynamic Libraries

The *Django* component builds the core of the system. It contains the main data model, which is definied using the Django[6] Object Relational Mapper (ORM). This means the data model is described using Python classes and will be mapped to tables in a relational database.
Additionally the Django component includes a tasks module, which provides callable tasks that provide atomic activities or methods to interact with the data model. Those tasks can be categorized into three groups:

Importer is a task that imports data from a source into the BACARDI database.

Exporter is a task that exports data from the BACARDI database into a specified format.

Processor is a task that executes computations or calculations on the data stored inside the BACARDI database.

Furthermore the Django component contains a web service component, which uses Django to provide access to the database and the systems tasks in form of a web-based API.
The *Flight Dynamic Library* component contains algorithms and functions written in Fortran that will be used or called with Python wrapper functions implemented using the F2x library (See `https://f2x.readthedocs.io/en/latest/`).
The *Airflow* component uses the open source Python framework *Apache Airflow* to handle task scheduling and processing. It's main purposeses in this system are the monitoring of workflow changes, the timely scheduling of processor workflows and the execution of unscheduled processes via it's web service or CLI. The Apache Airflow framework will be described in more detail in section 5.4.1.

[6]Open source Python web framework

3.3 DIMS

The Data Information and Management System (DIMS) is a multi-mission system developed in cooperation between Werum Software and Systems AG and the German Remote Sensing Data Centre (DLR-DFD). It is used for processing, archiving and distributing earth observation products.

3.3.1 Architecture

The DIMS consists of multiple components, which are shown in figure 5 and are briefly explained as follows:

Product Generation and Delivery is responsible for providing processing results to the user by making them accessible via FTP or similiar services.

User Information Services provides an interface for the user capable of ordering products or processing requests.

Product Library contains all input and output products and is used for providing those to the operational processing as well as archiving those products. (Long time archiving)

Order Management is used to manage users and their credits. It therefore decides over approval of product or production requests and forwards those to the production control.

Production Control generates request trees out of requests forwarded by the order management to examine required processing chains, which are passed to the Processing or Post-Processing System afterwards.

Processing System is used to start or trigger the processes or algorithms themselves.

Post-Processing System is responsible for processing post-processing requests.

Ingestion System is responsible for synchronizing external data from multiple sources and source types with the product library.

Operating Tool provides monitoring and controlling tools for the system.

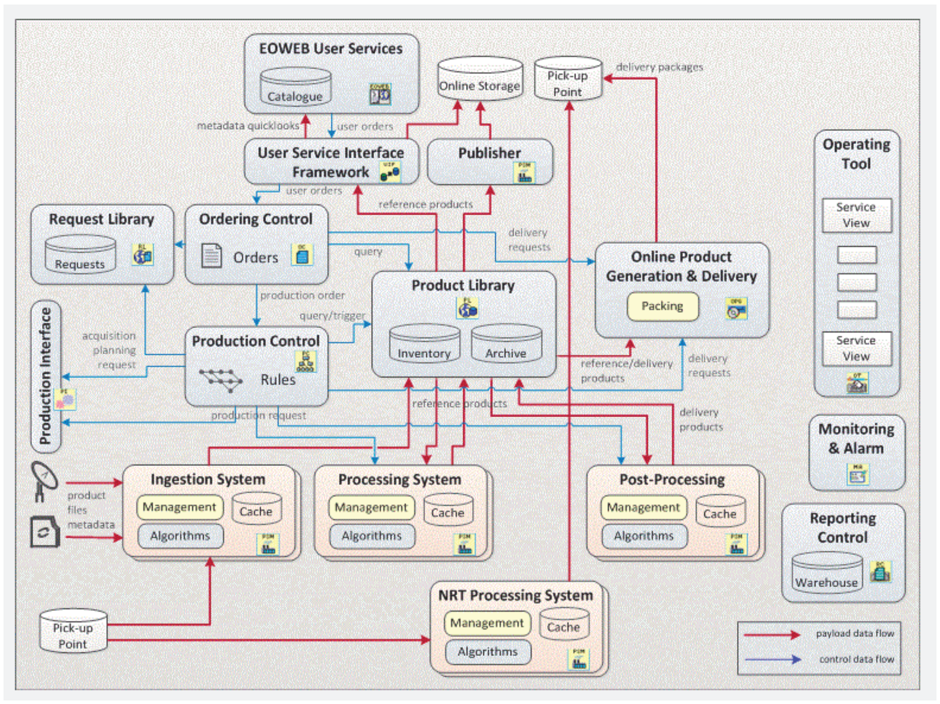


Figure 5: Processing systems in DIMS (functional view) (Böttcher et al., 2001)

3.3.2 PSM

One of the DIMS components is the Processing System Management (PSM), which is implemented using Java. It is used to manage processing jobs or instances and provides an interface between the processing and archiving mechanisms and the processing algorithms. A general overview of the components used with the PSM is shown in figure 6.

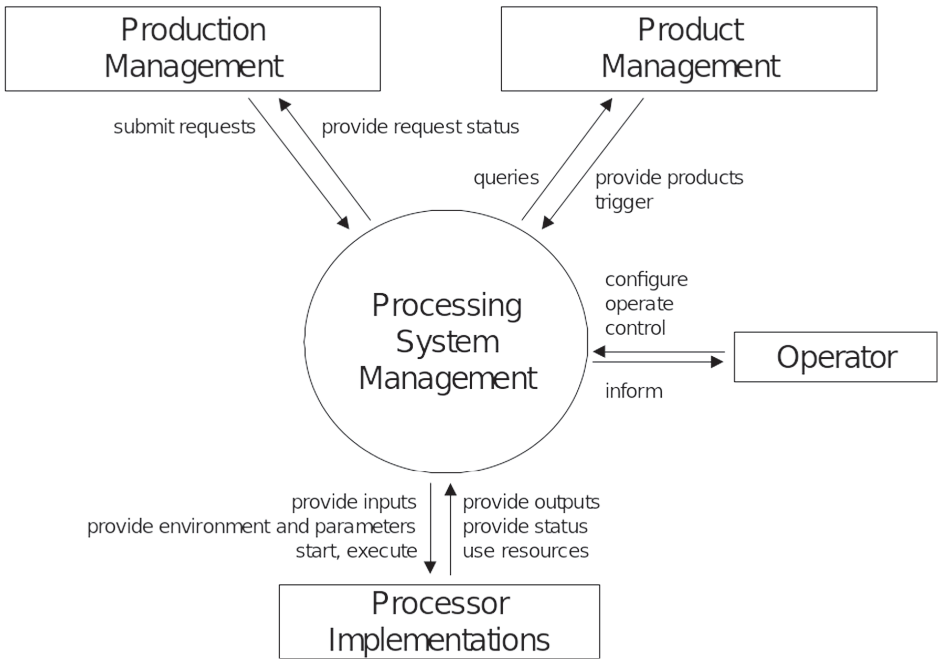


Figure 6: PSM system architecture (Böttcher, 2004)

The most common functionalities of the PSM are the queueing of production requests, the interaction with the DIMS Product Library, including retrieval of input products and transfer of output products, and, together with the OT (Operating Tool), the provision of a graphical user interface to monitor and control the processing system. Additionally it provides autonomous operation capabilities, including product cache management and pre-caching of input products, as well as a rule based workflow control and processor charging, scheduling and load balancing for multiple processors.
The PSM supports multiple scenarios for triggering processors, which include the systematic ingestion of products into the Product library, processes that are triggered on a timely basis or by the Product library, processes that are triggered on demand and are initiated by a user's order and processes that are used to transfer products from the Product Library to external destinations. Those scenarios are shown in figure 7.

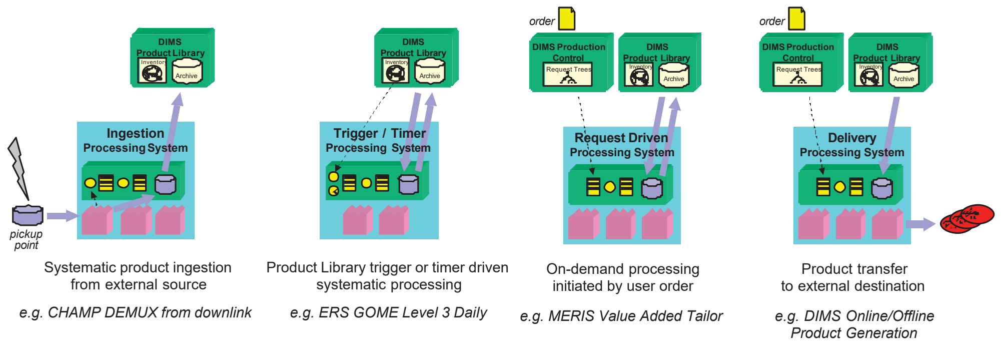

DIMS Product Library
Inventory
Archive
Ingestion Processing System
pickup point
Systematic product ingestion from external source
e.g. CHAMP DEMUX from downlink
Trigger / Timer Processing System
Product Library trigger or timer driven systematic processing
e.g. ERS GOME Level 3 Daily
order
DIMS Production Control
Request Trees
Request Driven Processing System
On-demand processing initiated by user order
e.g. MERIS Value Added Tailor
Delivery Processing System
Product transfer to external destination
e.g. DIMS Online/Offline Product Generation

Figure 7: PSM trigger scenarios (Böttcher et al., 2001)

The processors itself are implemented using a Java interface, which defines methods a processor has to provide to be run correctly in the PSM environment. This way the processors are standardized. The processors can be executed as Java functions or shell scripts.
The dependencies between processes or the workflows can be described in an XML representation.

3.4 Summary

The described systems provide a simple overview over processing systems in the environment of DLR. Some of the concepts mentioned in those systems should be used as an inspiration for the development of the new system. This includes common concepts, like the job and module based structure of all systems, the possibility to describe dependencies or relationships between different product types, as shown in the CHAMP processing system, and the use of a centralized database to store all products available in the system, as shown in the BACARDI project. Additionally the abilities of the DIMS, namely to describe processing chains as workflows and to trigger jobs based on a timed schedule as well as on other events like incoming data, should be integrated into the new system.
Even though the DIMS already provides a way to run algorithms written in various programming languages, the concept should be generalized to allow the use of arbitrary languages. Furthermore since the new system should process file-based data, the concept of storing data inside the central database has to be adjusted. Additionally it would be desirable to improve the ease of use by generalizing the way of describing interdependent files.

4 Proposed system concept

The following chapter deals with the concepts and the architecture of the new system.

4.1 Structure

This section focusses on the new systems architecture and structure as well as the components architecture and interplay.

4.1.1 Overview

A general architectural overview of the new system is depicted in figure 8.

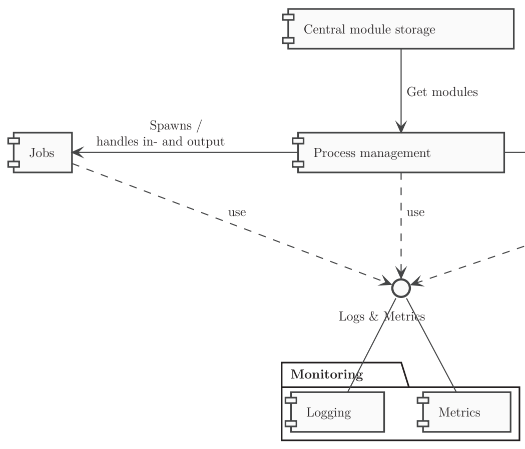


Figure 8: Component overview

The architecture of the system aims at a loose coupling of the components, so that individual components can be replaced by other implementations. This can be benefitial in performance optimization, e.g. by replacing the used database, or in case a certain software is no longer supported.
The system consists of the following components:

Process management including the workflow engine and the scheduler,

Data management including the metadata database and the file storage,

Central module storage which is used to store and distribute all modules that will be used in the processing system,

the jobs which will be spawned by the process management and used to run the modules and

the monitoring system which includes mechanisms to provide process logs and metrics of all components.

The process management will be responsible for monitoring the data management for changes and triggering or spawning jobs on a timely basis or on events utilizing the central module storage. Therefore certain dependencies have to be defined which describe the events on which a job has to be started. Furthermore the process management handles the injection of input data into the job's modules and the transfer of output data to the data management. Every component in the system will expose logs and metrics to an interface which collects those and exposes them to the administrators.

4.1.2 Process management

The process management will consist out of two components: a scheduler and a workflow engine.
The scheduler will be responsible for spawning jobs based on certain criteria. This can for example be a time interval or an event, e.g. incoming data. Every job in the system has to define those criterias to tell the scheduler, when the jobs need to be run. Since the system will run in a distributed architecture, the scheduler must also have the possibility to check health and resource availability of the systems nodes to ensure a proper scheduling on the processing nodes.
To guarantee that jobs can be spawned in case of incoming data, the process management has to communicate with the data management and watch or subscribe to changes.
The workflow engine will be responsible for combining modules or tasks into workflows. Since most workflows will be based on more than one task, e.g. download an external file, prepare the data and process it, there has to be a mechanism which allows configuration of chains and executes the processes in order as well as enabling communication between the modules of a workflow. This includes handling text output and temporarily storing file outputs.
The interplay of the components is shown figure 9.

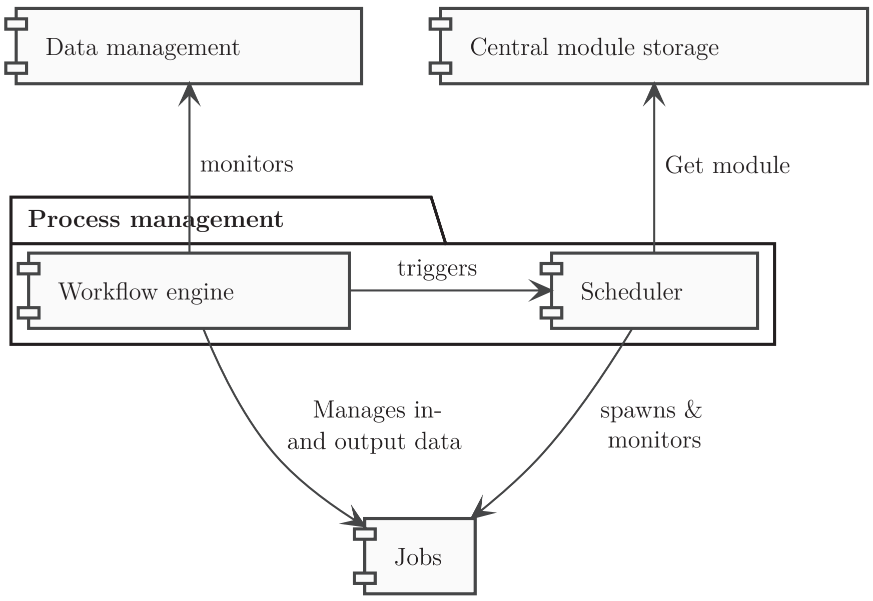


Figure 9: Process management components interplay

4.1.3 Data management

The data management will also include two components.
The first one is the file storage, which will be used to store any product file in the system. Those files can be mirrored from external sources or output products of finished processes inside the system. Since the process management will have to monitor and react to changes in the filesystem, there has to be a mechanism that supports this feature. Furthermore since the system will output temporary files, a garbage collection system must be available.
The file storage will have several interfaces that allow the modules to access, archive and human interactions. For the use by the modules a common protocol like FTP or HTTP should be used. In addition, a user interface via HTTP should be provided to allow the user to explore and download data.
Archiving should not necessarily be part of data management, but rather be implemented as an external module, e.g. as a scheduled and configurable cron job.
The second component of the data management will be a database. This database will be used to store the metadata for the files that are stored inside the local file storage. This includes the files path or location, defined parameters, like for instance the period of validity for a product, and additional metadata that will be extracted from each file. This metadata database will enable queries for files that are interdependent.
The general interplay of the components is shown in figure 10.

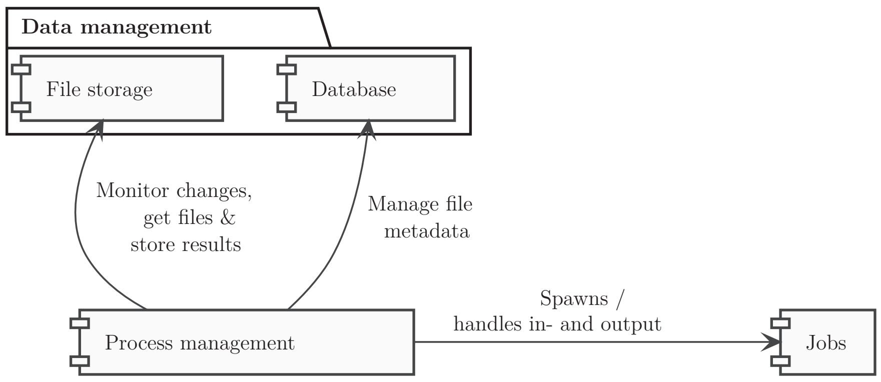


Figure 10: Data management components interplay

4.1.4 Monitoring

The monitoring system will include a logging and a metrics mechanism. Those will be centralized, retrieve the information of all system components through defined interfaces, store the results temporarily in an own storage instance and expose the data via individual web frontends.
The logs can be used to identify and resolve problems or bugs in the modules. The metrics will be an indicator of the systems performance, e.g. CPU or RAM utilization. They will help in decision making regarding the system's scaling.
This can be seen in figure 11.

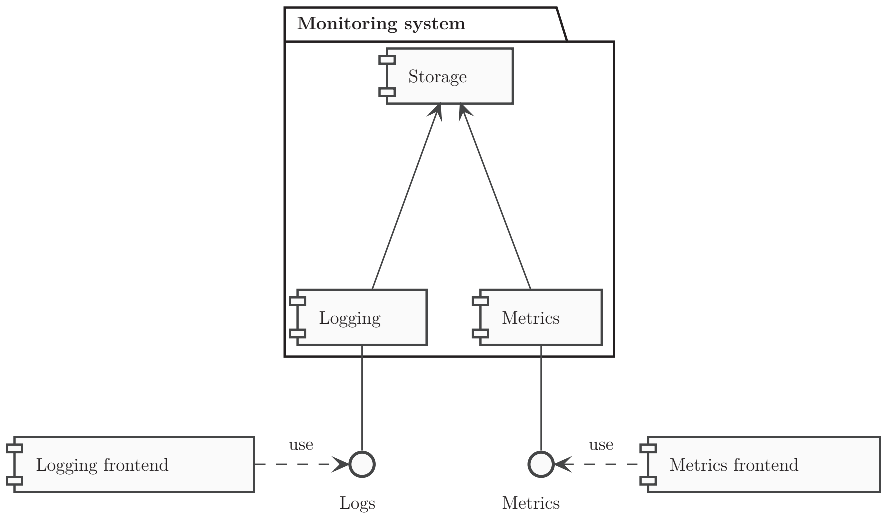


Figure 11: Monitoring components interplay

4.1.5 Central module storage

Each module has to be encapsulated inside an isolated environment, providing all of the algorithms or programs dependencies, e.g. interpreter, libraries, configurations, etc. This approach will prevent version conflicts or compatibility problems when running multiple modules, while also keeping the amount of software and libraries installed on the processing nodes minimal.
To make those environments accessible to all processing nodes they will be stored on a remote software or module storage. That enables the dynamic use of the environments while processing without the need of storing them on each node permanently.

4.2 System environments

According to the requirements the system must provide different environments for each of the following purposes:

- Operational processing
- Testing
- Postprocessing

Those environments have to be isolated so they can't interfere with each other.
As the name suggests the operational processing environment will be used for just that. Only modules and workflows that were tested and approved to run will be deployed there.
The testing environment will be used to test and prepare the modules and their respective configuration for the operational processing. In this environment the modules can be analyzed - for example with regard to resource consumption (CPU, Memory, etc) - and the correct behaviour of the processed can be ensured.
The postprocessing environment will be used for manually triggered workflows. This can for instance be useful if data has to be processed once again for a certain time period.

4.3 Naming conventions

The naming conventions of the new system will be according to the DLR naming convention guidelines. All file names will contain informations about

the producer, the data acquisition source and method, the product level, type and mode and the year, month, day, hour and minutes and seconds for the start and end time of the products validity period.
Those naming conventions are shown in detail in table 3 in appendix A.2. A filename could therefore look like the following example:

```
DLR_GNSS_GCG_L2_DIXV02_NC_EUROPE_2016-01-30T00-00-00_2016-01-30T00-05-00_030_D.nc
```

4.4 Handling of interdependent datasets

The resolving of dependencies between datasets is an important requirement of the system. (See section 2.1) As described with the CHAMP processing system (see section 3.1.4), the new system has to provide a mechanism that can be used to describe dependencies between datasets and automatically resolve those by finding and providing files that fulfill the specified relations. Therefore there has to be a possibilitie for the user to define rules which describe the relations of files.
The new system will for this purpose rely on a database, which will be used to store metadata about the files currently stored in the system. Therefore each new file has to undergo a decoding process. This process will extract the metadata that will be stored. Furthermore there has to be a process that keeps the file storage and the metadata base in sync, e.g. delete metadata of files that are not present in the system anymore.
This database will enable the user to describe relations between files with common query languages like SQL. The sequence of the resolving process is depicted in figure 12.

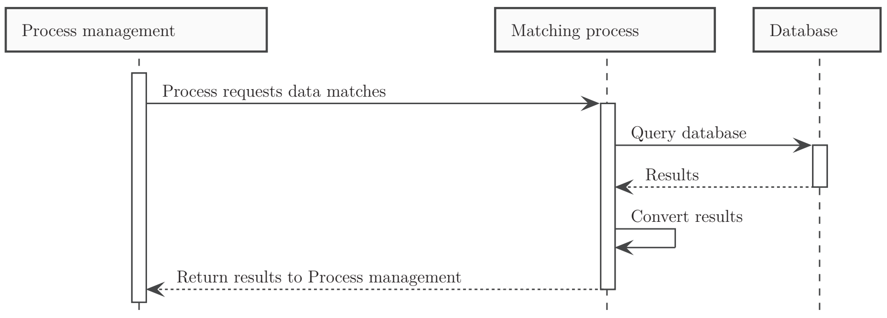


Figure 12: Sequence of resolving process

When the results are returned to the process management it has to decide how to proceed. Based on the amount of returned results or matches, the following processes could be skipped or one or more processes will be spawned.

4.5 Metadata datamodel

The metadata model of the system will be rather simplistic and universal. The goal is to implement a data model that can contain an arbitrary amount of metadata about the files without the need to change the data model when establishing new attributes. Therefore a dynamic data model will be used. It will contain static attributes, like the index, the location of the file and a product time, which indicates or represents the files validity. Additionally the data model will include an attribute that can take arbitrary data. This model is shown in figure 13.

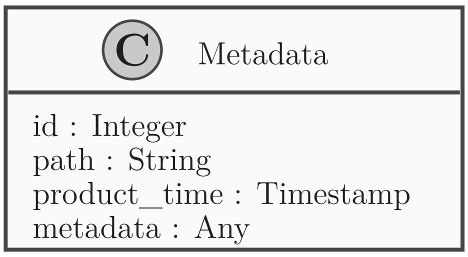


Figure 13: Data model schema

There will be a table for every product type stored in the system, which will be derived from the data model shown in figure 13. The name of the table will be equal to the name of the product type.

4.6 Management jobs

There are jobs in the system that will run in the same way as a user-defined job, but they will be used to handle certain aspects of the system, which the user should not get in contact with. They will be created and managed by the systems administrators and developers.
These jobs are used to reduce the complexity of the configuration process for the user. They offer increased simplicity and transparency.

4.6.1 Data management

The process of data and metadata handling is an example for a management job. It will be handled in the background as soon as a user-defined job or an external source produces data that will be stored inside the system. This data will be called artifacts in the following sections. Those artifacts will be stored in a dedicated storage area.
There are different ways in which incoming data can be handled. These are

shown in figure 14.
When new files are stored in the file storage the system has to respond to those changes. Therefore it has to detect these.
At first the job will generate a file name prefix, which will follow the naming conventions of the system. It will then copy the file from the artifact storage to its destination storage location.
Afterwards a decoding module will be started if the metadata of this file type has to be stored. The decoding module will extract all necessary metadata and return them as a result. Afterwards they will be submitted and stored into the database.
This step will be skipped if the metadata of the incoming data does not need to be stored.

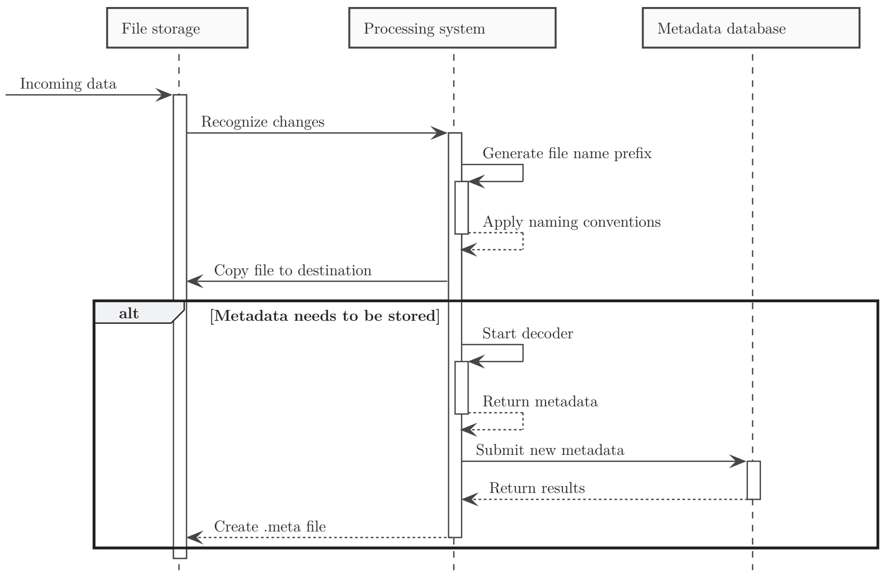


Figure 14: Incoming data sequence

This job also needs to have exception handling, in case any of the modules fails, including communication errors with the database or errors that occur while writing to the storage, etc. This should trigger the process management to retry executing the job a certain amount of times and clean up and give out notifications to the administrators if the amount of retries is exceeded.

4.6.2 Garbage collection

To avoid storing files that are not frequently or are not used at all in the system a periodically running management job will be established that is responsible of cleaning up those files. This kind of task is commonly known

as a garbage collection. The garbage collector will be responsible for cleaning up resources like for instance artifacts that are not longer needed.
The metadata database will contain informations about the expiration of file types. When this date passes the files will be deleted from the storage system. The process manag then will detect those changes and remove the corresponding metadata from the metadata database.
This process is shown in figure 15.

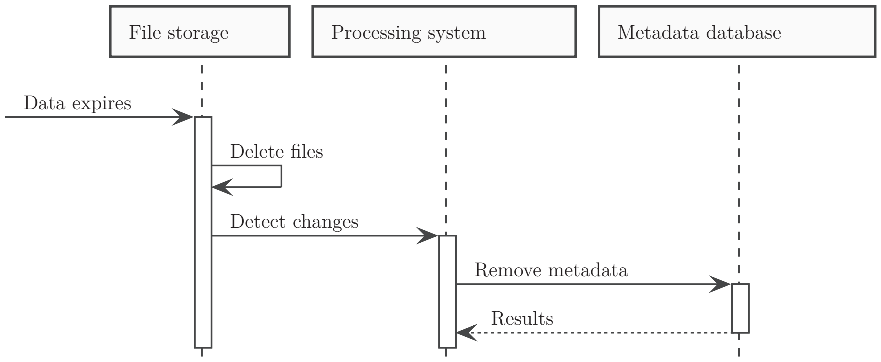


Figure 15: Garbage collection sequence

4.7 Creation of new modules

Modules are the building blocks of the system. Every algorithm or process that will run in the system will be provided as a module. A new module can be integrated into the system by following three steps:

Implementation The process or algorithm has to be implementede. Ideally it should provide configuration possibilities via configuration files or CLI options and arguments.

Environment The implemented module has to be wrapped in an environment, so that it can be uploaded to the module storage and can therefore be utilized by the systems hosts.

Workflow A new workflow has to be configured to use the provided module. For that parameters like the expected input location and the location of output files or directories have to be known and specified.

An interface to describe a workflow step is shown in figure 16.

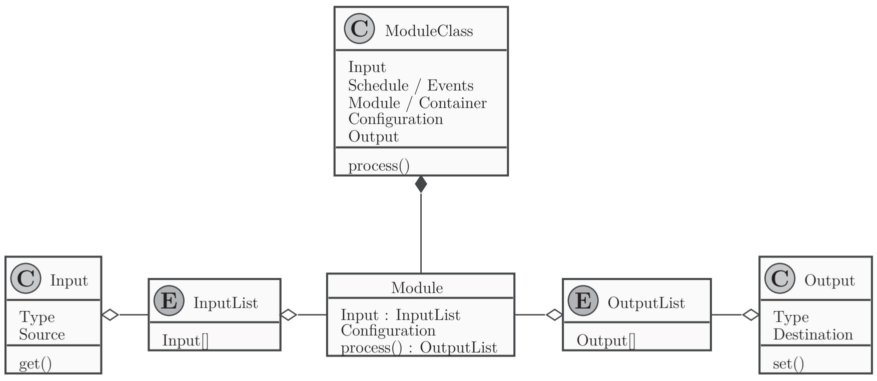


Figure 16: Overview of the module definition

The configuration includes the input, output and the module itself.
The input will be specified by type and source of the file. The inputs will then be put in a list. This list can also be empty, meaning that a process does not depend on any input.
The output will be specified with a type as well. Additionally it has a destination attribute, which describes, where the output will be stored. The output also gets stored in a list, which can be empty, meaning that the process does not produce any output files that need to be stored.
The module itself will be configured using the input and output lists. Furthermore it will take parameters like the identification of the used module and the schedule or events it has to be triggered on.
Additionally to defining the modules dependencies in the workflow configuration, the module itself must implement a method for the injection of the input. This should be done by using CLI arguments or options. This way the input files can be provided on file system paths, which will then be handed to the module. An example process can be called as shown in listing 3.

```
example-module -f /tmp/inputfile.txt -o /tmp/output
```

Listing 3: Example module call

This call starts the *example-module* with the input flag pointing to the file available on the path `/tmp/inputfile.txt` and specifies the output directory under `/tmp/output`.
CLI options and arguments can be implemented rather simply, but the implementation itself depends on the used programming language. An example implementation for Python is shown in listing 4 and an example implementation for C/C++ is shown in listing 5.

```
import click


@click.command()
@click.option("-f", "-file", help="Define input file")
@click.option("-o", "-output", help="Define output
 ↪  directory")
def main(file, output):
    print(file)
    print(output)
```

Listing 4: Python CLI options using Click

Using the *Click* Python library CLI options and arguments can be implemented using decorators as shown in listing 4. Python decorators are wrappers around functions, that can be used to modify the way a function is called. In this case Click injects CLI options and arguments into the function.
Click is only an example for an argument parsing library. Other examples are *argparse* or *optparse*.

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int opt;

    while((opt = getopt(argc, argv, "fo:")) != -1) {
        switch(opt) {
            case "f":
                printf("Input-File: %c", opt);
                break;
            case "o":
                printf("Output: %c", opt);
                break;
            case "?":
                printf("Unknown option");
                break;
        }
    }

    return 0;
}
```

Listing 5: C CLI options using getopt

A program written in C can retrieve options and arguments by using the *getopt* function as shown in listing 5. It can be used to loop over all input parameters and further process them, e.g. store the values in variables.
Other libraries used for argument parsing in the C programming language are for instance *argp* or *opt*.

4.8 User-defined modules

The user-defined modules in the system can be categorized into four groups.

- No dependencies
- One dependency
- Two or more dependencies
- External dependencies

Modules without dependencies do not depend on any files that need to be processed. Those modules can for instance be used to synchronize an external data source with the file storage.
Modules with one dependency do not necessarily need to query the metadata database for matching files but can be started as soon as the needed file arrives.
Modules with two or more dependencies most likely will have to query the metadata database for matching files. This query than returns the paths to the corresponding files so they can be transferred.
Modules with external dependencies can download those external files directly. Those files can not be filtered by the metadata database.
The example in figure 17 shows the general workflow for each module.
If an event occurs, e.g. new incoming data (see section 4.6.1), the corresponding modules will be started.
At first there has to be a check for the data dependencies of the started module.
If the module needs a single file or multiple files that aren't related to each other, the files can just be requested from the file storage.
If it needs multiple interdependent files, those files have to be filtered and matched. For that a query on the database will be used. The result will either be the paths of the corresponding files or a signal for the module, e.g. an empty list, that there are currently no files that resolve those dependencies, so the module gets stopped. If the query is successful and returns the paths, the files will be requested from the file storage and will be handed to the module.
If there are any dependencies on external files, e.g. from FTP- / HTTP-Server, they will be downloaded and handed to the module as well.
Afterwards the module will start, process the files and return the results, which will then be stored in the file storage.

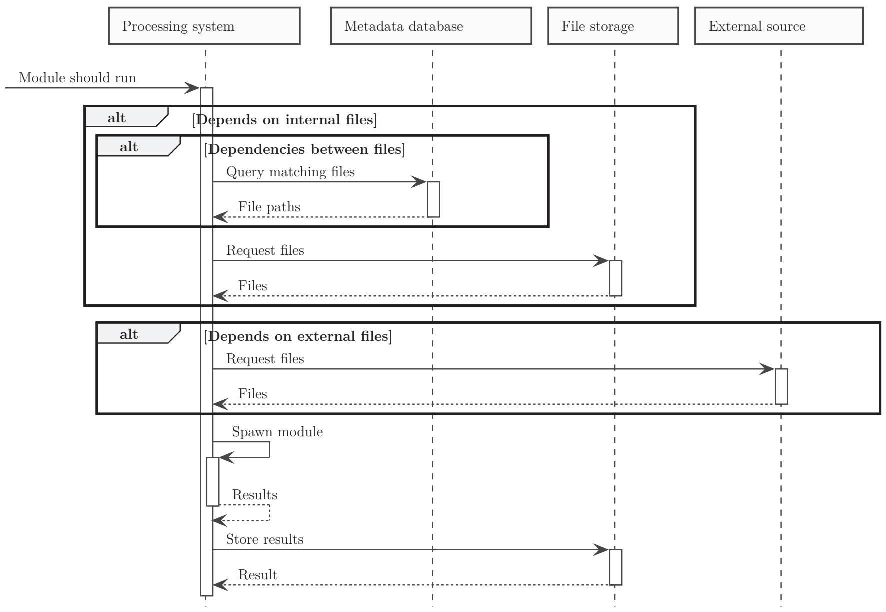


Figure 17: General sequence

4.9 Postprocessing

The postprocessing sequence just slightly differs from the general workflow. The main difference is that the process can be configured individually. Therefore the postprocessing can be used to set up processing workflows for certain files, time periods or other criteria. This way a workflow can for instance be configured to process data that was measured in a specific timespan or to process data in another resolution - temporal, geospatial, etc.
This sequence is shown in figure 18.

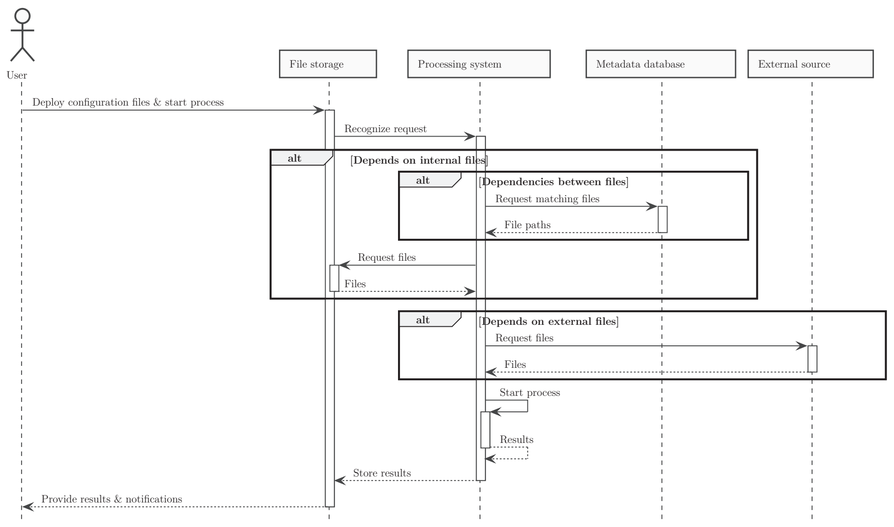


Figure 18: Postprocessing sequence

5 Supporting technologies

This section covers technologies that represent the state of the art and can be used for the implementation of a distributed processing system.

5.1 Containers

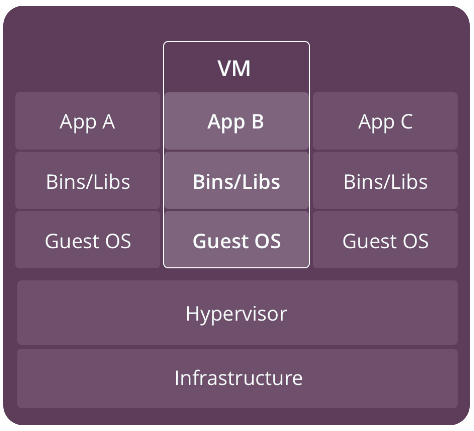
Containers are used for OS-level virtualization. They are isolated environments that bundle software, libraries, configurations and other dependencies and are executed by a container engine or a container runtime. Since containers are run by a single operating-system kernel, they are more lightweight than Virtual machines which is shown in table 2 by comparing resource consumption between KVM[7] and Docker while running a web server. (See Chae et al., 2019)

	KVM	Docker
Average idle of CPU	47.52 %	75.79 %
Average memory usage	740 MB	248 MB

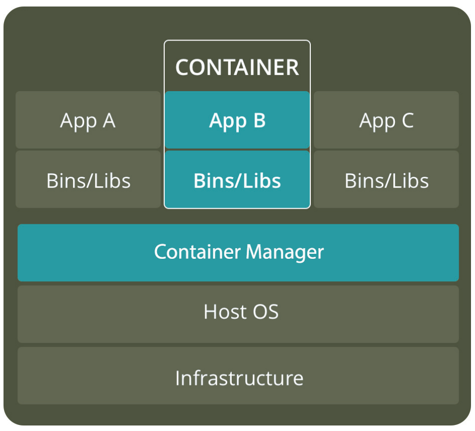
Table 2: Performance comparison of KVM and Docker while running a web server (Chae et al., 2019)

Figure 19 shows the architectural differences between containers and virtual machines.

[7]Kernel Virtual Machine; software providing a full virtualization solution



(a) Virtual machines architecture



(b) Container architecture

Figure 19: Architectural differences between Containers and Virtual Machines (Bauer, 2018)

Containers are created out of images, that contain all necessary parts of the virtualized operating system, except the kernel. They are split up into multiple layers, which describe the dependencies and are read-only. New layers can be added by using an existing image as a base image and adding new dependencies. When a container gets created out of an image, it adds a writable layer on top of the image, which takes the temporary changes of the container. This architecture is shown in figure 20. (See Liebel, 2017)

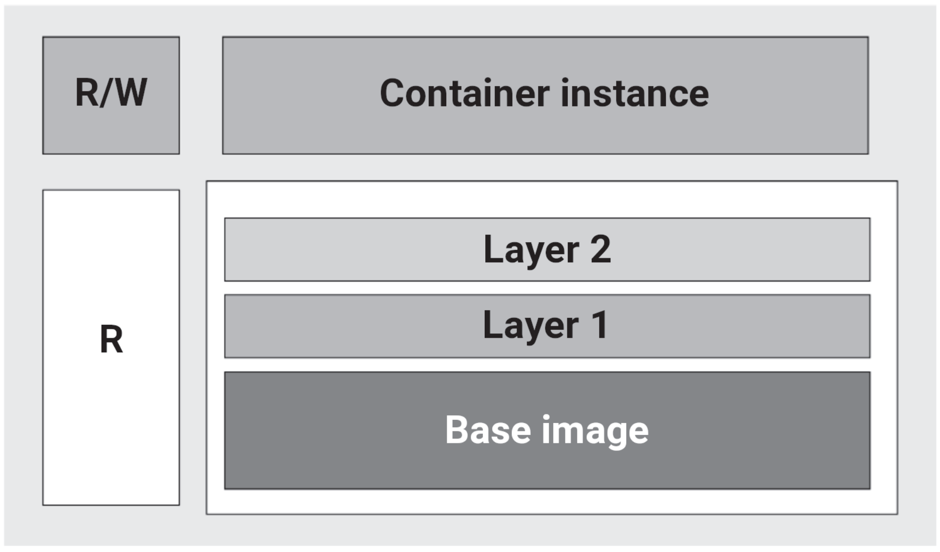


Figure 20: Layered architecture of container images (Own illustration based on Liebel, 2017)

Images can also be versioned, which means that multiple instances or versions of an image can exist. This can be used to provide multiple versions of the software or libraries inside the image (e.g. `python:2.7.17` and `python:3.7.5`) or to provide images with different base images or operating systems (e.g. `python:3.7.5-buster` and `python:3.7.5-alpine`).
To make images accessible to other hosts they can be uploaded and stored on a remote image storage solution, also known as image or container *registry*. There are multiple providers and implementations for cloud- and self-hosted container registries, including the *Docker Registry* (or *Docker Hub*), *Google Cloud Container Registry*, *GitLab Container Registry*, etc.
Since containers are isolated environments which provide all necessary dependencies of the software, they increase reproducibility and enable the user to run nearly any kind of software that is provided in form of a container on systems that have a container engine installed. Therefore, no additional software, like module dependencies or libaries, need to be installed on the executing hosts.
Most container engines store the logs of the containers on the host system. This enables the aggregation and forwarding of the logs to a central logging component.
The goal is to provide every module of the system as a container to create the isolated environments the modules will be encapsulated in. This way some of the requirements for the system, including *choice of implementation language*, *error handling* and *logging*, will be faced or even fulfilled.

5.2 Container Engines

As described in the previous section every module should run as a container, which enables the user to be free regarding the choice of programming language. Therefore the system must provide a container engine or runtime to be able to execute containers or container images. With respect to figure 8 the container engine will provide the modules environments and execute the modules. Therefore this section focusses on some of the currently available container engines and describes differences between them.

5.2.1 Docker

Docker was released by *dotCloud*, which was rebranded to *Docker Inc.* later, in March 2013. The Docker software is a set of products that use OS-level virtualization to deliver software in packages called containers.
Docker includes the following three components:

Docker Daemon or *dockerd*, which runs on the Docker host and is responsible for the centralised management of and the communication with the containers.

Docker Client or *docker*, which is a CLI that allows the user to communicate with the Docker daemon.

Docker API which is the centralized interface between the other two components. It takes all commands submitted via the Docker client and processes them using the Docker daemon.

Docker uses kernel namespaces and cgroups to isolate container instances from each other and from the host system.
The internals of Docker before and after version 1.11 are shown in figure 21. (See Liebel, 2017)

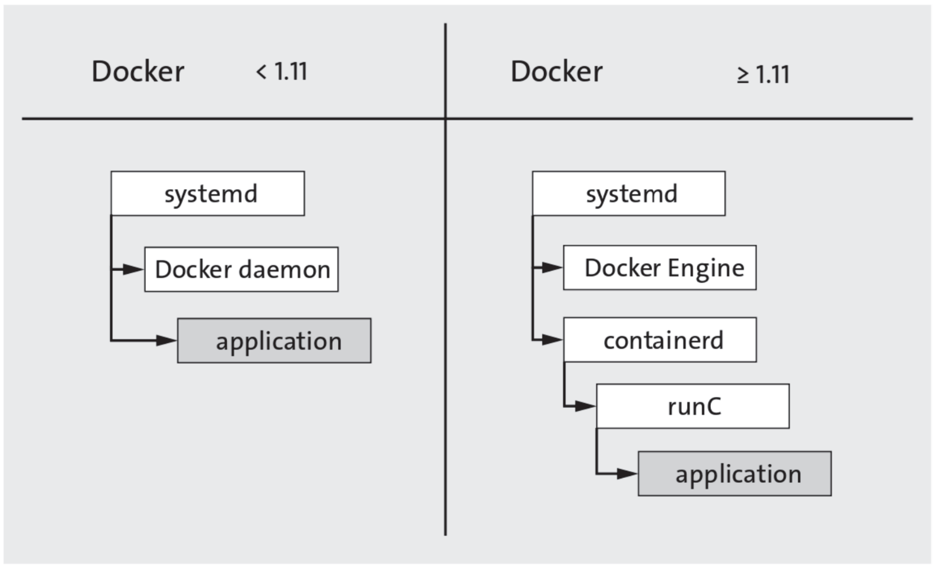


Figure 21: Docker software stack before and after version 1.11
(Liebel, 2017)

Docker containers will be created and configured using a *Dockerfile*. This file includes sequences of commands, that describe the structure of the container. An example of a Dockerfile using an interpreted language - in this case Python - is shown in appendix A.5.1. An example for a Dockerfile using a compiled language - in this case Golang - and a multi-stage build is shown in appendix A.5.2.

While the container of an interpreted language must always provide the interpreter, they tend to be bigger in size than images for compiled languages, since the compiled languages can utilize the multi-stage build system and run the binaries in minimalistic environments, e.g. *Alpine Linux*[8]. The current container image of Alpine Linux is only 5 Megabytes[9] in size, compared to Debians size of 114 Megabytes[10].

5.2.2 rkt

CoreOS - now *Container Linux* (See Red Hat, Inc., 2019b) - is a Linux operating system which targets the minification of the system by managing most parts of the OS itself by using containers. It initially used to be a dedicated platform for Docker containers but started developing an own container engine implementation.
This container engine is *rkt*, which was introduced in 2014. It's main focus points are security, reliability and scalability and it spawns container processes via the rkt-binaries, rather than running a central daemon. (See Liebel, 2017)
rkt provides a pod-native approach. That means it's core execution unit is a pod, which can include multiple applications or containers and acts synonymous to the Kubernetes pod concept. Additionally rkt enables the user to configure the execution environment, which eases the integration with other systems. (See Red Hat, Inc., 2019d)

5.2.3 cri-o

CRI-O is a lightweight, open source implementation of the Kubernetes *CRI* (Container Runtime Interface[11]), which aims to replace Docker or rkt as Kubernetes container engine and provide all the tools and software needed for that. It is developed by maintainers and contributors from companies like Red Hat, Intel, Suse, IBM and others. It enables using any OCI-compliant runtime as the container runtime for running Pods on Kubernetes. The currently supported container runtimes are runc and Kata Containers, but in principle any OCI-conformant runtime can be used. (See cri-o, 2019) The software consists of the following components:

[8]Alpine Linux is a very lightweight Linux distribution

[9]Last update: 21.10.2019 (See: `https://hub.docker.com/_/alpine`)

[10]Last update: 17.10.2019 (See: `https://hub.docker.com/_/debian`)

[11]Kubernetes CRI enables any container runtime to be connected to Kubernetes by using the CRI as a commuication middleware between Kubernetes and the runtime

OCI compatible runtime like for instance runc, Kata Containers or Clear Containers

containers/storage is a library that provides layer management and creation of root file systems for containers

containers/image is a library used for communication with container registries including pulling and pushing images

networking is described by the CNI and can be set up by using CNI plugins such as Flannel, Weave or Calico

monitoring is a utility that detects Out Of Memory situations and handles container monitoring and logging

Choice

The container engine of choice for the new system is Docker. This is mainly due to existing experiences and the ease of use of the software. This choice is explained in more detail in section 6.2.

5.3 Container-orchestration systems

A container-orchestration system is used to manage containers in scalable cluster environments. This includes the management of resources, scheduling, replicating and monitoring containers, health checking containers and cluster nodes as well as load balancing across container replication setups. With regard to figure 8 and 9 in this system the container-orchestration system takes the role of the scheduler inside the process management.

5.3.1 Docker Swarm

Docker *Swarm* (or simply *Swarm*) is a cluster management and orchestration platform embedded in the Docker Engine. It is built using a separated project called *swarmkit*. It is used to implement Docker's orchestration layer and is integrated into the Docker CLI.
A swarm is a cluster of at least one or more Docker hosts. The nodes of this cluster can act as managers, workers or both of these.
The central structure or main computation unit of a swarm is a *service*. It is used to define the tasks to run on the nodes. Services are described in a declarative way, which means that the configuration of a service describes the desired state and the software will determine the steps to reach that state.

Services can also be exposed or made available externally by using an ingress load balancing. (See Docker Inc, 2019)

5.3.2 Kubernetes

Kubernetes is a Container-orchestration system for automated application deployment, scaling and management across clusters of hosts. It is unopinionated in regards to the container engine or tools, like for instance Docker. It was originally designed and developed by Google. The first announcement was in 2014 and version 1.0 was released in July 2015. With this release Google partnered with the Linux Foundation to establish the Cloud Native Computing Foundation (CNCF).
The main features of Kubernetes are Scale-out possiblities, Load balancing, Resource request and limits, Self healing and Rolling updates. (See Liebel, 2017; The Kubernetes Authors, 2019c)
Some of it's commonly used API objects are as follows:

- Pods
- Deployments
- Services
- Volumes
- ConfigMaps
- Secrets

Choice

Kubernetes is the container-orchestration system of choice for the new system. This is due to its open source implementation, the set of features and the broad community behind the software. More details will follow in section 6.3.

5.4 Job and workflow systems

This section covers frameworks and tools for job or workflow based systems. In the scope of this system a job or workflow system will enable the user to configure chains of processes and define events that should trigger a job run, e.g. a time interval or arrival of new data. Additionally the job and workflow system will be responsible to commuicate with the container-orchestration system to spawn the modules respectively as shown in figure 9.

5.4.1 Apache Airflow

Apache Airflow is a Python framework used to implement workflows and handle those workflows scheduling. It consists out of five main components, which are as follows:

Database is used to monitor process definitions and to store metadata about the modules and their scheduling.

Scheduler is used to used to start modules based on their defined schedule.

Executor is called by the scheduler and used to execute the modules.

CLI is used for system management via the command line.

Web interface is used for system management and monitoring.

To improve scalability Airflow provides a variety of configuration options and supports third-party software to be used with the system.
By default SQLite will be used as the systems database. Since SQLite only supports one concurrent write at a time, concurrent writing operations can queue up and slow the system down. To scale up the system, the database can therefore be replaced. Due to the fact that Airflow was built using the SQLAlchemy Python library to interact with the database, every database compatible with SQLAlchemy can be used, even though MySQL or PostgreSQL are the recommended RDBMS.
Regarding the executor, the system uses a sequential approach by default. This executor can be replaced with the *LocalExecutor* for local concurrent execution. To support horizontal scaling Airflow also provides a *CeleryExecutor*, which uses Celery to distribute workloads, and a *KubernetesExecutor*, which uses Kubernetes to spawn and process workloads on distributed systems.
To describe workflows, Airflow uses the concept of DAGs, which are implemented and described using Python. The processing steps of each workflow are built upon *Operators*, which are provided by Airflow, but can also be implemented individually. (See The Apache Software Foundation, 2019; Stoffers et al., 2019; SQLite Consortium, 2019)

5.4.2 Argo Project

The Argo Project contains multiple components that are specifically developed for the use with Kubernetes. The goal of the project is to provide software that can be used to implement automated container-based systems. The currently available components are:

- Workflows and Pipelines
- Continuous Delivery
- Rollouts
- Events

While the Continuous Delivery and Rollouts components are mainly used for Git Operations and the management of Kubernetes Deployments update strategies, the Workflows and Pipelines and Events components can be used to implement event based processing systems.
The Workflows and Pipelines component provides a container native workflow engine for Kubernetes which supports both DAG and step based workflows. It is implemented using Kubernetes Custom Resource Definition (CRD). Therefore setting up Argo on a Kubernetes cluster includes deploying a controller, which monitors and manages those custom resources, and configuring Rule-based Access Control (RBAC) permissions.
Argo Workflows and Pipelines manages process scheduling and retries, handles input and output of the modules and provides mechanisms to implement conditionals, which enable the definition of conditions when a job should be skipped, loops or exit handlers, which will run, even when a job failed and can be used for cleanup or notification modules.
By default those workflows will be deployed and triggered using the Argo CLI. (See Argo Project, 2019) Using the Argo Events component those workflows can be triggered by events of different types, which include the following:

- S3[12]
- File
- Streams
- Webhooks[13]
- Schedule or Calendar
- Kubernetes Resources

Choice

Due to it's interplay with Kubernetes and the rather simple configuration and setup process, Argo is the choice of workflow tool for this system. This will be explained in more detail in 6.4.1 and 6.4.2.

5.5 File storage

This section focusses on possible solutions for the file storage of the system. As shown in figure 10 the file storage will be part of the data management. It will be responsible for storing all kinds of data or files needed for processing.

[12]Amazon S3 compatible events
[13]Typically HTTP POST request to defined URL endpoints

5.5.1 Minio

MinIO is an object storage system, which is compatible with the Amazon S3 cloud storage service's API. It is implemented in the Go programming language and most suitable for storing unstructured data, e.g. photos, videos, log files, etc. File sizes of up to 5 Terabytes are possible. It also provides many features that Amazon S3 also offers, including Bucket[14] Lifecycle, which can be used for automatic garbage collection, and S3 events, which will be triggered by create, update or delete data transactions.
Minio is made of two components:

- Minio Server
- Minio Client

The Minio Server is main server software running on a dedicated host and provides the main functionalities for storage and communication. Furthermore it can be used to create a distributed storage system by setting up a Minio Cluster.
The Minio Client is a CLI used for communication with the Minio Server. It provides commands for server and cluster configuration, data replication and data transactions and can communicate with remote servers via HTTP. Additionally Minio provides SDKs[15] including libraries for Java, Go, Python, etc. (See MinIO, Inc., 2019; Wernicke, 2017)

5.5.2 GlusterFS

GlusterFS is a free and open source, scalable, distributed network filesystem most suitable for data-intensive tasks like media streaming, data analysis or cloud storage. It can scale to serveral Petabyes and can handle thousands of clients, which - in the scope of GlusterFS - are devices or machines that mount a GlusterFS volume. Furthermore it is POSIX[16] compatible and can use any on-disk filesystem that supports extended attributes. (See Gluster, 2019; Wernicke, 2017)
In GlusterFS a *volume* represents a mounting point and is a collection of *bricks*. A brick is the basic unit of storage in GlusterFS and is built on an export directory on a server. GlusterFS supports different kinds of volumes, which are comparable to the different types of RAIDs[17].

[14]Amazon S3 unit of organization comparible to directories

[15]Software Development Kits

[16]„The Portable Operating System Interface refers to a family of related standards“ specified by the IEEE Computer Society (The Open Group, 2017)

[17](Redundant Array of Inexpensive Disks) storage virtualization technology used to combine multiple physical disks into one or more logical units

5.5.3 Apache Hadoop / HDFS

Apache Hadoop is a project of the Apache Software Foundation whose goal is the development of open source software for reliable, scalable and distributed computing. The project includes a number of modules, e.g. Hadoop YARN (job scheduling and cluster resource management), Hadoop Ozone (Object storage), Hadoop Submarine (machine learning engine) and the Hadoop Distributed File System (HDFS).
HDFS is a distributed file system designed to handle very large files and provide high-throughput access to application data. It's main goals are detection of hardware failures and quick, automatic recovery, handling files that are typically Gigabytes to Terabytes in size and providing streaming access to those files.
The HDFS can be accessed via the provided FileSystem Java API, a C language wrapper for this Java API, a REST API or by using a NFS gateway. (See Apache Software Foundation, 2019)

Choice

The most basic choice for this part of the system could be each processing nodes filesystem or network storage, e.g. NFS shares. Since the goal is to avoid single points of failure a more sophisticated solution or tool is preferred. Thus Minio is chosen as the systems file storage. Additionally to its replication and sharding capabilities, Minio has a high degree of compatibility to Argo Workflows and Argo Events as artifact storage and event emitter. This will be described in more detail in section 6.5.1.

5.6 Database

This section covers database management systems that can be used to implement a metadata database for the system. As seen in figure 10 is part of the system's data management and will be responsible for storing metadata about the files that are present in the system.

5.6.1 PostgreSQL

PostgreSQL is an open source RDBMS. It additionally provides a class concept, which is based on unique object ids that will be used to unambiguously represent instances of a class. A class can be a table. The rows of this table are the instances or the objects of this class. Inheritance is provided by allowing the user to derive child tables. Those characteristics make PostgreSQL an object-relational database management system.

PostgreSQL can be clustered or replicated using a master-slave architecture - namely *Streaming Replication*[18], which is available since version 9.0 - or by using third party software, e.g. *pgpool* as a synchronized replication server. This pooling service acts as a middleware between the application and the database and can send SQL queries or transactions to multiple database instances at the same time. (See Weinstabl, 2004)
PostgreSQL natively supports sharding since version 11.0. (See Haas, 2018)

5.6.2 Redis

Redis is a key-value NoSQL database. Its main focus point is performance, which is often limited by the hosts network connection rather than its hardware. The data gets stored inside the memory, which leads to the fact that the amount of data Redis can store is limited. Despite this the data can be persisted using snapshots in regular intervals or a log file. It supports five different data types, atomic operations and provides commands to enable the use of transactions.
Since Redis does not provide support for JOIN-operations, it is particularly suitable for data that is not in relation to other data.
It also supports to setup a master-slave architecture to enable horizontal scaling, but those clusters are not strictly consistent, but only *eventually consistent*, which is a consistency model in distributed database environments describing the lack of data consistency that can occure while the database nodes are synchronizing.
Sharding is currently not supported, but has to be handled on application level. (See Hollosi, 2012)

5.6.3 MongoDB

MongoDB is a document-based NoSQL database. It groups documents in collections, which are similiar to tables in relational databases. The data inside those collections, which will be stored in BSON[19] documents, is schemaless. That means that the documents can contain arbitrary data. MongoDB limits the size of the BSON documents for efficiency reasons to 16 megabytes.
MongoDB provides the possibility to arrange servers in Replication Sets to guarantee availability in the event of a server failure. Replication Sets are also crucial for data security, since they mirror the data on multiple servers to avoid data loss in case of a server failure.
The Replication Sets are based on a master-slave architecture. The special

[18]See `https://wiki.postgresql.org/wiki/Streaming_Replication`
[19]BSON is a binary data format based on JSON (See `http://bsonspec.org`)

feature is that the master is not fixed, but will be selected by the servers involved. The master is called *Primary*, the slaves are called *Secodary*. Writing operations are exclusively handled by the *Primary*, while reading operations can also be delegated to the *Secondaries*.
The servers of a Replication Set monitor each other. In case the *Primary* fails, a *Secondary* will be voted and promoted to the new *Primary*. In most cases this will be the *Secondary* with the most recent data state. To be promoted the *Secondary* needs a majority of the votes. This majority rule is intended to prevent the existence of two *Primaries*. Therefore a Replication Set should always consist out of an odd number of servers.
MongoDB also supports sharding based on a sharding-key to avoid single points of failure. (See Hollosi, 2012)

Choice

Due to existing experience, the broad community behind it and the possibility to query data via SQL, PostgreSQL is the database management system of choice. Additionally it offers a simple solution for the dynamic metadata field, which will be discussed in more detail in section 6.5.2.

6 Implementation

This section covers the implementation details of the system.

6.1 General

This sections gives general informations and an overview over the system.

6.1.1 Hardware

The prototype system was implemented on four hosts, each with 8 CPU cores, 16 GB of memory and 256 GB of storage, provided by a *VMware vSphere* cluster.
The hosts run a *CentOS 7* operating system.

6.1.2 Software

This section gives an overview of the software used in the system and it's purpose.

Docker is the Container engine of the system. It ensures the availability of container images, through building or downloading from a registry, allows to run container and provides the logs of those.

Kubernetes is the container-orchestration system of the processing system. It spans a virtual cluster and network, provided by Calico, and allows to run or deploy containers on this cluster.

Argo Workflows and Pipelines is the workflow engine of the system, which takes in YAML configurations and parses those to workflow descriptions. Argo is used to trigger job runs in the order defined by the configuration, handle in- and output of as well as provide the communication between the modules.

Argo Events is an event-based dependency manager, which is used to trigger and parameterize Argo Workflows based on events, e.g. incoming data or calendar scheduling.

Minio is used as the systems storage. All files are stored as objects and are made accessible by using Minios HTTP interface.

PostgreSQL is used as the database to store the metadata of all files in the system, enabling the users to query for files that are interdependent.

Figure 22 shows the systems architecture in respect to figure 8 and the components interplay with the concrete software used for implementation to unterstand the role each software takes in the system.

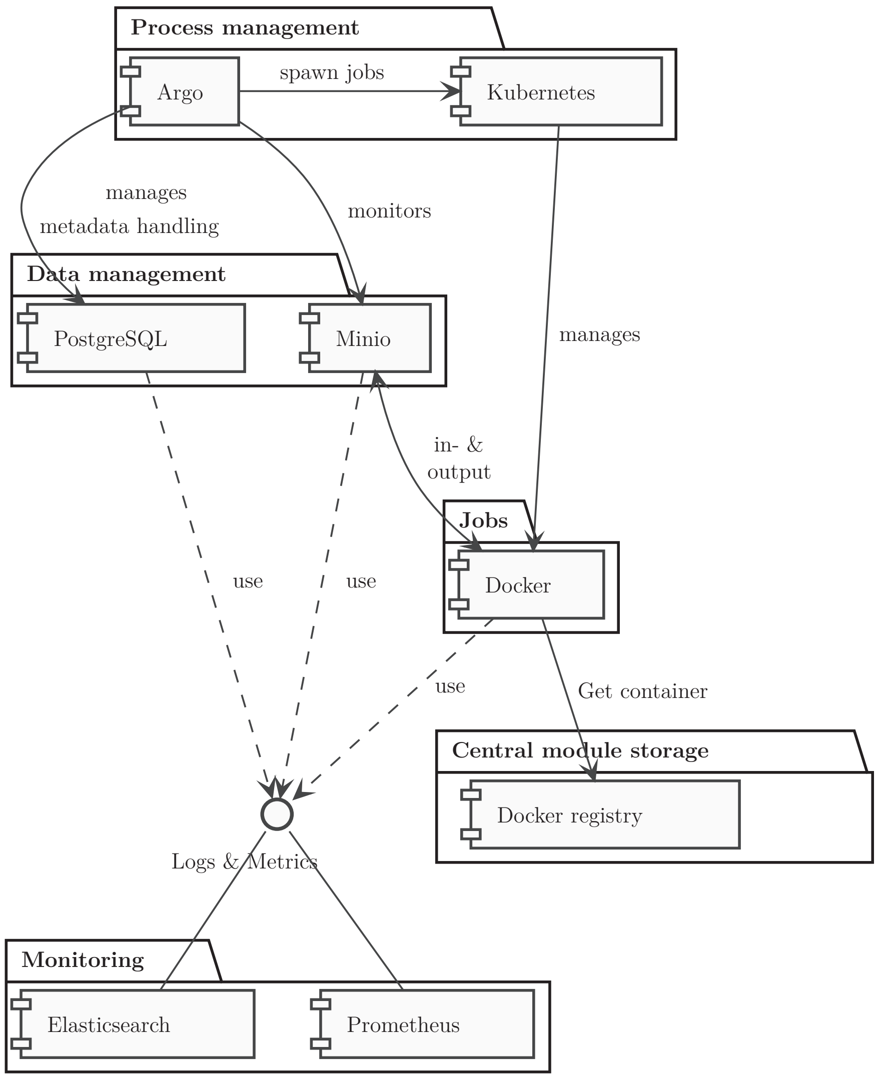


Figure 22: Software and components interplay

6.2 Container Engine

The container engine on the prototype system is Docker. This software is used due to already existing experience and the easy of use of the software. Additionally Docker currently is the default container runtime of Kubernetes. Therefore no additional configurations regarding the runtime have to be made

to set up the system.
Since the hosts run CentOS 7, the up-to-date Docker packages are not in the default repositories. Therefore the Docker CE repository is added before the software will be installed. The version used in the system is *19.03.04*.
The daemon is configured to use *cgroupfs* as its *cgroupdriver*, instead of the *systemd* driver recommended by Kubernetes. This is due to some problems that occured during the development process while using *systemd*.
The daemons configuration file can be seen in appendix A.3.

6.3 Container-orchestration system

Kubernetes is the container-orchestration system used for implementing the system. This is due to its open source implementation, the set of features and the broad community behind the software. The version used in the system is *15.3*.
The Kubernetes cluster was initialized using the kubeadm CLI.

Container Network Interface (CNI)

The Container Network Interface (CNI) used in the cluster is Calico, which was deployed using the default manifests provided by the Calico Project website.

Package Manager

Helm is a package manager for Kubernetes which enables developers to write complex manifest templates, which are called *Helm Charts* and are configured using the corresponding configuration file. It includes the server-sided service *Tiller* and the client-side *Helm* CLI. (See Helm, 2019)
Currently there are the *ingress-nginx* Ingress controller (See The Kubernetes Authors, 2019a) and the *MetalLB* load balancer (See Anderson, 2019) deployed which provide external access to the services running inside the system. Those Charts are deployed using the stable Helm repository.

6.4 Process management

This section covers the process management of the system which is implemented using the components of the Argo Project.

6.4.1 Argo Workflows and Pipelines

The Argo Workflows and Pipelines module is used to describe the workflows using YAML. It handles the processing steps or DAGs as well as input and output handling and communication between modules.
Argo is deployed using the Argo Workflows Helm chart. Therefore the Argo repository has to be added to helm by executing `helm repo add argo https://argoproj.github.io/argo-helm`.
The structure of a workflow definition can be seen in listing 6.

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  # METADATA #
spec:
  # WORKFLOW PARAMETERS #
  entrypoint:
  templates:
    # TASK / WORKFLOW TEMPLATES #
```

Listing 6: Argo Workflow definition structure

The *apiVersion* references the Argo API, which is followed by the *kind* option, which in this case is a workflow. Those options are mandatory for every Kubernetes manifest.
The *metadata* section contains metadata about the workflow, e.g. the namespace where it has to be deployed to or the name prefix for the pods spawned for that workflow.
The *spec* section contains the configuration of the workflow itself. It can contain parameters like the maximum duration a workflow can take before being shut down (*activeDeadlineSeconds*) or the duration exited pods have to be persisted for before being garbage collected (*ttlSecondsAfterFinished*). The most important options in this section are the *templates*, which define the tasks of the job itself, and the *entrypoint*, which defines the starting point of the workflow. Additionally there is an option for an *onExit* handler, which defines a task or workflow that will be executed, regardless of the main workflows result.
The *templates* section of the workflow definition contains a list of templates. Those templates will always contain a name and a description, what should be executed in this template. There are multiple options for that descrip-

tion[20], but in this case the most commonly used ones are:

container specifies a template that will execute a single container. (See appendix A.6.1)

script specifies a template which works similiar to the container template, but additionally offers the possiblity to access the standard output of the container. (See appendix A.6.2)

steps specifies a template that will execute a list of templates referenced by name in the given order. Those steps can be run sequentially or in parallel. (See appendix A.6.3)

dag specifies a template that executes templates ordered by using a Directed Acyclic Graph. (See appendix A.6.4)

6.4.2 Argo Events

The Argo Events module is used to react to events like incoming data, calendar events, etc. It provides logical connections between those event sources and grants the user to let the workflows respond respectively.
This module is deployed using the Argo Events Helm chart, which needs the Argo repository to be enabled as well.
To register a workflow that reacts on certain events, four components have to be defined:

Event source is used to describe the event to listen to. This can for example be a calendar event, which is similiar to a cronjob, or a S3 bucket, which throws events in case of data transactions. (See appendix A.7.1)

Gateway consumes the events specified in the event source, converts them into *cloudevents specification*[21] compliant events and sends them to the corresponding sensor. (See appendix A.7.2)

Sensor receives the events from the gateway and triggers jobs based on defined logical connections between the event dependencies. (See appendix A.7.3)

Workflow describes the tasks that have to be executed or triggered. Since those workflows are managed by the Argo Workflows and Pipelines

[20]See `https://github.com/argoproj/argo/blob/master/pkg/apis/workflow/v1alpha1/workflow_types.go#L241`

[21]See `https://github.com/cloudevents/spec`

module the configuration process is the same. Because these workflows must not be triggered immediatly when deployed to the cluster, they can for instance be wrapped inside a ConfigMap to be persisted and made available cluster-wide. (See appendix A.7.4)

6.5 Data management

This section focusses on the implementation details of the data management system including the metadata database and the file storage.

6.5.1 File storage

Minio used for the file storage. The reasons for this are the support by Argo and it's simplicity, including the HTTP interface which handles data or file transfers and takes Kubernetes volume configurations off the user.
This way event listeners or sensors for specified prefixes and suffixes can be registred on certain buckets which grants the user a great amount of control about what kind of files will trigger each job.
Argo also uses Minio as an artifact storage. This means specified output files of the modules will be uploaded to the artifact bucket automatically. Afterwards they are processed by the data management job and moved to their intended destination.
Furthermore Minio supports a bucket lifecycle so that an automatic garbage collection of files can be configured. For example a bucket can have a lifecycle of a certain timespan which will cause Minio to delete all files older than this regularly.
In the current setup Minio is running on a dedicated host outside of the Kubernetes cluster.
Every product type has it's own bucket. That means that for example all files or products related to the scintillation will be stored inside the scintillation bucket.

6.5.2 Metadata database and datamodel

The metadata database is a PostgreSQL instance. This enables the users to use the established SQL standard when defining the dependencies between files.
PostgreSQL is deployed using the Helm chart of the stable repository.
Just as with the file storage, every product type has it's own namespace which in case of the database is represented by a table. All tables for the products have the same structure, which includes a unique id for every row,

a timestamp, which represents the product time and enables the module to determine the expiration of the data, the files location, including the Minio bucket and object key, and the metadata extracted by the corresponding decoder. The metadata will be stored in JSON format and can therefore contain arbitrary data.
Listing 7 shows how JSON data can be queried using SQL.

```
SELECT
    metadata ->> 'expiration' as expiration
FROM
    product
ORDER BY
    expiration;
```

Listing 7: Example SQL JSON query

The database communication is provided by the metadata module or CLI. It is implemented using Python and the Python library SQLAlchemy. This library provides an Object Relational Mapper, which means that Python classes and objects will be mapped to SQL tables and tuples. It includes functions to manipulate and query the data, which internally get transformed into SQL queries.
Currently the CLI contains three subcommands to add, delete and query for data. It implements the data model shown in figure 13 by using a Python class with the described attributes.
New data is added using the add command. The program takes in a JSON file containing the product name, the files location, the product time and the additional metadata. It then checks if the table corresponding to the product type already exists. If it does not it will be created automatically by deriving from the defined class (see section 4.5) and adding the product or table name. Finally the data gets commited to the database.
Deletion of database entries is done by calling the delete command. This command is intended to be used by a garbage collection job that keeps the metadata database in sync with the data basis or the filesystem. It's input arguments are a table or product name and a file key. The program then checks if the specified table exists. If it does the entry containing the given file path will be deleted.
Data can be queried by using the query command. It therefore takes an SQL query, which will then be forwarded to the database.
Listing 8 shows an example for a more complex query.

```sql
SELECT
    pre.bucket,
    pre.path,
    chb.bucket,
    chb.path
FROM
    pretopo as pre,
    chbias as chb
WHERE
    (CASE age(pre.product_time, chb.product_time) < INTERVAL
    ↪  '0' THEN -age(pre.product_time, chb.product_time) ELSE
    ↪  age(pre.product_time, chb.product_time)) < INTERVAL '20
    ↪  minutes'
AND
    age(now(), pre.product_time) < INTERVAL '1 day';
```

Listing 8: Example SQL query

Using the query module the result will be converted into a list. The following steps can depend on the output. So for example a module that needs to be run for every match gets spawned an n-amount of times, where n equals the number of found matches. Those modules will then be executed in parallel. This process is shown in figure 23.

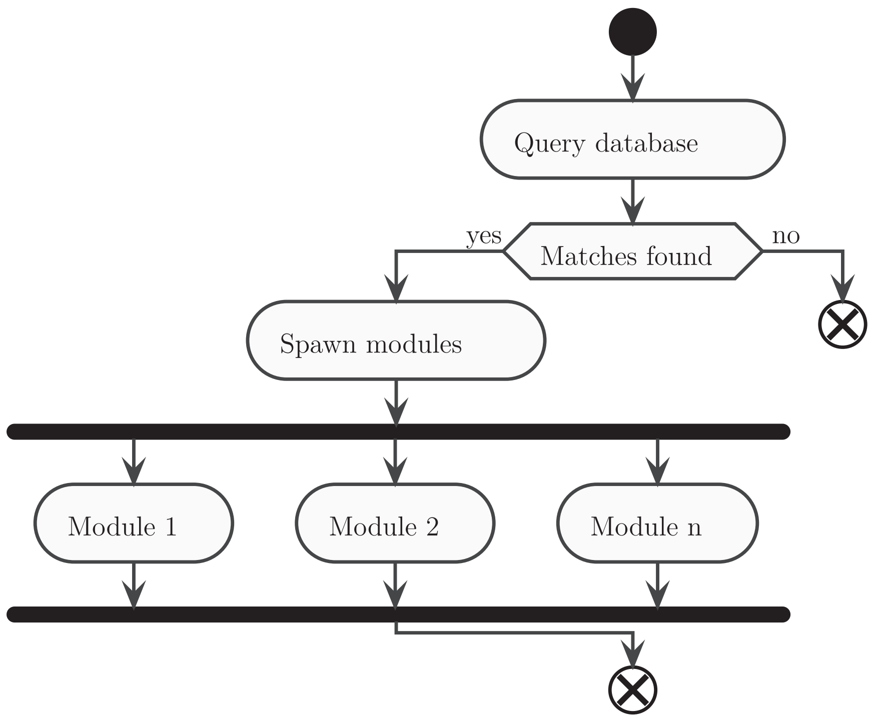


Figure 23: Sequence of a module which uses data matching

6.6 Logging

The logging solution is implemented using the Elastic Stack (ELK)[22]. This includes Elasticsearch[23] and Kibana[24]. This provides the use of a centralized log storage combined with a search engine, utilized and made accessible by a web frontend. This enables the user to query for certain data or logs using a graphical user interface, storing those queries and creating dashboards, which improve usability and transparency.
To deliver the logs from the hosts to the Elasticsearch instance, there has to be a log forwarder, which in this case is Fluent-Bit[25] due to it's lightweight architecture and the focus on Docker and Kubernetes environments.
Those products are deployed to the cluster and configured using the *elasticsearch* and *fluent-bit* Helm charts out of the stable repository.

[22]Open source product stack developed by Elasticsearch B.V.
[23]Distributed, JSON-based search- and analytics-engine (See Elasticsearch B.V., 2019)
[24]Elasticsearch web frontend used for data visualization (See Elasticsearch B.V., 2019)
[25]Open source log forwarder aimed towards the use with Docker and Kubernetes (See Treasure Data, 2019)

6.7 Metrics

The metrics solution is implemented using Prometheus[26] due to its wide adoption and ease of use. It is deployed and configured using the *prometheus* stable Helm chart. This provides a centralized metrics solution, that can be set up and configured using a Kubernetes CRD. The default web frontend for the Prometheus deployment is Grafana. This enables the exploration of metrics of the whole system or single components of the system.

6.8 Infrastructure

This section covers general infrastructure elements of the final system.

6.8.1 CI/CD

For Continuous Integration / Continuous Delivery (CI/CD) or deployments of most resources to the system the GitLab CI is used. Therefore the user does not need to have any knowledge about the communication with the system but can use a preconfigured automatic deployment system instead.
Also by controlling the deployments to the system with the CI system the corresponding repository will always be a clear indicator for the status and the configuration of the deployment.
The cluster is configured in a GitLab group so every repository inside this group has access to it. Clusters can also be configured on repository level which will be prioritized over group clusters.
The CI's pipelines are triggered by commits or pushes to the repositories. Based on the branch targeted by the commit, different actions can be executed, e.g. commits to the master branch can trigger a production pipeline and commits to every branch except the master can trigger a testing pipeline. Since Kubernetes uses namespaces to create isolated environments, each job should have it's own namespace, which helps to keep the monitoring of the system more simple. GitLab uses the concept of namespaces by default so there is no additional setup needed. The namespace of each job or repository is generated out of it's repositories name, ID and environment[27] name. For example a pipeline in a repository with the title *example-job*, the ID *45* and the environment *production* will deploy all manifests to the namespace *example-job-45-production*.

[26]Open source monitoring and time series database system (See Prometheus Authors, 2019)

[27]GitLab Environments provide a way to track deployments. At least one environment must be available to deploy manifests to Kubernetes.

Appendix A.4 shows an example GitLab CI configuration, which deploys all Kubernetes manifests inside the repository to a Kubernetes cluster configured in GitLab. The file has to be placed inside the repository for GitLab to run the defined pipelines. To ease the process of creating a new job for the user, repository templates will be created, which provide a basic directory structure, licenses, documentation and GitLab CI files.

6.8.2 System environments

The three different environments of the system, which are described in section 4.2, will be realised by setting up multiple instances of the system. Therefore each environment will have it's dedicated hosts. Those can easily be initialized using the Ansible playbook.
The deployments to the environments will be handled by the GitLab CI. For example pushes to an arbitrary branch, except the master branch, of the repository will deploy the manifests to the testing environment. Pushes to the master branch will trigger a deployment to the operational processing environment and remove the deployed resources from the testing environment.
To trigger a postprocessing workflow there are multiple possible solutions.

Argo CLI The currently used solution would be to submit the postprocessing workflows via the Argo CLI. This way no additional server-side setup would be needed but the communication with the cluster has to be configured on the users device.

Argo Events It would also be possible to register Argo Events resources on postprocessing S3 buckets, which start workflows as soon as the user stores configurations of the workflow inside the bucket. Since Argo Events also provides support for Slack[28], the triggering and parametrization of postprocessing workflows via text messages would be possible.

6.8.3 Image registry

An image registry allows the storage and distribution of container images. Dockers default registry is the Docker Hub (See `https://hub.docker.com`), which is based on the Docker registry. This registry can also be run locally by using the *registry* container image provided by Docker. (See `https://hub.docker.com/_/registry`) This way the images can be stored locally and therefore kept internally and private.
In this system a Docker Registry is setup outside of the Kubernetes cluster.

[28]Slack is a cloud-based proprietary instant messaging platform developed by Slack Technologies (Wikipedia, 2019b)

6.8.4 Utilities

To keep the process of configuration easy for the user, a utilities container is provided.
This container contains modules which will make the communication with the systems resources easier. It is also used in the management jobs.
The container includes the following modules and functions:

metadata The *metadata* module contains functions to communicate with the database.

- add
- query
- delete

mc The *mc* module contains functions to communicate with the Minio server.

- find

mail The *mail* module allows to send mail notifications, e.g. if a job failed and needs investigation.

key-decode The *key-decode* module is used to decode the S3 keys received via Argo Events.

create-prefix The *create-prefix* module is used to generate a file name prefix based on the current date, time and a given time format. This tool is used in management jobs to guarantee the naming conventions.

6.9 Setup

The initial setup of the system was realized using an Ansible *Playbook*. A Playbook is a collection of tasks (called *Plays*) Ansible should execute on a specific pool of hosts to orchestrate, configure, administer or deploy a system. (See Red Hat, Inc., 2019a)
An example of a plays definition is shown in listing 9. This simple play instructs Ansible to use the *yum* package manager to install the packages *vim* and *htop*.

```
- name: Install extras
  yum:
    name: "{{ packages }}"
    state: present
  vars:
    packages:
      - vim
      - htop
```

Listing 9: Ansible Play example

Those Plays can be grouped by *Roles*. In Ansible roles can be used to form groups out of lists of plays. Assigning a role to a group of hosts (or a set of groups, or host patterns, etc.) implies that they should implement a specific behavior. Roles can include certain variable values or defaults. The file structure of Ansible roles make them redistributable, reusable components allowing to share logic or behaviour among playbooks. (See Red Hat, Inc., 2019a)
The implemented playbook includes the following roles:

common The role *common* adds the EPEL[29] repository and installs common packages.

selinux The role *selinux* disables the SELinux[30] module since it is not fully supported by Docker and Kubernetes yet.

docker The role *docker* installs Dockers prerequisites, adds the Docker Community Edition repository, installs Docker CE and docker-compose[31] and configures, reloads and enables the Docker daemon.

iptables The role *iptables* enables the *net.bridge.bridge-nf-call-ip6tables* and *net.bridge.bridge-nf-call-iptables* to ensure correct networking capabilities for Kubernetes.

swap The role *swap* disables all swap partitions of the hosts since they are not supported by Kubernetes.

[29]„EPEL (Extra Packages for Enterprise Linux) is a Fedora Special Interest Group that creates, maintains, and manages a high quality set of additional packages for Enterprise Linux, including, but not limited to, Red Hat Enterprise Linux (RHEL), CentOS and Scientific Linux (SL), Oracle Linux (OL)“ (Red Hat, Inc., 2019c)

[30]Security-Enhanced Linux

[31]Tool for running applications containing multiple container

firewalld The role *firewalld* configures all ports required for Kubernetes' communication.

kubernetes The role *kubernetes* adds the Kubernetes repository, installs kubeadm and kubeletand starts and enables the kubelet service.

master The role *master* is limited to the master group. It installs kubectl, creates a user, initializes the Kubernetes cluster while storing the join command needed by the workers and installs Calico inside the cluster.

workers The role *workers* use the join command, stored while executing the *master* role, to integrate the worker hosts into the cluster.

In order to run the playbook Ansible has to be installed. This can be done by executing the command `pip install ansible`.
The command `ansible-playbook <PATH/TO/PLAYBOOK/` rolls out the playbook to the specified hosts.
The hosts are defined in an *Inventory* file. By default Ansible looks for this file under `/etc/ansible/hosts`. Listing 10 shows an example for an Inventory file which specifies the hosts by their respective hostnames.

```
all:
  hosts:
  children:
    master:
      hosts: k8s-master-01
    workers:
      hosts: k8s-worker-0[1:3]
```

Listing 10: Example Ansible Inventory file

6.10 Management jobs

This section covers some of the implementation details of the management jobs.

6.10.1 Data handling

Incoming data can originate from external or internal sources. The management job which handles incoming data is implemented using an event listener from Argo Events. It listens on the artifact bucket on the Minio storage

server. The incoming events are then filtered by prefix and suffix. This way the configuration to use for the data handling job will be determinded. If there is no configuration which matches the incoming events prefix or suffix, no job will be spawned.
This job takes five input parameters:

- S3 bucket
- S3 key
- Destination bucket
- Decoder
- File name

The S3 bucket and key will be extracted from the incoming event. The destination bucket specifies the location, where the file will be transfered to.
The decoder specifies, whether the incoming file should be decoded and metadata should be stored and which decoder should be used. After the database commit with the new metadata finished, an empty file will be created on the local storage which symbolizes the presence of metadata for this file in the metadata database. This file extends the name of the incoming file by a *.meta* extension.
Since all artifacts are stored as archived files and are automatically extracted by Argo in the download process, the incoming data can either be a file or a directory. The file handler respects that, but if the incoming data is a single file the original file name gets lost. Therefore a file name can be specified in the job configuration to allow for a homogeneous naming convention system.
The sequence of the workflow is shown in figure 24.

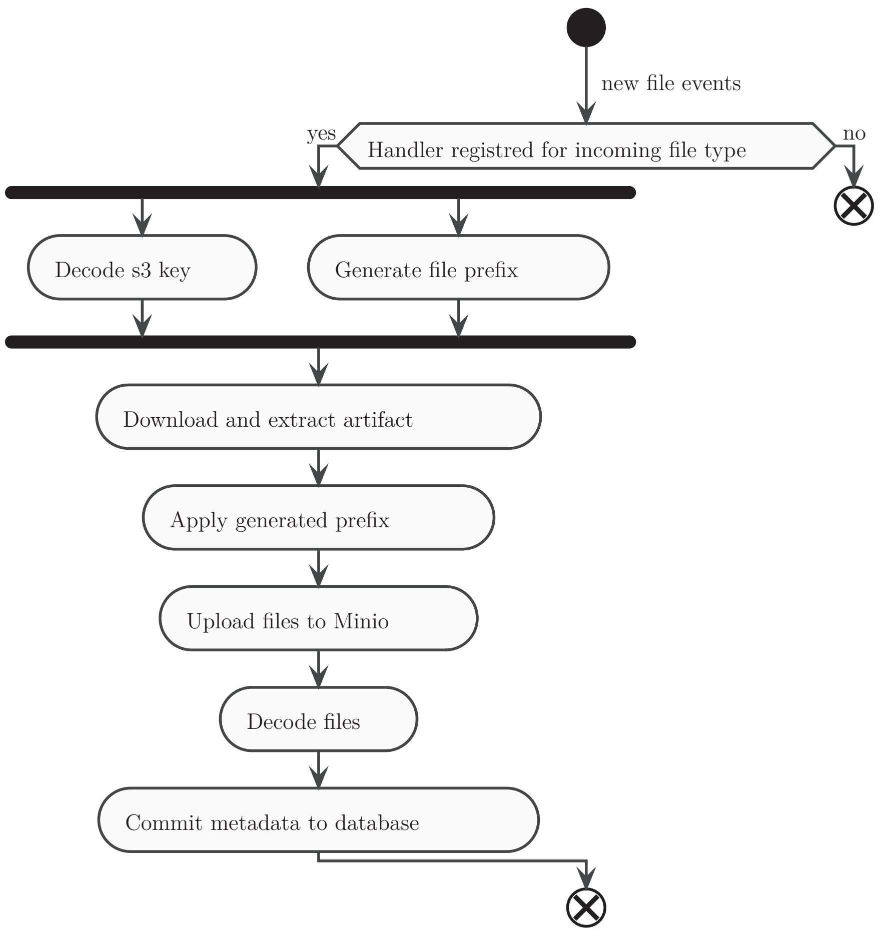


Figure 24: Sequence of the file handling job

The job checks all available configurations. If there are any matching prefixes and suffixes, the job will be spawned with the matched configuration.
Since the S3 object key delivered by the event is a URL encoded string[32] (e.g. `npsm%2F2019%2F294%2F08%2F20191021_0800_negrd.gif`) the key has to be decoded.
The decoding happens concurrently to the generation of the file's prefix. The prefix follows the naming convention mentioned in section 4.6.1.
After both modules finished the file will be downloaded and extracted. The next module then prepares the data for the local storage by renaming and applying the generated prefix to it.
The files will then be uploaded to the local storage and decoded. The metadata will then be commited to the database.

[32]URI encoding is a an encoding mechanism, that enables the use of „unsafe“ characters in URLs by using a percent (%) sign as an escape character (Berners-Lee, 1994)

If this job fails it will additionally be retried until a maximum of 2 times.

6.10.2 Deleted data

The management job, which handles deleted data or the garbage collection, is also implemented using an Argo event listener. It watches for the deletion of files in certain buckets. Based on that it spawns a module which takes the S3 key of the deleted file, decodes it and queries the database for the corresponding metadata entry which is then deleted.
The sequence of the job is shown in figure 25.

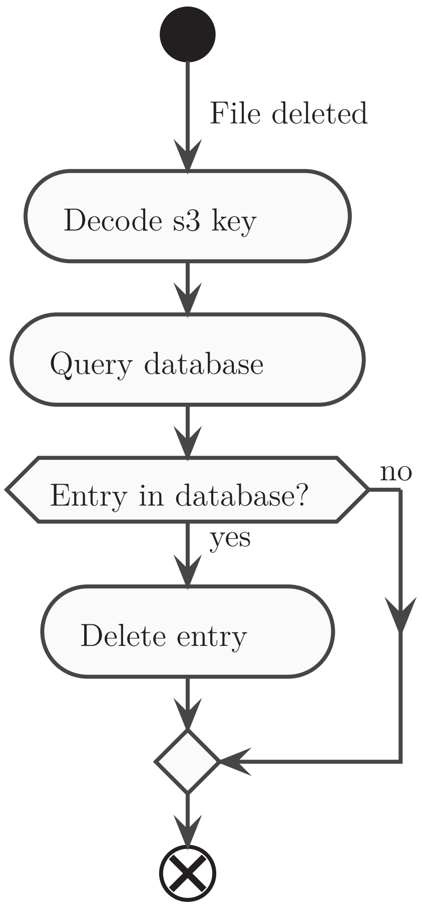


Figure 25: Sequence of the garbage collection job

6.11 Example user-defined jobs

This section shows some examples of user-defined jobs and modules that are currently running in the system.

6.11.1 Scintillation synchronization

The scintillation synchronization job (short: *sync-sct*) is used to synchronize the local storage with an external data source. It is meant to run in certain time intervals and synchronize data from different stations. Therefore the stations are defined in a ConfigMap.
In the first step all files of this ConfigMap get mounted to a module which than generates a list of the stations.
The subsequent step takes the output of the first step and spawns a sync-sct module for every item or every station in the generated list, which will then run in parallel. Those modules will then authenticate and communicate with the corresponding FTP server and download new data. Afterwards the new files will be uploaded to the local storage.
This workflow is visualized in figure 26.

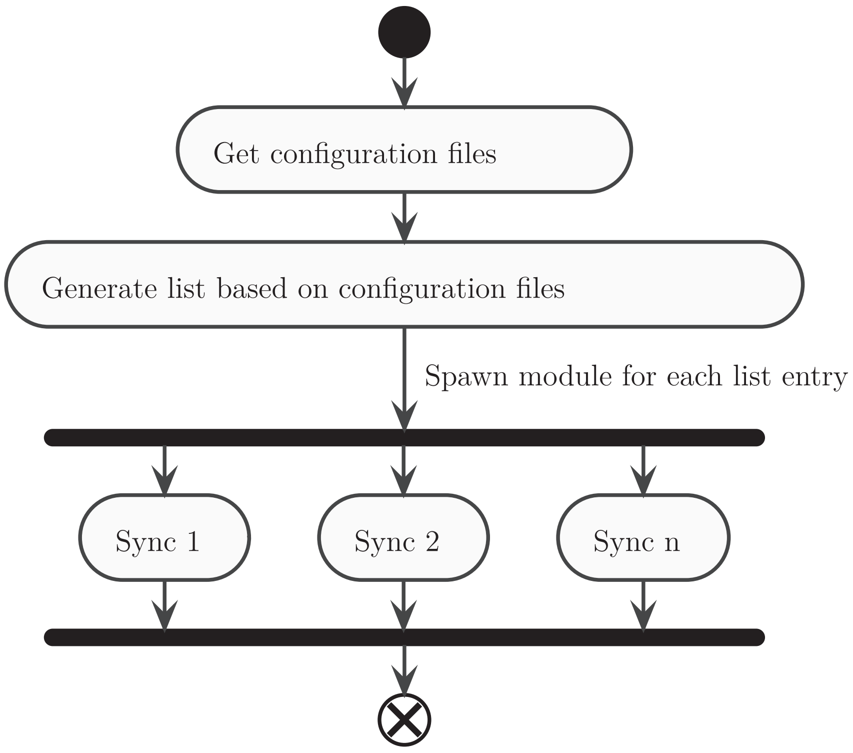

Get configuration files
Generate list based on configuration files
Spawn module for each list entry
Sync 1
Sync 2
Sync n

Figure 26: Sequence of scintillation synchronization job

6.11.2 Scintillation plot

The scintillation plot job (short: *plot-sct*) is used to generate plots visualizing scintillation events. It is meant to be run when a scintillation file gets created

or updated in the local storage.
The job is started when an event for a new scintillation file is received. It then extracts the file location out of the event and downloads the file. Afterwards the plot-sct module starts and generates the plots. Therefore a ConfigMap, containing and describing all products that shall be produced, is mounted. The plots are then stored as artifacts and will be uploaded to the corresponding bucket in the last step.
The configuration of this workflow can be seen in appendix A.7. There is also an example postprocessing workflow definition for this module shown in appendix A.8.
An output example of this module using the data from the Bahir Dar station is shown in figure 27.

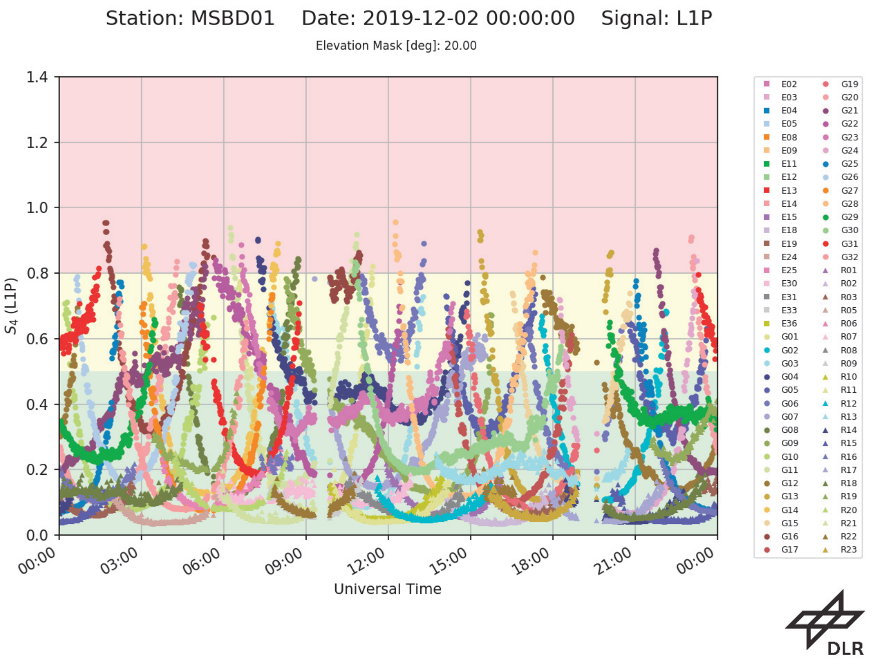


Figure 27: Output example of the scintillation plot module

6.11.3 Neustrelitz plasmasphere model

The Neustrelitz plasmasphere model job (short: *npsm*) is used to generate plots visualizing the total electron content distribution of the plasmasphere. It is meant to run in certain time intervals and depends on the F10.7, which is provided as an external file by NASA OMNIWeb Service.
In the first step the file containing the F10.7 is downloaded. This happens every time the process runs since new data is expanded to the file.

The second step runs the model which depends on the time, date and the F10.7. The time and date default to the current time. This includes the year, day of year and the hour. The F10.7 gets extractet from the fluxtable which was provided in step 1. Therefore the file gets read, outliers get removed and the value nearest to the input time gets picked.

6.11.4 CHAMP Processors

To visualize an example for the data matching three modules of the CHAMP satellite mission were implemented with dummy workflows.

- PRETOP-O
- CHBIAS
- ADDBIAS

The sequence for this chain can be seen in figure 28, where the used modules are marked by a red box.

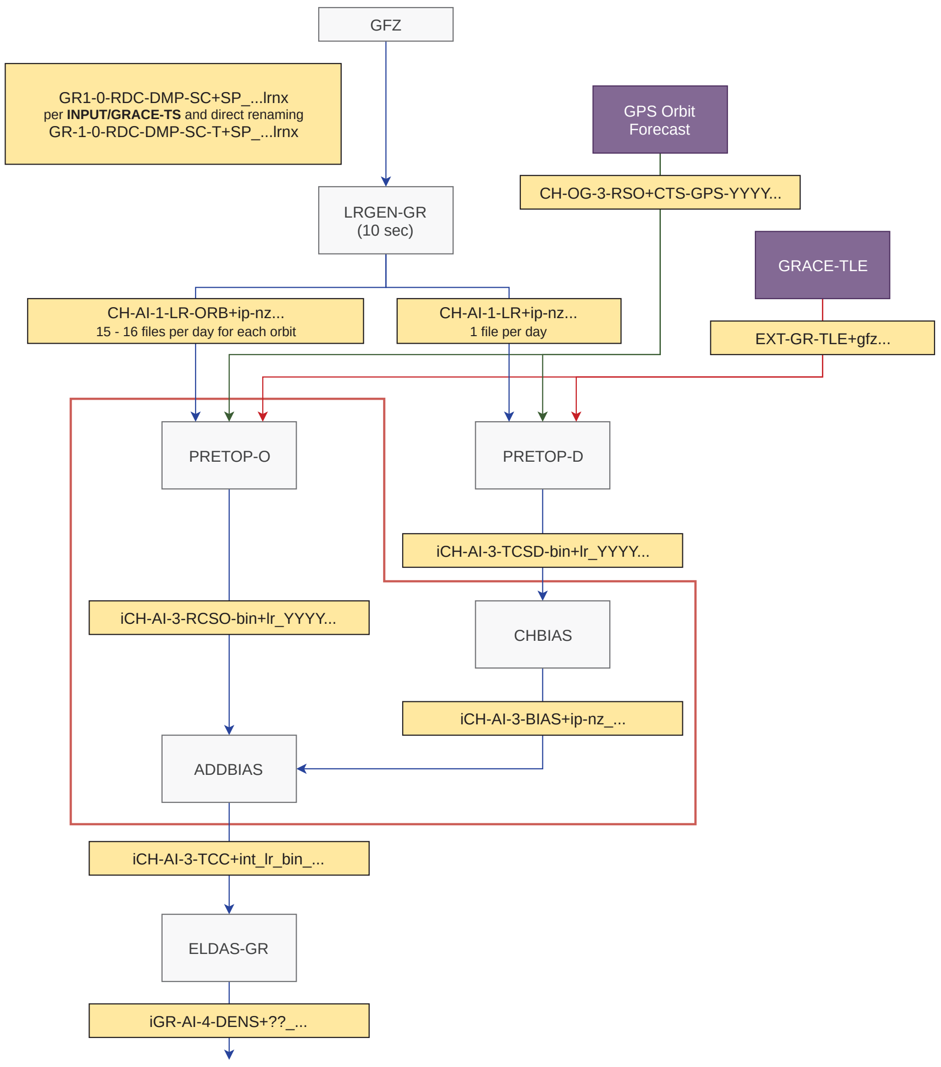

GFZ
GR1-0-RDC-DMP-SC+SP_...lrnx
per INPUT/GRACE-TS and direct renaming
GR-1-0-RDC-DMP-SC-T+SP_...lrnx
GPS Orbit Forecast
CH-OG-3-RSO+CTS-GPS-YYYY...
LRGEN-GR (10 sec)
GRACE-TLE
CH-AI-1-LR-ORB+ip-nz...
15 - 16 files per day for each orbit
CH-AI-1-LR+ip-nz...
1 file per day
EXT-GR-TLE+gfz...
PRETOP-O
PRETOP-D
iCH-AI-3-TCSD-bin+lr_YYYY...
iCH-AI-3-RCSO-bin+lr_YYYY...
CHBIAS
iCH-AI-3-BIAS+ip-nz_...
ADDBIAS
iCH-AI-3-TCC+int_lr_bin_...
ELDAS-GR
iGR-AI-4-DENS+??_...

Figure 28: Sequence of the example CHAMP modules (Own representation of Wehrenpfennig, 2002)

The PRETOP-O and the CHBIAS jobs listen on the creation of JSON files with the sample metadata in the corresponding buckets. The files then get copied and decoded. The decoder takes the metadata in the incoming JSON file and injects the product name, the bucket and the object key. Afterwards the data will be inserted into the database. Once this module is finished an empty file with the name of the original file plus the *.meta* extension is copied to the folder of the original file. This file symbolizes the existance of metadata in the database.

The ADDBIAS job listens on the creation of *.meta* files on the PRETOP-O and CHBIAS buckets. This means that the sensor of the job has multiple dependencies. Thus it has to connect the dependencies logically. In this case they are connected by a logical OR, which means that the creation of a *.meta* file in any of the locations triggers the job.
The job's first module will then query the database with the query previously shown in listing 8. For every row of the result of the query a module will be spawned which is responsible for the further processing.

6.12 Postprocessing

This section covers the usage or creation of postprocessing workflows.
The postprocessing works similiar to the operational system, but differ in the way the processes will be triggered. The postprocessing workflows will be triggered manually by the user and not by events.
The postprocessing workflows will be deployed and started or submitted using the Argo CLI. Configurations have to be provided through ConfigMaps. Specific files, that shall be used, have to be placed inside the Minio storage beforehand.
An example workflow is shown in appendix A.8. This workflow can be started by calling `argo submit -n <NAMESPACE> workflow.yml`.

7 Outlook

This section covers a summary of the new system created in this work and gives an overview of improvements that can be made in further developments.

7.1 Summary

The newly developed system represents a functional prototype of a processing system. It is capable of creating jobs on a schedule as well as in case of incoming data or files. Additionally those events can be combined to a collection of dependencies, for example the incoming of multiple files at a certain time.
The jobs itself will be defined as workflows - a sequence of processing steps - which have to have at least one processing step. Each job and each processing step can have input in form of arguments, environment variables or files, as well as output in form of files or plain text. Those in- and outputs, also known as artifacts, can be exchanged between the processing steps.
Each processing step runs a module, which is a script or a programm encapsulated inside a container. This enables the use of small, reusable modules regardless of the programming language. The modules will then be scheduled and executed by a Kubernetes cluster, which majorly enhances the horizontal scaling possibilities.
All artifacts of the system will be stored in a Minio S3-compatible storage instance. Thus files are provided by an HTTP interface. Additionally the web UI grants the user the download and review of the files.
Incoming data are subjected to a decoding process. This process will extract all metadata from the new file and will upload those to a PostgreSQL database. The files location as well as it's time of validity are stored in dedicated attributes, while additional metadata will be stored as JSON.

7.2 Possibilities of improvement

Since the implemented system is a prototype, not every part of it is in a finalized state by the time of writing. This section covers possibilities of improvement, which became clear during development and usage of the system. Those improvements can be categorized in four groups:

- Stability
- Security
- Usability

- General features

7.2.1 Stability

Most of the workflows in the prototype system where used for testing. Therefore those modules have not been optimized and offer some possibilities of improvement.

Resource Requests and Limits To ensure that a module only uses the amount of resources (CPU and Memory) it should need, resource requests and limits should be used. Those are configuration options provided by Kubernetes to limit the compute resources a container can use. Those definitions are also used by the Kubernetes scheduler to determine, if a node has enough resources available to run the container.

Volumes Modules that produce a high amount of write operations to the disk should use volumes. Those volume can be external drives, e.g. NFS shares. This way the overflow of the hosts storage will be avoided. Furthermore a clean up strategy can be defined, which allows to configure how the files should be treated once the module finished.

Improvements can also be made to other components of the system by eliminating single points of failure. This includes upgrading the system to a multi-master setup, which means that multiple of the Kubernetes cluster's hosts will be promoted to the master role. That way will be avoided, which will help stabilizing the system in case of a master node failing, thus improving the availability and reduce the downtime of the system. Additionally the etcd service[33] itself should run as a cluster with an odd amount of members. (See `https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/`)
To improve data and file availability and reduce downtime in case of the file storage or the database failing, those components should be run in a distributed architecture as well.

7.2.2 Security

To increase security all communcation paths should use a secure version of their respective protocolls, e.g. HTTPS. Therefore certificates should be provided and configured to be used by the corresponding services.

[33]consistent key value store used as Kubernetes' backing store for all cluster data (See The Kubernetes Authors, 2019d)

7.2.3 Usability

Since currently there is no support for wildcards or subdomains in the host network DNS, the access to the dashboards provided by the system is not user friendly. Configuring the support should enable the user to access those dashboards through URLs like `argo.<domain>.de` or `minio.<domain>.de`. Another possibility is to use external proxies which forward the requests to the cluster, so the services are made accessible by visiting the proxies domain. Another improvement to the usability of the system can be to ease the process of configuring workflows by providing a graphical user interface, which enables the user to configure his workflows using graphical elements rather than writing YAML.

7.2.4 Features

In the current system only one of the required environments is implemented, which is the testing environment. The other environments have to be created using the Ansible Playbook. Furthermore the environments have to be configured to be optimized for their use-case, like for example high availability for the operational processing environment. (See 7.2.1)
Another possibility of improvement could be the relocation of the systems components. For example the mid-term archive could be run inside, the logging and metrics services could be run outside of the cluster. Running all components inside a cluster would improve isolation of the different environments of the system. Running the services outside the clusters could improve accessibility since information about all environments are bundled in a centralized location.
Another feature could be the implementation of a REST API. Currently the query module directly communicates with the database. Thus the user has to know the location and the basic model of the database. To avoid this the REST API could be used to abstract the database communication. This would enable validation of queries, manipulation of results - thereby minimization of the query module - and would therefore improve the ease of use for the user.
In the current system the logs and metrics are visualized by two independent web frontends - Kibana and Grafana. Since Elastic also provides an APM[34] module, which also integrates with Prometheus, the perfomance monitoring can also be integrated into the Kibana frontend. (See `https://www.elastic.co/de/what-is/kubernetes-monitoring`) This way logs and metrics can be bundled on a central platform and the amount of software

[34]Application Performance Monitoring

products in the system can be reduced.

7.2.5 Interfaces

The developed system provides the fundamental basis for operational processing. To benefit from that, interfaces, documentations and guidelines have to be defined. Those should describe and make clear to the user how a module can be wrapped inside of a container and how it can be integrated into the system afterwards. This includes for example code snippets for commonly used setups, templates for Argo's Event sources, Sensors, Gateways and Workflows as well as guidelines for in- and output handling.
Since those things were not implemented in this work, they will have to be developed in the future.

Glossary

Amazon S3 (Simple Storage Server) is a cloud-based object storage service provided by Amazon with focus on scalibility and high availability. 45, 46, 54, 60, 61, 64–66, 72

Ansible is an IT automation tool, which can be used to configure systems, deploy software, and orchestrate more advanced IT tasks such as continuous deployments or zero downtime rolling updates (Red Hat, Inc., 2019a). 60–63, 74

API Application Programming Interface. 17, 43, 46, 53, 74, 76–79

Artifacts are a list of files and directories created by a job once it finishes (GitLab Inc., 2019b). 28–30, 47, 55, 63, 64, 68, 72, 76

Calico is an open source networking and network security solution for containers, virtual machines, and native host-based workloads (Project Calico, 2019). 42, 50, 52, 63, *see* CNI

Continuous Integration / Continuous Delivery (CI/CD) are methodologies to automatically execute scripts to test, deploy and deliver software and minimize the chance of introducing errors by human interactions (GitLab Inc., 2019a). 59

CLI Command-line Interface. 17, 30–33, 40, 42, 44–46, 52, 56, 60, 71, 77

CNCF Cloud Native Computing Foundation. 43

Container Network Interface (CNI) is a specification and a collection of libraries to implement container network plugins (Cloud Native Computing Foundation, 2019). 42, 52, *see* container

ConfigMap is a Kubernetes API object that allows decoupling of the configuration artifacts from the image content to keep containerized applications portable (The Kubernetes Authors, 2019d). 43, 55, 67, 68, 71

Container-orchestration systems allow users to control when containers start and stop, group them into clusters, and coordinate all of the processes that compose an application. Container orchestration tools allow users to guide container deployment and automate updates, health monitoring, and failover procedures. (Hewlett Packard Enterprise Development LP, 2019). 42, 43, 50, 52, 78, *see* container

Containers are a way to use OS-level virtualization. They are isolated environments that bundle software and all it's dependencies, e.g. libraries, configurations, etc. (Kofler, 2014; Liebel, 2017). i, 37–45, 50, 51, 54, 60, 61, 72, 76, 77, 94, *see* OS-level virtualization

Custom Resource Definitions (CRD) are extensions of the Kubernetes API that can be used to create custom Kubernetes objects (The Kubernetes Authors, 2019d). 45, 59

DAG Directed Acyclic Graph. 44, 45, 53, 54, 95

Deployments is a Kubernetes API object, which provides declarative updates for Pods and ReplicaSets (The Kubernetes Authors, 2019d). 43, 45, *see* pod & ReplicaSet

DIMS Data Information and Management System. 12, 18–21

Docker is a set of products that use OS-level virtualization to deliver software in packages called containers (Liebel, 2017). i, 37, 39–43, 50–52, 58, 60, 62, 90

Garbage collection is the process of automatically cleaning up ressources that are not used any more. 5, 6, 24, 30, 46, 53, 55, 66

Horizontal scaling describes the process of scaling by increasing the amount of nodes in a system rather than adding resources to the existing nodes (vertical scaling). 7, 16, 44, 79

Ingress is a Kubernetes API object that manages external access to the services in a cluster (The Kubernetes Authors, 2019d). 77

Ingress Controller is the software responsible for managing the Ingress resource in Kubernetes (The Kubernetes Authors, 2019d). 52, *see* ingress

kubeadm is a CLI for creating and managing Kubernetes clusters (The Kubernetes Authors, 2019b). 52, 63, *see* Kubernetes

kubectl is a CLI for running commands against Kubernetes clusters (The Kubernetes Authors, 2019d). 63, *see* Kubernetes

kubelet is the primary node agent that runs on each Kubernetes node (The Kubernetes Authors, 2019d). 63, *see* Kubernetes

Kubernetes is a container-orchestration system for automated application deployment, scaling and management across clusters of hosts (Liebel, 2017; The Kubernetes Authors, 2019c). i, 41, 43–45, 50–53, 55, 58–60, 62, 63, 72, 73, 76–79, *see* container-orchestration system

Load Balancer is a software that implements and provides load balancing strategies and mechanisms. 52, *see* load balancing

Load Balancing refers to efficiently distributing incoming network traffic across a group of servers (NGINX Inc., 2019). 43, 78

Namespace is a Kubernetes API object which helps creating isolated environments (The Kubernetes Authors, 2019d). 53, 55, 59

Object storage is a storage architecture that manages data as objects, instead of files or blocks. Object stores are supposed to raise the level of abstraction by moving functionalities, like space management, into the storage device itself (Factor et al., 2005). 46, 47

Open Container Initiative (OCI) is an open governance structure aiming to create open industry standards around container formats and runtimes (The Linux Foundation, 2019). 41, 42

OS-level virtualization refers to a virtualization mechanism, that does without real virtual machines, but rather uses the common kernel and parts of the host's file system (Kofler, 2014). 37, 39, 77

Pod is a Kubernetes API object representing the smallest deployable unit of computation in Kubernetes (The Kubernetes Authors, 2019d). 41, 43, 53, 77, 78

PSM Processing System Management. 19–21

Rule-based Access Control (RBAC) is Kubernetes default method of controlling access to resources based on the roles of individual users (The Kubernetes Authors, 2019d). 45

RDBMS Relational Database Management System. 44, 47, 79

ReplicaSet is a Kubernetes API object that is used to maintain a stable set of replica Pods running at any given time (The Kubernetes Authors, 2019d). 77, *see* pod

Scale-Out synonym for horizontal scaling. 7, 43, *see* horizontal scaling

Secret is a Kubernetes API object that allows storage and management of sensitive information, like passwords, tokens and keys (The Kubernetes Authors, 2019d). 43

Service is a Kubernetes API object that is used to expose applications or ports (The Kubernetes Authors, 2019d). 43, *see* pod

Sharding describes the process of automatic splitting of data on several servers. 7, 47–49

Single Point of Failure describes a part of a system that, in case of its failure, will stop the entire system from working (Wikipedia, 2019a). 7, 47, 49, 73

Structured Query Language (SQL) is a standardized language, that allows the user the communication with compatible RDBMS, including querying and manipulating data (Steiner, 2017). 48, 55–57

Virtual Machines are a way to emulate virtual hardware (CPU, RAM, storage, network, etc.) and allow to run arbitrary operating systems on this virtual host (Kofler, 2014). 37, 78

Volume is a Kubernetes API object that can be used to persist data by connecting it to storage drivers (The Kubernetes Authors, 2019d). 43, 55

YAML is a human friendly data serialization standard for all programming languages (The YAML Project, 2019). 50, 53, 74

List of Figures

List of Tables

List of Listings

References

ANDERSON, D., 2019. *metallb* [online]. Github, Inc. [visited on Oct. 26, 2019]. Available from: `https://github.com/danderson/metallb`.

APACHE SOFTWARE FOUNDATION, 2019. *Apache Hadoop 3.2.1 - HDFS Architecture* [online] [visited on Oct. 28, 2019]. Available from: `https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html`.

ARGO PROJECT, 2019. *Argo: Open source Kubernetes native workflows, events, CI and CD* [online] [visited on Oct. 19, 2019]. Available from: `https://argoproj.github.io/`.

BASU, Santimay; BASU, Sunanda, 1981. Equatorial scintillations - a review. *Journal of Atmospheric and Terrestrial Physics*. Vol. 43, no. 5-6, pp. 473–489. Available from DOI: `10.1016/0021-9169(81)90110-0`.

BAUER, R., 2018. *Whats the Diff: VMs vs Containers* [online]. Backblaze [visited on Dec. 6, 2019]. Available from: `https://www.backblaze.com/blog/vm-vs-containers/`.

BERNERS-LEE, T., 1994. *Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web*. RFC Editor. Available from DOI: `10.17487/RFC1630`. Technical report.

BÖTTCHER, M., 2004. *Design Document: Processing System Management*. Ed. by WERUM SOFTWARE AND SYSTEMS AG. Deutsches Fernerkundungsdatenzentrum.

BÖTTCHER, M.; REISSIG, R.; MIKUSCH, E.; RECK, C., 2001. Processing Management Tools for Earth Observation Products at DLR-DFD. Available also from: `https://www.dlr.de/eoc/en/Portaldata/60/Resources/dokumente/5_tech_dat/psm_dasia2001_paper_1.0.pdf`.

CHAE, M.; LEE, H.; LEE, K., 2019. A performance comparison of linux containers and virtual machines using Docker and KVM. *Cluster Computing*. Vol. 22, no. 1, pp. 1765–1775. ISSN 1573-7543. Available from DOI: `10.1007/s10586-017-1511-2`.

CLOUD NATIVE COMPUTING FOUNDATION, 2019. *CNI* [online] [visited on Nov. 11, 2019]. Available from: `https://github.com/containernetworking`.

CRI-O, 2019. *cri-o: Lightweight Container Runtime for Kubernetes* [online] [visited on Oct. 23, 2019]. Available from: `https://cri-o.io/`.

DOCKER INC, 2019. *Swarm mode key concepts* [online] [visited on Nov. 25, 2019]. Available from: `https://docs.docker.com/engine/swarm/key-concepts/`.

ELASTICSEARCH B.V., 2019. *Elastic Stack: Elasticsearch, Kibana, Beats und Logstash* [online] [visited on Nov. 4, 2019]. Available from: `https://www.elastic.co/de/products/elastic-stack`.

FACTOR, M.; METH, K.; NAOR, D.; RODEH, O.; SATRAN, J., 2005. Object storage: The future building block for storage systems. In: IEEE (ed.). *2nd International IEEE Symposium on Mass Storage Systems and Technologies*, pp. 119 –123. ISBN 0780392280. Available from DOI: `10.1109/LGDI.2005.1612479`.

GERMAN AEROSPACE CENTER (DLR), 2019a. *DLR strengthens Germany as a technology location with seven new institutes* [online] [visited on Jan. 20, 2020]. Available from: `https://www.dlr.de/content/en/articles/news/2019/02/20190627_dlr-strengthens-germany-as-a-technology-location.html`.

GERMAN AEROSPACE CENTER (DLR), 2019b. *Institute for Solar-Terrestrial Physics* [online] [visited on Jan. 20, 2020]. Available from: `https://www.dlr.de/so/en/desktopdefault.aspx/tabid-13420/23417_read-54014/`.

GITLAB INC., 2019a. *Introduction to CI/CD methodologies* [online] [visited on Oct. 30, 2019]. Available from: `https://docs.gitlab.com/ee/ci/introduction/index.html#introduction-to-cicd-methodologies`.

GITLAB INC., 2019b. *Introduction to job artifacts* [online] [visited on Oct. 29, 2019]. Available from: `https://docs.gitlab.com/ee/user/project/pipelines/job_artifacts.html`.

GLUSTER, 2019. *GlusterFS Documentation* [online] [visited on Oct. 28, 2019]. Available from: `https://docs.gluster.org/en/latest/`.

HAAS, R., 2018. *Built-in Sharding for PostgreSQL* [online]. EnterpriseDB Corporation [visited on Nov. 5, 2019]. Available from: `https://www.enterprisedb.com/blog/built-sharding-postgresql`.

HEISE, S.; JAKOWSKI, N.; WEHRENPFENNIG, A.; REIGBER, C.; LÜHR, H., 2002. Sounding of the Topside Ionosphere/Plasmasphere Based on GPS Measurements from CHAMP: Initial Results. *Geophysical Research Letters*. Vol. 29. Available from DOI: `10.1029/2002GL014738`.

HELM, 2019. *Helm: The package manager for Kubernetes* [online] [visited on Oct. 26, 2019]. Available from: `https://helm.sh/`.

HEWLETT PACKARD ENTERPRISE DEVELOPMENT LP, 2019. *What is Container Orchestration?* [online] [visited on Oct. 28, 2019]. Available from: `https://www.hpe.com/uk/en/what-is/container-orchestration.html`.

HOLLOSI, A., 2012. *Von Geodaten bis NoSQL: Leistungsstarke PHP-Anwendungen: Aktuelle Techniken und Methoden für Fortgeschrittene*. Munich: Hanser Verlag. ISBN 9783446431225.

JAKOWSKI, N.; WEHRENPFENNIG, A.; HEISE, S.; REIGBER, C.; LÜHR, H.; GRUNEWALDT, L.; MEEHAN, T., 2002. GPS Radio Occultation Measurements of the Ionosphere from CHAMP: Early Results. *Geophysical Reasearch Letters*. Vol. 29. Available from DOI: `10.1029/2001GL014364`.

KOFLER, M., 2014. *Linux: Das umfassende Handbuch*. 1. Auflage. Bonn: Rheinwerk Computing. ISBN 9783836225915.

KRIEGEL, M., 2012. *Operationelle Prozessierung satellitengestützter GNSS-Messungen zur Ableitung des Gesamtelektronengehalts (TEC)*. Master's thesis. Hochschule Neubrandenburg.

KRIEGEL, M.; JAKOWSKI, N.; BERDERMANN, J.; SATO, H.; MERSHA, M. W., 2017. Scintillation measurements at Bahir Dar during the high solar activity phase of solar cycle 24. *Annales Geophysicae*. Vol. 35, no. 1, pp. 97–106. Available from DOI: `10.5194/angeo-35-97-2017`.

LIEBEL, O., 2017. *Skalierbare Container-Infrastrukturen Das Handbuch für Administratoren*. Bonn: Rheinwerk Computing. ISBN 9783836243667.

MCCREA, I.; AIKIO, A.; ALFONSI, L.; BELOVA, E.; BUCHERT, S.; CLILVERD, M.; ENGLER, N.; GUSTAVSSON, B.; HEINSELMAN, C.; KERO, J.; KOSCH, M.; LAMY, H.; LEYSER, T.; OGAWA, Y.; OKSAVIK, K.; PELLINEN-WANNBERG, A.; PITOUT, F.; RAPP, M.; STANISLAWSKA, I.; VIERINEN, J., 2015. The science case for the EISCAT 3D radar. *Progress in Earth and Planetary Science*. Vol. 2. Available from DOI: `10.1186/s40645-015-0051-8`.

MINIO, INC., 2019. *Minio Documentation* [online] [visited on Oct. 28, 2019]. Available from: `https://docs.min.io/`.

NATIONAL OCEANIC AND ATMOSPHERIC ADMINISTRATION, 2019a. *F10.7 cm Radio Emissions* [online] [visited on Oct. 16, 2019]. Available from: `https://www.swpc.noaa.gov/phenomena/f107-cm-radio-emissions`.

NATIONAL OCEANIC AND ATMOSPHERIC ADMINISTRATION, 2019b. *Ionosphere* [online] [visited on Jan. 16, 2020]. Available from: `https://www.swpc.noaa.gov/phenomena/ionosphere`.

NGINX INC., 2019. *What Is Load Balancing?* [online] [visited on Oct. 28, 2019]. Available from: `https://www.nginx.com/resources/glossary/load-balancing/`.

NOJA, M., 2010. *Retrieval of plasmaspheric Total Electron Content from space-based GPS observations including a receiver differential code bias and multipath estimation*. Master's thesis. Hochschule Neubrandenburg.

PARK, J.; NOJA, M.; STOLLE, C.; LUEHR, H., 2013. The Ionospheric Bubble Index deduced from magnetic field and plasma observations onboard Swarm. *Earth, Planets, and Space*. Vol. 65, pp. 1333–1344. Available from DOI: `10.5047/eps.2013.08.005`.

PIGNALBERI, A.; PEZZOPANE, M.; TOZZI, R.; DE MICHELIS, P.; COCO, I., 2016. Comparison between IRI and preliminary Swarm Langmuir probe measurements during the St. Patrick storm period. *Earth Planets and Space*. Vol. 68. Available from DOI: `10.1186/s40623-016-0466-5`.

PROJECT CALICO, 2019. *About Calico* [online] [visited on Oct. 24, 2019]. Available from: `https://docs.projectcalico.org/v3.10/introduction/`.

PROMETHEUS AUTHORS, 2019. *Prometheus: Monitoring system and time series database* [online] [visited on Nov. 4, 2019]. Available from: `https://prometheus.io/`.

RED HAT, INC., 2019a. *Ansible Documentation* [online] [visited on Oct. 24, 2019]. Available from: `https://docs.ansible.com/ansible/latest/index.html`.

RED HAT, INC., 2019b. *CoreOS Container Linux Documentation* [online] [visited on Oct. 26, 2019]. Available from: `https://coreos.com/os/docs/latest/`.

RED HAT, INC., 2019c. *Fedora Project Wiki: EPEL* [online] [visited on Oct. 24, 2019]. Available from: `https://fedoraproject.org/wiki/EPEL#What_is_Extra_Packages_for_Enterprise_Linux_.28or_EPEL.29.3F`.

RED HAT, INC., 2019d. *rkt: A security-minded, standards-based container engine* [online] [visited on Oct. 28, 2019]. Available from: `https://coreos.com/rkt/`.

SATO, H.; KIM, J. S.; JAKOWSKI, N.; HÄGGSTRÖM, I., 2018. Imaging High-Latitude Plasma Density Irregularities Resulting from Particle Precipitation: Spaceborne L-Band SAR and EISCAT Observations. *Earth, Planets and Space*. Vol. 70. Available from DOI: `10.1186/s40623-018-0934-1`.

SQLITE CONSORTIUM, 2019. *Appropriate Uses For SQLite* [online] [visited on Nov. 26, 2019]. Available from: `https://sqlite.org/whentouse.html`.

STEINER, R., 2017. *Grundkurs Relationale Datenbanken: Einführung in die Praxis der Datenbankentwicklung für Ausbildung, Studium und IT-Beruf*. Wiesbaden: Springer Vieweg. ISBN 9783658179786. Available from DOI: `10.1007/978-3-658-17979-3`.

STOFFERS, M.; MEINEL, M.; WEIGEL, M.; SIGGEL, M.; FIEDLER, H.; RACK, K.; WASSER, Y., 2019. BACARDI: A System To Track Space Debris. In: FLOHRER, T.; JEHN, R.; SCHMITZ, F. (eds.). *1st NEO and Debris Detection Conference*. ESA Space Safety Programme Office.

THE APACHE SOFTWARE FOUNDATION, 2019. *Apache Airflow Documentation* [online] [visited on Nov. 26, 2019]. Available from: `https://airflow.apache.org/`.

THE KUBERNETES AUTHORS, 2019a. *ingress-nginx* [online]. Github, Inc. [visited on Oct. 26, 2019]. Available from: `https://github.com/kubernetes/ingress-nginx/`.

THE KUBERNETES AUTHORS, 2019b. *kubeadm* [online]. Github, Inc. [visited on Oct. 24, 2019]. Available from: `https://github.com/kubernetes/kubeadm`.

THE KUBERNETES AUTHORS, 2019c. *Kubernetes: Production-Grade Container Orchestration* [online] [visited on Oct. 21, 2019]. Available from: `https://kubernetes.io/`.

THE KUBERNETES AUTHORS, 2019d. *Kubernetes Documentation* [online] [visited on Oct. 29, 2019]. Available from: `https://kubernetes.io/docs/`.

THE LINUX FOUNDATION, 2019. *Open Container Initiative* [online] [visited on Nov. 5, 2019]. Available from: `https://www.opencontainers.org/`.

THE OPEN GROUP, 2017. *POSIX© 1003.1 Frequently Asked Questions (FAQ Version 1.16)* [online] [visited on Nov. 7, 2019]. Available from: `http://www.opengroup.org/austin/papers/posix_faq.html`.

THE YAML PROJECT, 2019. *The Official YAML Web Site* [online] [visited on Oct. 28, 2019]. Available from: `https://yaml.org/`.

TREASURE DATA, 2019. *Fluent Bit: Cloud Native Log Forwarder* [online] [visited on Nov. 4, 2019]. Available from: `https://fluentbit.io/`.

WEHRENPFENNIG, A., 2002. *CHAMP Processor Documentation: System Components*.

WEINSTABL, P., 2004. *PostgreSQL: Administration und Einsatz*. Boeblingen: Computer- und Literatur-Verlag. ISBN 9783936546224.

WENZEL, D.; JAKOWSKI, N.; BERDERMANN, J.; MAYER, C.; VALLADARES, C.; HEBER, B., 2016. Global ionospheric flare detection system (GIFDS). *Journal of Atmospheric and Solar-Terrestrial Physics*. Vol. 138-139, pp. 233–242. Available from DOI: `10.1016/j.jastp.2015.12.011`.

WERNICKE, M., 2017. *Untersuchung der Leistungsfähigkeit unterschiedlicher Cloud-Speicherdienste mit der Schnittstelle des Simple Storage Service (S3)*. Master's thesis. Frankfurt University of Applied Sciences.

WIKIPEDIA, 2019a. *Single point of failure* [online] [visited on Nov. 5, 2019]. Available from: `https://en.wikipedia.org/wiki/Single_point_of_failure`.

WIKIPEDIA, 2019b. *Slack (software)* [online] [visited on Oct. 17, 2019]. Available from: `https://en.wikipedia.org/wiki/Slack_(software)`.

A Appendices

A.1 Declaration on oath

I hereby assure you that I have written this Master's thesis without the help of third parties and only with the sources and aids indicated. All references taken from the sources are marked as such. This work has not yet been submitted to any examination authority in the same or similar form.

(Date)	(Signature)

A.2 Naming Conventions

1	2	3	4	5	6	7	8	9	10	11	12
Product / Owner	Acquisition Source	Acquisition Method	Level	Type	Mode	Reg. Cover	Start Time	End Time	DOY	Output Type	File Extension
DLR	JR055	GCI	L1A	TCAV	NC	EUROPE	2016-03-24T10-11-11	2016-03-24T10-12-10	002	D	.dat

Table 3: Naming conventions

A.3 Docker daemon configuration

```
{
    "exec-opts": ["native.cgroupdriver=cgroupfs"],
    "log-driver": "json-file",
    "log-opts": {
        "max-size": "100m"
    },
    "storage-driver": "overlay2",
    "storage-opts": ["overlay2.override_kernel_check=true"]
}
```

Listing 11: Docker daemon configuration

A.4 GitLab CI configuration

```
image: roffe/kubectl:latest

deploy:
  stage: deploy
  environment:
    name: deploy
  tags:
    - centos7
    - python
  script:
    - kubectl delete -f gateway/ || echo "No gateways to delete"
    - kubectl delete -f sensor/ || echo "No sensors to delete"
    - kubectl apply -f configmaps/
    - kubectl apply -f rbac/
    - kubectl apply -f secrets/
    - kubectl apply -f gateway/
    - kubectl apply -f sensor/

  only:
    - master
```

Listing 12: GitLab CI configuration example

A.5 Dockerfile examples

A.5.1 Python

```
FROM python:3.7.4-alpine

WORKDIR /opt/app

ADD src/ .
ADD setup.py .

RUN pip install .

CMD ["npsm", "-help"]
```

Listing 13: Dockerfile example using the NPSM module written in Python

A.5.2 Golang

```
######## BUILD STAGE #######
FROM golang:latest as builder

WORKDIR /opt/app

COPY go.mod go.sum ./
RUN go mod download

COPY . .

RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o main .

######## RUN STAGE #######
FROM alpine:latest

WORKDIR /opt/app

RUN apk -no-cache add ca-certificates
COPY -from=builder /opt/app/main .

EXPOSE 8080

CMD ["./main"]
```

Listing 14: Multistage Dockerfile example using Golang

A.6 Argo configuration examples

A.6.1 Container template

```
- name: npsm
  container:
    image: registry-endpoint:5000/npsm
    command:
      - npsm
    args:
      - 2019
      - 294
      - 13
```

Listing 15: Argo container template example

A.6.2 Script template

```
- name: decode
  script:
    image: registry-endpoint:5000/utilities
    command:
      - key-decode
    args:
      - "npsm%2F2019%2F294%2F08%2F20191021_0800_negrd.gif"
```

Listing 16: Argo script template example

A.6.3 Steps template

```
- name: npsm-steps-workflow
  steps:
    - - name: decode-key
        template: decode
    - - name: npsm-process
        template: npsm
```

Listing 17: Argo steps template example

A.6.4 DAG template

```
- name: npsm-dag-workflow
  dag:
    tasks:
      - name: decode-key
        template: decode
      - name: npsm-process
        template: npsm
        dependencies:
          - decode-key
```

Listing 18: Argo DAG template example

A.7 Example process configuration

A.7.1 Event source

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: plot-sct-event-source
  labels:
    argo-events-event-source-version: v0.10
data:
  scintillation: |
    bucket:
      name: scintillation
    endpoint: minio-endpoint:9090
    events:
      - s3:ObjectCreated:*
    filter:
      # prefix: ""
      suffix: ".sct"
    insecure: true
    accessKey:
      name: plot-sct-keys-secret
      key: accesskey
    secretKey:
      name: plot-sct-keys-secret
      key: secretkey
```

Listing 19: Event source configuration example

A.7.2 Gateway

```
apiVersion: argoproj.io/v1alpha1
kind: Gateway
metadata:
  name: plot-sct-gateway
  labels:
    gateways.argoproj.io/gateway-controller-instanceid: argo-events
    argo-events-gateway-version: v0.10
spec:
  processorPort: "9330"
  eventProtocol:
    type: "HTTP"
    http:
      port: "9300"
  template:
    metadata:
      name: "plot-sct-gateway"
      labels:
        gateway-name: "plot-sct-gateway"
    spec:
      containers:
        - name: "gateway-client"
          image: "argoproj/gateway-client"
          imagePullPolicy: "Always"
          command: ["/bin/gateway-client"]
        - name: "artifact-gateway"
          image: "argoproj/artifact-gateway"
          imagePullPolicy: "Always"
          command: ["/bin/artifact-gateway"]
      serviceAccountName: "default"
  eventSource: "plot-sct-event-source"
  eventVersion: "1.0"
  type: "artifact"
  watchers:
    sensors:
      - name: "plot-sct-sensor"
```

Listing 20: Gateway configuration example

A.7.3 Sensor

```
apiVersion: argoproj.io/v1alpha1
kind: Sensor
metadata:
  name: plot-sct-sensor
  labels:
    sensors.argoproj.io/sensor-controller-instanceid: argo-events
    argo-events-sensor-version: v0.10
spec:
  template:
    spec:
      containers:
        - name: "sensor"
          image: "argoproj/sensor"
          imagePullPolicy: Always
      serviceAccountName: default
  eventProtocol:
    type: "HTTP"
    http:
      port: "9300"
  dependencies:
    - name: "plot-sct-gateway:scintillation"
  triggers:
    - template:
        name: plot-sct-trigger
        group: argoproj.io
        version: v1alpha1
        kind: Workflow
        source:
          configmap:
            name: plot-sct-workflow-configmap
            key: wf
      resourceParameters:
        - src:
            event: "plot-sct-gateway:scintillation"
            path: s3.object.key
          dest: spec.templates.0.container.args.1
```

Listing 21: Sensor configuration example

A.7.4 Workflow

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: plot-sct-workflow-configmap
data:
  wf: |
    apiVersion: argoproj.io/v1alpha1
    kind: Workflow
    metadata:
      generateName: plot-sct-
    spec:
      ttlSecondsAfterFinished: 21600
      activeDeadlineSeconds: 120
      entrypoint: plot-sct
      volumes:
        - name: minio-connection-volume
          secret:
            secretName: plot-sct-minio-connection
        - name: plot-sct-settings
          configMap:
            name: plot-sct-settings
      templates:
        - name: plot-sct
          container:
            image: registry-endpoint:5000/sct-plotting-service:latest
            command:
              - plot-sct
            args:
              - "-object-key"
              - S3_KEY_PLACEHOLDER
              - "-cfg-plotting"
              - "settings/plotting/settings.json"
            volumeMounts:
              - name: minio-connection-volume
                mountPath: /opt/app/settings/minio/
              - name: plot-sct-settings
                mountPath: /opt/app/settings/plotting/
```

Listing 22: Workflow configuration example

A.8 Example postprocessing workflow

```
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: plot-sct-post-
spec:
  entrypoint: plot-sct-postprocessing

  volumes:
    - name: minio-connection
      secret:
        secretName: plot-minio-connection
    - name: plot-sct-settings
      configMap:
        name: plot-settings

  arguments:
    parameters:
      - name: s3-bucket
        value: scintillation

  templates:
    - name: plot-sct-postprocessing
      parallelism: 2
      steps:
        - - name: generator
            template: get-keys
        - - name: plot-sct-loop
            template: plot-sct
            arguments:
              parameters:
                - name: object-key
                  value: "{{item}}"
            withParam: "{{steps.generator.outputs.result}}"

    - name: get-keys
      script:
        image: registry-endpoint:5000/mc-python:latest
        volumeMounts:
          - name: minio-connection
            mountPath: /opt/app/settings/connection.yml
        env:
          - name: MINIO_BUCKET
            value: "{{workflow.parameters.s3-bucket}}"
        command:
          - "mc"
          - "find"
        args:
```

```
        - "-suffix"
        - "*.sct"

  - name: plot-sct
    inputs:
      parameters:
        - name: object-key
      artifacts:
        - name: tmp-sct
          path: /tmp/tmp.sct
          s3:
            bucket: "{{workflow.parameters.src-s3-bucket}}"
            key: "{{inputs.parameters.object-key}}"
            endpoint: minio-endpoint:9090
            accessKeySecret:
              name: minio-connection
              key: accesskey
            secretKeySecret:
              name: minio-connection
              key: secretkey
    container:
      image: registry-endpoint:5000/sct-plotting-service:latest
      volumeMounts:
        - name: plot-settings
          mountPath: /opt/app/settings/plotting/
      command:
        - plot-sct
      args:
        - "-cfg-plotting"
        - "settings/plotting/settings.json"
```

Listing 23: Workflow configuration for a postprocessing job