DAIKIRI
Diagnostische KI für industrielle Daten

DAIKIRI
Erklärbare Diagnostische KI für industrielle Daten

Project Number: 01IS19085 **Start Date of Project:** 01/01/2020 **Duration:** 30 months

Deliverable 5.2

Final Version and evaluation of DAIKIRI platform

Dissemination Level	Public
Due Date of Deliverable	Month 30, 30/06/2022
Actual Submission Date	Month 30, 30/06/2022
Work Package	WP5 — Platform
Task	T5.2, T5.3, T5.4 T5.5
Type	Report
Approval Status	Final
Version	1.0
Number of Pages	26

GEFÖRDERT VOM

Bundesministerium
für Bildung
und Forschung

This project has received funding from the Federal Ministry of Education and Research (BMBF) within the project DAIKIRI under the grant no 01IS19085.

History

Version	Date	Reason	Revised by
0.0	20/06/2022	Draft - White box	Ashraf Ibrahim
0.1	10/08/2022	Draft - Explainer	Sören Brunk Carolin Walter
0.2	17/08/2022	Draft - NFR	Pascal Schulz
1.0	18/08/2022	Final version submitted	Carolin Walter

Author List

Organization	Name	Contact Information
USU	Carolin Walter	carolin.walter@usu.com
USU	Sören Brunk	soeren.brunk@usu.com
PmOne	Ashraf Ibrahim	ashraf.ibrahim@pmone.com
PmOne	Pascal Schulz	pascal.schulz@pmone.com

Executive Summary

In this deliverable, we describe the integration of all DAIKIRI components created in work-packages 2-4 into our shared platform. The general setup and deployment of our shared platform, based on the workflow execution engine Flyte, is shown in deliverable 5.1.

Contents

1 Introduction

Based on the work done in packages 2-4 and the initial platform version (5.1), the goal of this deliverable is to sum up the development of the final DAIKIRI platform in *Flyte*. The objective of the work package is to integrate the individual work items into the *Flyte workflow engine*. For this purpose, various workflows were created, which correspond to the individual sections. A total of 4 workflows were implemented: 1) a *white box workflow*, 2) a *black box workflow*, 3) a *combined approach* consisting of the two preceding workflows and 4) a *SML workflow*, which is executed after adoptions made by the human in the loop. The *white box workflow* and the *black box workflow* were implemented as standalone version, to make them independently executable.

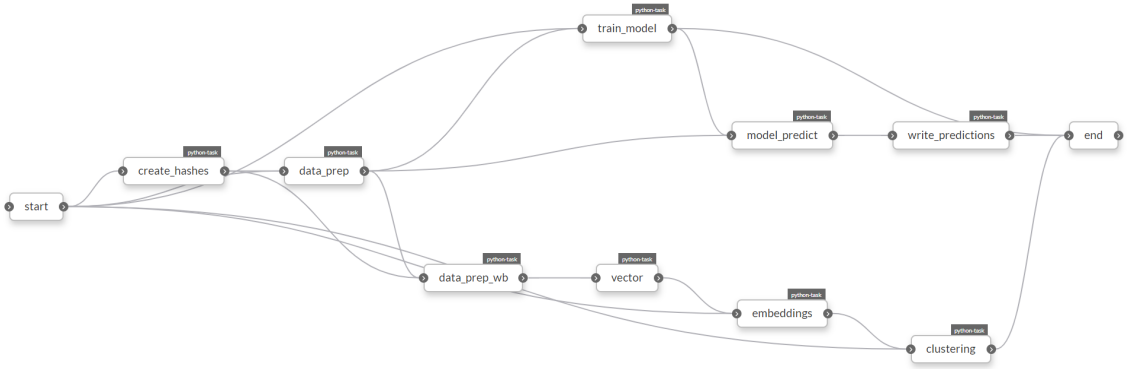

Figure 1: Combined workflow

The *combined workflow* thus can be divided into two parts: The *data_prep*, *train_model*, *model_predict*, *write_predictions* are implementations of the *black box workflow* and *data_prep*, *vector*, *embeddings*, *clustering* are part of the *white box workflow*. [1] The *sml workflow* is a separate process that is executed after completion of the combined approach.

[1] The name of Tasks in the standalone version can differ, to guarantee readability.

2 Data Retrieval

At first, we created a *data-retrieval* task, which aim is to gather new data for the pipeline. The Task reads new data from a specified location and processes the data if necessary to return a dataframe. The data retrieval task varies depending on whether the implementation is based on a combined pipeline adaption, where *white box* and *black box* approaches are combined, or whether its based on a standalone *white box* implementation. Both approaches are demonstrated here. For the sake of simplicity, we will focus on the *white box* approach for presenting the further implementations.

2.1 White box Approach

This task is logically separated into two different steps:

- Retrieving data
- Manipulating data

For demonstration purpose we are going to use a publicly available data-set (*pyrimidine* data), to explain the implementation of the pipeline steps.

```
@task()
def get_data()->pd.DataFrame:
    url ='https://www4.stat.ncsu.edu/~boos/var.select/pyrimidine.txt'
    s=requests.get(url).content
    df = pd.read_csv(io.StringIO(s.decode('utf-8')),sep=',')
    return df
```

The **def** keyword is a python specific keyword, initializing a function, followed by its name (**get_data()**) and a specification of the output of the function **->pd.DataFrame**[2]. The first two lines of code correspond to the *retrieval* step. Here, the data is consumed from a static URL and reduced to its content - this is saved to an internal variable called **s**. Printing this would return an unordered collection of strings:

> x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,x17,x18,x19,x20,x21,x22,x23,x24,
> x25,x26,y,0.5,0.26,0.1,0.9,0.9,0.9,0.1,0.367,0.42,0.26,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1,
> 0.367,0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.571,0.5,0.26,0.1,0.1,0.1,0.5,0.1,0.367,0.58,0.42,
> 0.26,0.1,0.9,0.1,0.9,0.1,0.1,0.26,0.9,0.367,0.1,0.1,0.1,0.5,0.367,0.9,0.9

The next step is equivalent to the idea of *data manipulation*. Using pandas, a flexible and open source data manipulation package for python, we are transforming the unordered collection to a dataframe, which is returned from our function and consumed by the later tasks:

x1	x2	x3	x4	x_n	y
0.5	0.26	0.1	0.9	$value_n$	0.571
0.5	0.26	0.1	0.1	$value_n$	0.900

The use of a stand-alone task for reading and processing data is particularly evident in regard to possible extensions. For example the Task can further be enhanced by defining the URL as input argument:

[2] pd.DataFrame is a pandas specific notation for a dataframe.

```
@task()
def get_data(url:str)->pd.DataFrame:
    '''
    Input:
            url : url for data retrieval
    '''
    s=requests.get(url).content
    df = pd.read_csv(io.StringIO(s.decode('utf-8')),sep=',')
    return df
```

In that case the URL would become dynamic and thus data from different inputs would be consumable. Surly the step of data manipulation needs to be adjusted in dependency to the data source. The usage of a stand-alone task, gives the opportunity, to implement a dynamic and input related data consumption with ease:

```
@task()
def get_data(url:str,filetype:str)->pd.DataFrame:
    '''
    Input:
            url : url for data retrieval
    '''
    if filetype == 'html':
            s=requests.get(url).content
            df = pd.read_csv(io.StringIO(s.decode('utf-8')),sep=',')
    elif filetype == 'csv':
        df = pd.read_csv(path=url)
    elif filetype == 'parquet':
        df = pd.read_parquet(path=url)
    else:
        # additional filetype
    return df
```

For our final platform, we used the Google Cloud and data stored on a bucket. A Bucket[3] is a file system for saving data on the google cloud. Every file in that case receives a certain URL:

`gs://bucketname/your_path_to_file.filetype`

where **gs** stands for **google storage**, **bucketname** for your globally unique name of your Bucket, followed by the path to your file and finally ending with your file type. This procedure is Cloud independent, because regardless which cloud is chosen, there always will be a URL to the respective file. For a local application, the URL then corresponds to the path of the file on your local system.

2.2 Black box Approach

The *data_prep* task of the black box approach preprocesses the data as needed for the subsequent training task. More details can be found in Walter and Witter [2022].

Enabling the task cache in combination with a cache version is helpful to reduce unneeded redundant computations. If there are no changes in the input parameters and the cache version is the same, Flyte does not execute the task at all, but re-uses the output from the actual computation. In most cases, downstream tasks can therefore read the hashes from cache.

[3] To connect to a Bucket or other types of cloud related storage's, permissions as well as IP rules must be considered.

Listing 1 Data Preparation for black box approach

```
@task(cache=True, cache_version="3", requests=Resources(cpu="2", mem="8G"))
def data_prep(input: pd.DataFrame, seed: int=1337) -> Dict[str, pd.DataFrame]:
    data = _preprocess_data(data=input)
    print('Loaded data')
    data = _join_anomaly_types(data=data)
    print('Created joined anomalies')
    feats = _compute_generic_features(feats=data, num_steps=3)
    print('Computed generic features')
    feats = _compute_specific_features(feats=feats, window_size=20)
    print('Computed specific features')
    feats = _clean_dataset(feats=feats)
    print('Cleaned dataset')
    datasets = _split_dataset(feats=feats, seed=seed)
    print('Split dataset')
    datasets = _drop_cols(data=datasets)
    return datasets
```

2.3 Combined Approach

The black box workflow is supposed to run with the same input data as the *white box workflow*, which we ensure through a combined workflow. For training and evaluation of the black box ML model the data needs to be splitted into train, evaluation and test set. Therefore, for every data point a hash is calculated over all features of this data point. This enables us later to refer to a event i.e. data point.

Listing 2 Hash creation

```
@task(cache=True, cache_version="3", requests=Resources(cpu="2", mem="8G"))
def create_hashes(input: CSVFile) -> pd.DataFrame:
    path = input.download()
    df = _load_data(path)
    df['supplier_item_number'] = df['supplier_item_number'].astype(str)
    df['hash'] = [hashlib.md5(str(row).encode('utf-8')).hexdigest() for index, row in df.iterrows()]
    df = df.set_index('hash')
    return df
```

The *data_prep* task of combined approach resembles a *pass through of input data*, where the data is processed in the same manner as for the black box workflow. But here only the test set data is used. Similar to the white box approach, the output is then consumed by the latter task.

```
@task(cache=True, cache_version="3", requests=Resources(cpu="2", mem="8G"))
def data_prep_wb(input: pd.DataFrame, data: Dict[str, pd.DataFrame]) -> pd.DataFrame:
    df = _get_split(hashed = input, splited = data)
    return df
```

3 Integration Embeddings

The embeddings part of the *white box pipeline* consists of a vectorization task and an embeddings task. The implementation is explained in Demir [2021].

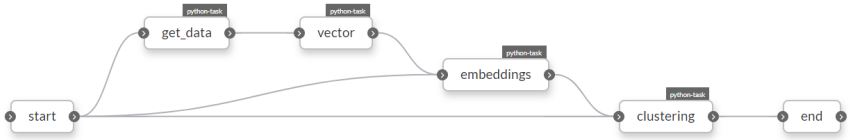

Figure 2: White box workflow

3.1 Vectorization

Similar to the previous topic, we have created a *Flyte* task to perform the vectorization:

```
# Importing vectograph library
from vectograph.transformers import GraphGenerator
from vectograph.quantizer import QCUT

@task()
def vector(df: pd.DataFrame) -> pd.DataFrame:
    X_transformed = QCUT(min_unique_val_per_column=5
    ,num_quantile=4,duplicates='drop').transform(df.rank(method='first'))
    X_transformed.index = 'Event_' + X_transformed.index.astype(str)
    kg = GraphGenerator().transform(X_transformed)
    # Create Dataframe for Flyte transmission
    df=pd.DataFrame(kg)
   # Map Colnames tp string for parquet
    df.columns = df.columns.map(str)
    return df
```

Description for the steps used here, e.g. **QCUT**, can be found in Demir [2021]. To use the vectograph library, the *Flyte* workflow needs to import the **GraphGenerator** and the **QCUT** function. For demonstration purposes, we use the test data mentioned above. The **QCUT** function returns an ontology based dataframe, where each row corresponds to an event:

Index	Feature_Category_x1	Feature_Category_x2	Feature_Category_xn
Event_0	x1_quantile_3	x2_quantile_1	xn_quantile_n
Event_1	x1_quantile_3	x2_quantile_1	xn_quantile_n
Event_2	x1_quantile_1	x2_quantile_2	xn_quantile_n
Event_3	x1_quantile_0	x2_quantile_3	xn_quantile_n

The following line of code alters the index column to be more readable with adding 'Event_' previous to the numeric value. This also is necessary as the **GraphGenerator** needs a string based input

for the index column. The created *X_transformed* object is then consumed by the *GraphGenerator*, returning a knowledge graph which is stored as an object named *kg*. This array type object is then transformed to a dataframe:

col_0	col_1	**col_2**
Event_0	Feature_Category_x1	x1_quantile_3
Event_0	Feature_Category_x2	x2_quantile_1
...	...	...
Event_n	Feature_Category_xn	xn_quantile_n

Flyte stores data in between tasks as parquet files, which needs columns to have string names. To ensure string column names, the following line of code `df.columns = df.columns.map(str)` was added to make the pipeline robust against different column name notations. As above, the final dataframe is returned and thus made consumable for the **Embeddings** task.

3.2 Embeddings

As in the previous section, the embeddings (Demir [2021]) part was implemented as a standalone task within the *Flyte* workflow. Unlike vectorization, this task uses input arguments that must be entered into *Flyte* via the workflow interface. Which arguments are entered via the interface and which are provided with a default value can be varied as desired. However, it is important that the *surface arguments* are passed on to the workflow task. The following variable arguments have been defined:

Variable Arguments	Description
model	The model for the Knowledge Graph Embedding
learning_rate	Learning Rate of embeddings model
num_folds_for_cv	Number of cross Validation Folds
max_num_epochs	Max. number of epochs for converging

The dataframe input corresponds to the output of the previous task.

```
# Importing executioner from embeddings
from executer import Execute

@task()
def embeddings(filename: pd.DataFrame,model: str,learning_rate: float
,num_folds_for_cv: int,max_num_epochs:int) -> pd.DataFrame:

    argparser = argparse.ArgumentParser()

    args = argparse.Namespace(path_dataset_folder=filename,
                              model="{}".format(model),
                              num_folds_for_cv=num_folds_for_cv,
```

```
                                    max_num_epochs=max_num_epochs,
                                    check_val_every_n_epochs=1000,
                                    embedding_dim=10,
                                    scoring_technique='NegSample',
                                    learning_rate=learning_rate,
                                    eval=1,
                                    large_kg_parse=1,
                                    deserialize_flag=None,
                                    add_reciprical=False,
                                    read_only_few=0,
                                    storage_path='DAIKIRI_Storage',
                                    negative_sample_ratio=1,
                                    batch_size=512,
                                    shallom_width_ratio_of_emb=1.5,
                                    input_dropout_rate=0.1,
                                    hidden_dropout_rate=0.1,
                                    feature_map_dropout_rate=0.1,
                                    apply_unit_norm=False,
                                    num_workers=1

                                    )


    exec = Execute(args)
    df=exec.start()
    # Map Colnames tp string for parquet
    df.columns = df.columns.map(str)
    return df
```

The original framework for embeddings was written as a CLI tool, which means that it was intended and designed to be executed via command line. This is obvious due to the use of the python based *argparse* package, which allows exactly this type of execution. In order to not rewrite the code and to ensure the functionality of some dependencies within submodules, the arguments selected for input are stored as `args=argparse.Namespace()`. The Namespace class stores the elements with a string representation, which then can be parsed into the execution framework.

The embeddings framework was further adapted in that the execution was initiated via the immediate execution script. For this purpose, the input arguments - both the standardized ones and the ones defined at the interface - were transmitted to the execution script and an execution was initiated.

```
# Execute and starting embeddings
exec = Execute(args)
df=exec.start()
```

The result again is a dataframe containing the embeddings, which corresponds to the results of the knowledge graph embedding model. Following the example data from above, the embeddings would return a result similar to this:

	0	1	2	3	4	5	6	7	8	9
Event_0	0.06	-0.031	-0.11	0.01	0.159	-0.031	-0.119	0.110	0.08	-0.086
...	...	...	...	...	...	...	...	...	...	...
Event_*n*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*

The task returns this dataframe and follows the procedure mentioned above, where numerical columns are converted to *string*, so they can be stored as parquet.

4 Integration Semantification

4.1 Clustering

Explanations on the functionality as well as the purpose of the clustering task can be found here Zahera et al. [2020]. This element of the pipeline was also designed as an independent task, which had variable input arguments similar to the embeddings. Likewise, this CLI based framework was also converted using a similar approach as above.

Furthermore, it was also designed to let the user define variable input arguments on the *Flyte* interface. For the following task, these values were defined as manually determinable:

Variable Arguments	Description
model	The clustering model: *kmeans*, *hdbscan* or *agglomerative*
alpha	The degree of conservativeness in cluster selection (only for *hdbscan*)
min_samples	The minimum number of neighbours to a core point (only for *hdbscan*)
cluster_selection_epsilon	Cluster radius definition (only for hdbscan)
min_cluster	The minimum number of points per cluster (only for hdbscan)
n_cluster	The number of clusters in total (only for *kmeans* and *agglomerative*)

However, there is a difference to the previous one: the input arguments to determine specifics of the clustering procedure must be passed as a config file. Following the same principles as above, this was implemented as part of the *Flyte* task, by creating a dictionary based on the given inputs. This corpus becomes part of the input arguments and parsed to the specific clustering method:

```
config = {
        "alpha" : alpha,
        "metric" : metric,
        "cluster_selection_method": cluster_selection_method,
        "min_samples": min_samples,
        "min_cluster_size": min_cluster_size,
        "allow_single_cluster": allow_single_cluster,
        "cluster_selection_epsilon": cluster_selection_epsilon,
        "n_clusters": n_cluster
    }
```

Static or default values were added at the *Flyte* workflow code block. To select a clustering method, an if statement was added, which initiates a certain method based on the input for *model* argument.

```python
# Importing clusering library
from Clustering import *

@task()
def clustering(file: pd.DataFrame, model: str,alpha: float,
                cluster_selection_method: str,min_samples: int,
                min_cluster_size: int, allow_single_cluster: bool,
                cluster_selection_epsilon: float,
                n_cluster: int) -> pd.DataFrame:

    # config file for static input
    config = {
        "alpha" : alpha,
        "metric" : metric,
        "cluster_selection_method": cluster_selection_method,
        "min_samples": min_samples,
        "min_cluster_size": min_cluster_size,
        "allow_single_cluster": allow_single_cluster,
        "cluster_selection_epsilon": cluster_selection_epsilon,
        "n_clusters": n_cluster
    }

    if model == "agglomerative":
        cluster = Agglomerative_Clustering(data=file, config=config)
        df=cluster.return_Output()
    elif model == "kmeans":
        cluster = Centroid_Clustering(data=file, config=config)
        df = cluster.return_Output()
    elif model =="hdbscan":
        cluster = Density_Clustering(data=file, config=config)
        df = cluster.return_Output()
    else:
        raise ValueError("No valid Custering submitted")


    return df
```

The task returns a dataframe, containing clustered entities for each embedding defined earlier. Using the clustering task on the previously returned embeddings dataframe, would thus return the following:

Entity	Cluster
Event_0	0
Event_1	0
Event_2	2
Event_*n*	*value*

4.2 Axiom Generation

The axioms are generated via the LabENT module. More information about LabENT can be found in Zahera and Heindorf [2021]. With this tool, human interaction is intended to (re-)label clusters. Afterwards, the ontology can be written to an OWL file. LabENT is deployed on the same cluster as the DAIKIRI platform to easily enable interactions.

5 Integration Structured Machine Learning

Structured machine learning is wrapped in a separate workflow since the afore achieved output of the semantification process can be first adjusted manually via LabENT (chapter 4.2). After adoptions made by a human in the loop the *SML workflow* can be started. The *SML workflow* can be executed after the *white box workflow*, after the *combined workflow* or as standalone on external input data.

5.1 Structured Machine Learning

Listing 3 SML taks definition

```
from ontolearn.concept_learner import CELOE
from ontolearn.model_adapter import ModelAdapter
from owlapy.model import OWLNamedIndividual, IRI, OWLOntology, OWLReasoner
from owlapy.namespaces import Namespaces

@task(cache=True, cache_version="1")
def SML(ontology: FlyteFile, pos_examples: FlyteFile, neg_examples: FlyteFile,
        max_hypotheses: int = 50)-> FlyteFile:
    ontology = ontology.download()
    pos_examples = pos_examples.download()
    neg_examples = neg_examples.download()
    NS = Namespaces('ex', 'http://daikiri-semantificaion.de/onto.owl#')

    positive_examples = read_examples(NS, pos_examples)
    negative_examples = read_examples(NS, neg_examples)

    # Only the class of the learning algorithm is specified
    model = ModelAdapter(learner_type=CELOE,
                         reasoner_factory=ClosedWorld_ReasonerFactory,
                         path=ontology)

    model.fit(pos=positive_examples, neg=negative_examples)
    model.save_best_hypothesis(n=max_hypotheses, path='predictions')
    return FlyteFile(path="predictions.owl")
```

For structured machine learning the open-source software library Ontolearn[4] is used. Ontolearn provides efficient solutions for concept learning on RDF knowledge bases. It uses an OWL ontology

[4] https://github.com/dice-group/Ontolearn

and two txt files with positive and negative examples. Within the SML function the namespace is hard-coded and needs to be adjusted for an other use case.

With **ModelAdapter** a new concept learner is created and trained afterwards. Subsequently, the best hypothesis are saved.

5.2 Verbalization

The verbalization component in DAIKIRI is based on the daikiri-verbalizer open-source project[5] which is written in Java. This is in contrast to most other DAIKIRI components, which are written in Python. We first tried to use the Flyte Java API to implement the task, but we soon ran into issues with authentication, as the flytekit-java API is not as mature as its Python counterpart. So we decided to use another approach to integrate verbalization into our workflow. This approach leverages the raw-container support of Flyte. From our Java verbalization project, we build a standalone Docker container that reads inputs and writes outputs from and to specific locations and in a format understood by Flyte. Then we use a raw-container task that executes the container. It takes the SML model and a semantification ontology as input, runs the verbalization process and returns explanations for the input concepts in natural language as CSV file. The raw-container task definition is shown in listing 4.

Listing 4 Raw-container task definition for verbalization

```
verbalize = ContainerTask(
    name="verbalize",
    input_data_dir="/var/inputs",
    output_data_dir="/var/outputs",
    inputs=kwtypes(input=FlyteFile, semantification=FlyteFile),
    outputs=kwtypes(verbalizations=CSVFile),
    image="europe-west4-docker.pkg.dev/usuai-daikiri/flyte-daikiri/verbalization:0.1.0",
    command=[
        "/opt/docker/bin/verbalization",
        "/var/inputs",
        "/var/outputs",
    ],
)
```

Finally, once we have defined the raw-container task, we can use it as part of the workflow just as any other task. Figure 3 shows the workflow for SML and verbalization.

Figure 3: SML workflow

[5] https://github.com/dice-group/daikiri-verbalizer

6 Black box Workflow and Simple Verbalization

For the *black box workflow*, we took the black box implementation described in Walter and Witter [2022], and converted it into Flyte tasks. A standalone version of the *black box workflow* looks as follows:

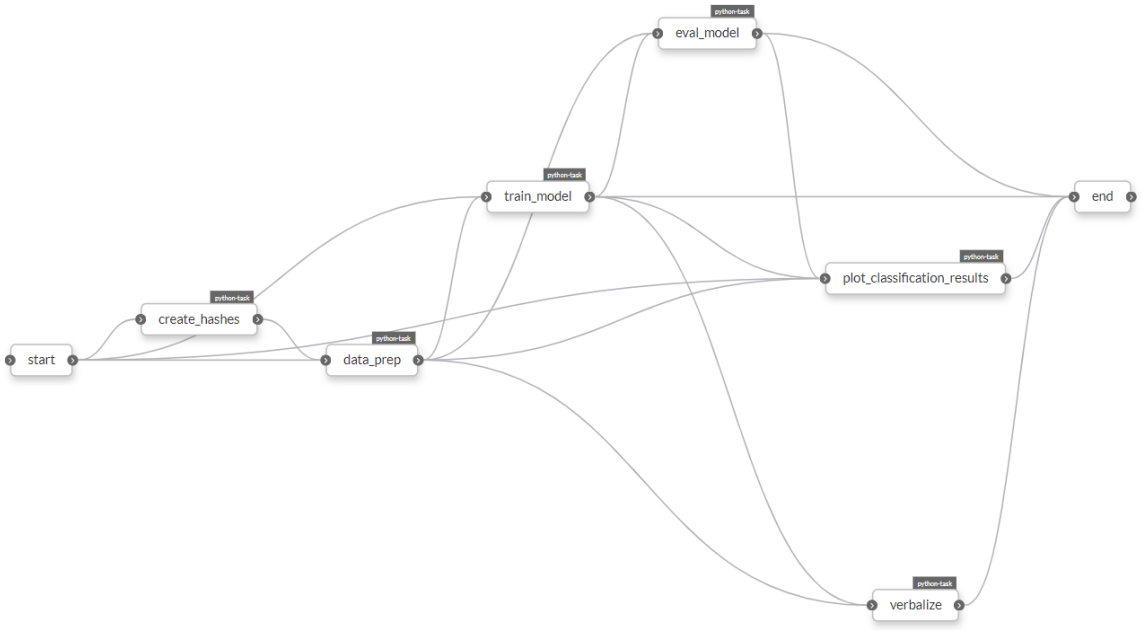

Figure 4: black box workflow

Listing 5 black box workflow with simple verbalizations

```
Output = NamedTuple("Output", model=JoblibSerializedFile,
                    eval=Dict[str, float], plot=PNGImageFile, verbalizations=pd.DataFrame)

@workflow
def pipeline(
        input: CSVFile,
        ano_ignore: str = ano_types[0],
        seed: int = 1337) -> Output:
    hashed_data = create_hashes(input=input)
    data = data_prep(input=hashed_data, seed=seed)
    model_file = train_model(data=data, ano_ignore=ano_ignore, seed=seed)
    eval = eval_model(data=data, model_file=model_file)
    plot = plot_classification_results(
      data=data, model_file=model_file, ano_ignore=ano_ignore, seed=seed, perf=eval)
    verbalizations = verbalize(data=data, model_file=model_file)
    return Output(model=model_file, eval=eval, plot=plot, verbalizations=verbalizations)
```

After hash generation and data preparation as already treated in chapter 2.3 and chapter 2.2 a ML model is learned, evaluated and the classification results are plotted. In case of the *black box workflow*

a simple verbalization mechanism is used. First, SHAP values are calculated for the test data split. Then, based on the SHAP values, all data point get a prediction and are explained by their most important features and their actual values. This information is put together in natural language and saved to a csv file.

7 Combined Approach

This approach uses the whole white box as it is and combines it in a smart way with the black box approach. The data is labeled by the black box approach and the model explanations are extracted by the *white box* and *SML workflow*. The *combined workflow* begins with the common task of `create_hashes`. After hash creation the workflow splits into two branches. One for the black box model and another one for the white box model.

Listing 6 Pipeline for Combined Workflow

```
Output = NamedTuple("Output", bb_model=JoblibSerializedFile,
                    pred_out=PredictionOutputs, cluster=pd.DataFrame)

@workflow
def pipeline(
        input: CSVFile,
        ano_ignore: str = ano_types[0],
        seed: int = 1337,
        model: str = "Shallom",
        learning_rate: float = 0.01,
        cv: int = 2,
        epoch: int = 2,
        cluster_model: str = 'kmeans',
        n_clusters: int = 5,
        min_cluster_size: int = 4) -> Output:
    hashed_data = create_hashes(input=input)
    # BB:
    data = data_prep(input=hashed_data, seed=seed)
    model_file = train_model(data=data, ano_ignore=ano_ignore, seed=seed)
    pred = model_predict(model_file=model_file, data=data)
    pred_out = write_predictions(pred=pred)
    # WB:
    df = data_prep_wb(input=hashed_data, data=data)
    df_vec = vector(df=df)
    df_embed = embeddings(filename=df_vec, model=model, learning_rate=learning_rate,
                          num_folds_for_cv=cv, max_num_epochs=epoch)
    cluster = clustering(file=df_embed, alpha=0.1, metric="precomputed",
                         cluster_selection_method="leaf", model=cluster_model,
                         min_samples=10, min_cluster_size=min_cluster_size,
                         allow_single_cluster=True, cluster_selection_epsilon=0.9,
                         n_cluster=n_clusters)
    return Output(bb_model=model_file, pred_out=pred_out, cluster=cluster)
```

Since there are no further data dependencies after the creation of the hashes, Flyte will automatically run the white box and the black box part of the workflow in parallel.
The outputs of the pipeline are a file of the black box model, the predictions for the test data i.e. two files with data points which are positive and negative examples, respectively, and the clusters. These outputs can then serve as input into the *SML workflow* (see chapter 5).

8 Coverage of non-functional requirements to ensure practical usability

System requirements are divided into functional and non-functional requirements. This chapter deals with the definition, classification and evaluation of non-functional requirements. In the first subchapter, various definitions are compiled and summarized. Based on this summary, non-functional requirements of the DAIKIRI platform are derived. The last section discusses the evaluation of the elicited non-functional requirements.

8.1 What are non-functional requirements?

A large number of different definitions can be found in the literature that define non-functional requirements (NFR). Glinz [2007] has collected different definitions from the literature to identify similarities. The results of his work can be found in Table 8.
After collecting and considering several definitions of NFR from the literature, it becomes clear that there is no uniform consent in the literature on how NFR are defined. Two important works in the NFR framework literature Anton [1997] and Davis [1993] are worth highlighting. These works distinguish functional from other quality attributes while considering cost, time, and productivity. This subdivision differs from previous works, which define only requirements in terms of detailed functions, constraints and attributes.

8.2 Classification of non-functional requirements

As with the definition of NFR, there are a variety of different classification approaches in the literature. The ISO/IEC 9124 standard [Clements [2006]] describes that NFR can be classified into four different classes. The approach distinguishes between quality during use, external quality, internal quality, and process quality. The acronym FURPS stands for a model for classification of software quality attributes or non-functional requirements, which was developed by Hewlett Packard. Later the model was extended by additional attributes and renamed to FURPS+ [Robertson and Robertson [1999]].

- **Functionality** – Functionality, capabilities, generality, security
- **Usability** – Human factor, aesthetics, consistency, documentation
- **Reliability** – Frequency/severity of failure, recoverability
- **Performance** – Speed, efficiency, resource consumption, response time
- **Supportability** – Testability, extensibility, adaptability, maintainability, configurability, installability, transferability

For the DAIKIRI research project, we will make the classification of non-functional requirements according to FURPS.

8.3 Identification of non-functional requirements

For this chapter, the previous definitions and classification approaches are used to survey NFRs of the DAIKIRI platform.
Performance summarized requirements that are relate to the speed of the DAIKIRI platform. Under the point speed the performance of the single tasks of a workflow as well as the performance of a complete run is considered. Also the performance improvement of the parallelization is added.
Requirements such as aesthetics or consistency can be summarized under the point usability. However, not only subjective points such as appearance are taken but also the existence of a documentation.

Source	definition
Anton [1997]	Describe the non-behavioral aspects of a system, by capturing the characteristics and constraints under which a system must operate.
Davis [1993]	The required overall characteristics of the system, including portability, reliability, efficiency, human engineering, testability, understandability, and modifiability.
Committee et al. [1990]	The term is not defined. The standard distinguishes between design requirements, implementation requirements, interface requirements, performance requirements and physical requirements.
Committee and Board [1998]	The term is not defined. The standard defines the categories of functionality, external interfaces, performance, attributes (portability, security, etc.) and design constraints. Project requirements (e.g., schedule, cost, or development requirements) are specifically excluded.
Jacobson [1999]	A requirement that specifies system properties, such as environmental and implementation performance, platform dependency, maintainability, extensibility, and reliability. A requirement that specifies physical constraints for a functional requirement.
Kotonya and Sommerville [1998]	Requirements that do not relate specifically to the functionality of a system. They set constraints for the product to be developed and the development process, and they specify external conditions that the product must meet.
Mylopoulos et al. [1992]	Global requirements for development or operating costs, performance, reliability, maintainability, portability, robustness, and the like. (...) There is neither a formal definition nor a complete list of non-functional requirements.
Ncube [2000]	The behavioral characteristics that the specified functions must exhibit, e.g., performance, ease of use.
Robertson and Robertson [1999]	A characteristic or quality that the product must have, such as a certain appearance, speed, or accuracy characteristic.

Table 8: NFR defintion

An important point when developing an application is the reliability of the application. Non-functional requirements, which are summarized under the point Reliability, are failure frequency and recoverability of the application.
The last category we divide NFR into is Supportebility. The category includes requirements such as transferability of the application to the project sponsor.

8.4 Evaluation of non-functional requirements of the DAIKIRI platform.

In the previous chapter, the NFRs were divided into four categories: performance, usability, reliability and supportability. This chapter deals with the evaluation of these four NFRs.

Performance

The backend of the DAIKIRI platform is based on the open source system Kubernetes, which was developed by Google[6]. Through Kubernetes it is possible to parallelize workflows, tasks or calculations in DAIKIRI. Parallelization allows multiple tasks to be executed simultaneously, which improves runtime. The figure below depicts the workflow "workflows.combined.pipeline.pipline". The workflow shown trains the black-box and white-box model. A parallelization strategy is particularly useful for training the two models. In the figure 5 it can be seen that the first two tasks *create_hashes* and *data_prep* must run sequentially. The subsequent task black-box (*data_prep_wb*, *vector*, *embeddings* and *clustering*) and white-box (*train_model*, *model_predict* and *write_predictions*) can be trained in parallel.

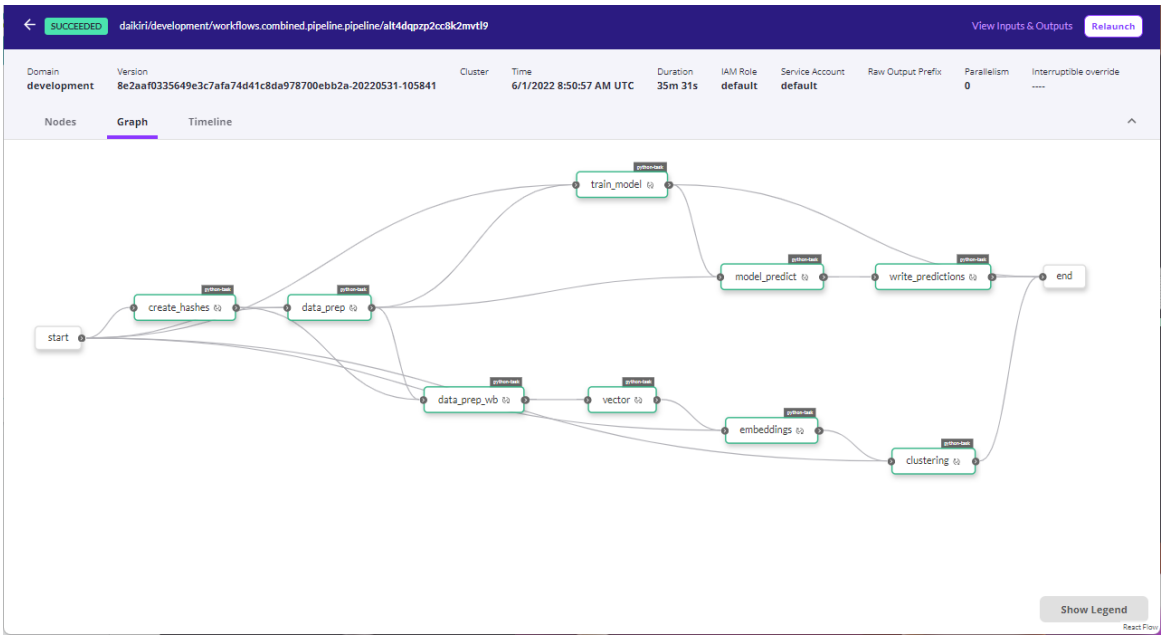

Figure 5: black box and white box workflow

The Gantt chart in the figure 6 shows how DAIKIRI automatically parallelizes the different tasks. On the figure 6 it can be seen that after the task *data_prep* the two models black-box and white-box run in parallel. In total, 5 tasks are parallelized in the "workflows.combined.pipeline.pipline".

[6] https://cloud.google.com/learn/what-is-kubernetes?hl=de

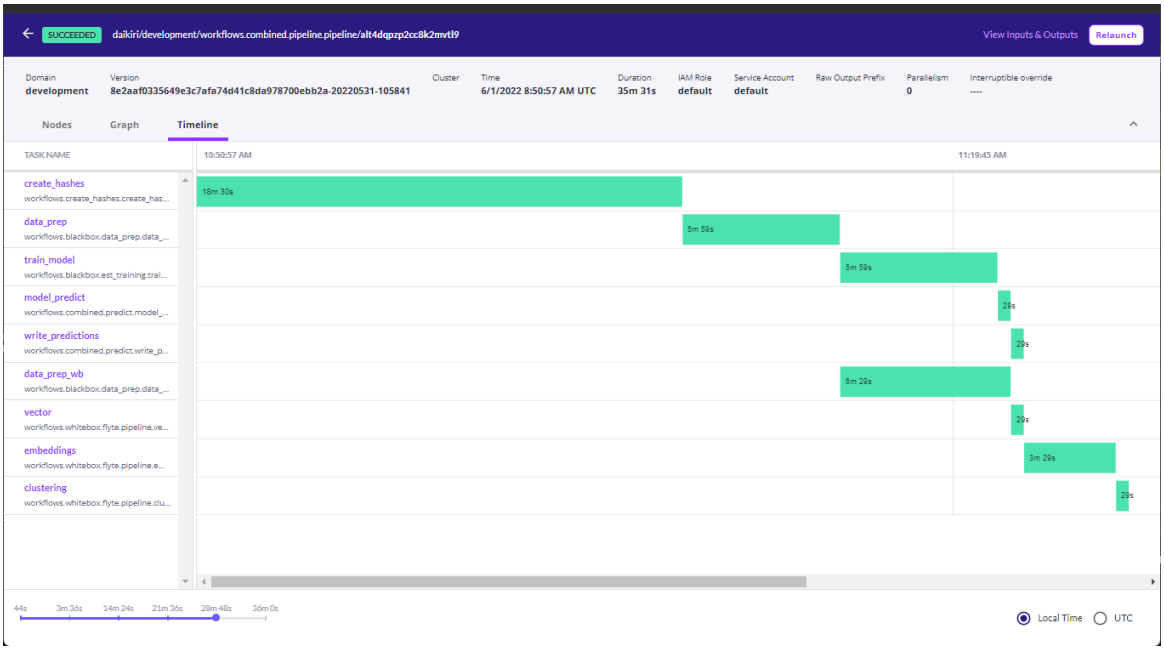

Figure 6: grantt chart black box and white box workflow

Parallelization can reduce the runtime by 6:22 minutes. This is an improvement of about 16%

Usability

The evaluation of usability is difficult because this requirement is a subjective judgment. To overcome this problem, we are concerned with the design or arrangement of the elements within the user interface (UI) of DAIKIRI. The requirement for the UI is to provide the user with an intuitive user interface. The UI of the DAIKIRI platform can be divided into three parts. On the left side there is a navigation bar that allows the user to navigate between 3 different areas. The user has the possibility to view the project dashboard as well as access the workflows via the navigation bar. The last area that is accessible to the user via the navigation bar are the individual tasks of a workflow.
The main focus of the DAIKIRI UI is in the middle and right area of the UI. All relevant information about the three areas is displayed there. For example, all existing workflows are displayed there with additional information. During the conceptual design of DAIKIRI, relevant software development requirements for usability were also taken into account. Software developers have the possibility to develop in the three environments . In the context of the DAIKIRI platform, the environments are Development, Staging and Production. These development environments can be accessed via the navigation bar at the top of the page. It was particularly important within DAIKIRI to maintain a consistent naming convention. Thus, as shown in the figure 7, the workflows are defined according to the following scheme:

- A workflow begins with *workflows*
- After that comes the function of the workflow
- Finally, the word *pipeline* comes twice.
- All three previous items are separated with a ..

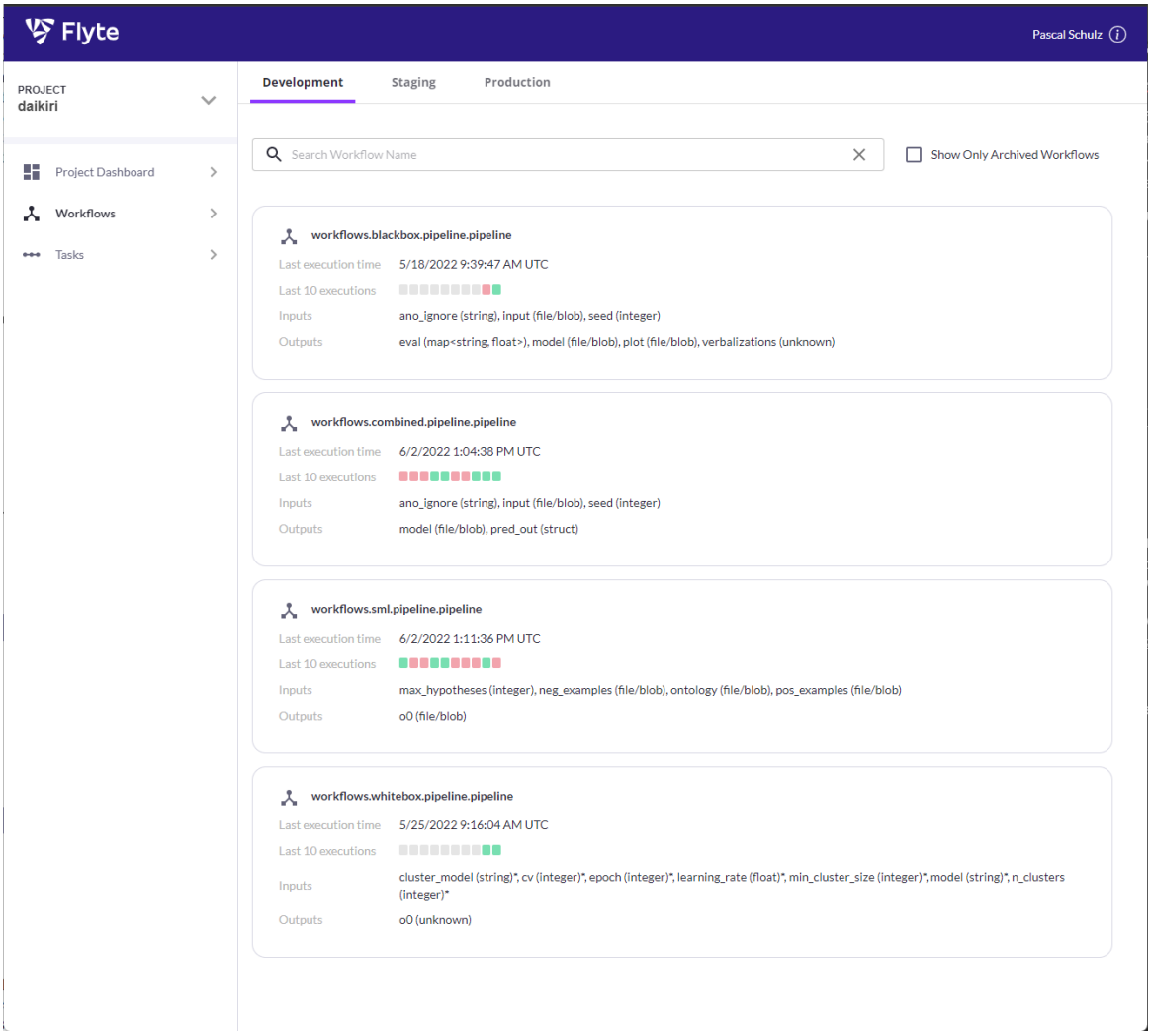

Figure 7: flyte user interface

Reliability

The reliability requirements of the DAIKIRI platform are limited to the failure frequencies of runs. This is especially relevant when training the two machine learning models. To find the optimal combination of hyperparameters it is necessary to train a large number of models. In order to avoid restarting the parameter tuning in between, it was especially important that the DAIKIRI platform runs stable.
To evaluate these requirements a hyperparameter tuning has been simulated. The simulation contains 46 runs in which different hyperparameters were tested. The results of the simulation can be found in the figure 8. Out of 46 runs, there was a single operation that ran into an error.

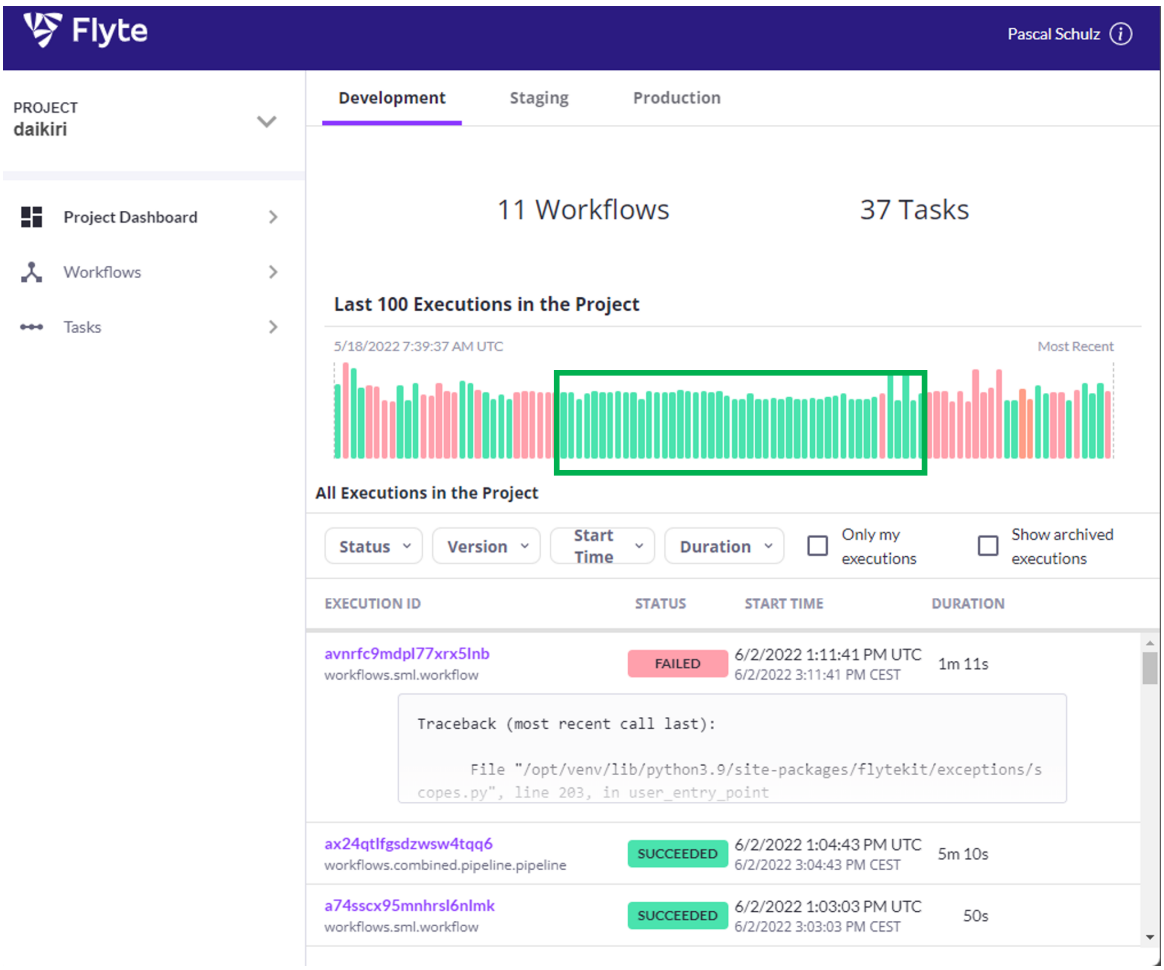

Figure 8: hyperparameter tuning

Supportebility

The last NFR evaluated in this chapter is Supportebility. This requirement is about the handover of the DAIKIRI platform to the project sponsor. In addition, the recoverability of the software in case of software errors is evaluated.
The handover of the software as well as the recoverability of the software are covered by a Git repository. Via the Git repository, the source code of the DAIKIRI platform can easily be handed over to project sponsors.
Git is not only used in this NFR to commit the project, but Git can also be used to restore an older but bug-free version of DAIKIRI. Should an error occur in the software.

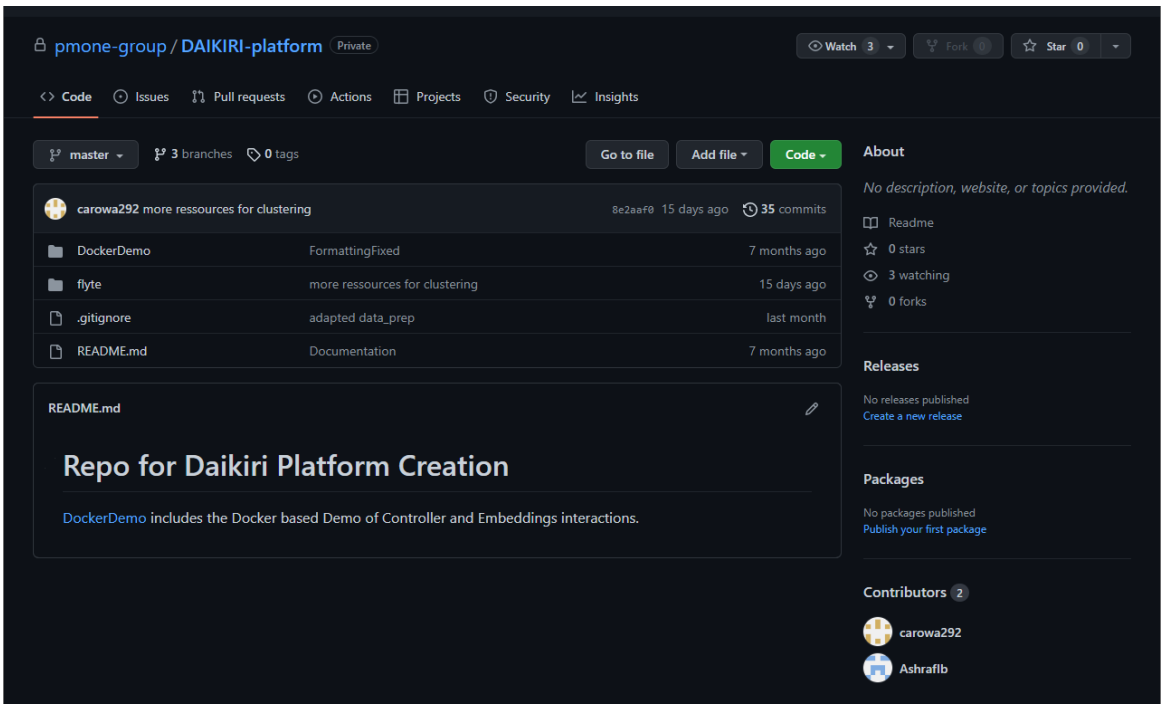

Figure 9: DAIKIRI Github repository

9 Conclusion

In this deliverable we showed how the DAIKIRI components were integrated into the DAIKIRI plattform. We showed difficulties and pitfalls of the integration process. Benefits from using Flyte as workflow engine are also highlighted.

References

Ana I Anton. *Goal identification and refinement in the specification of software-based information systems*. Georgia Institute of Technology, 1997.

Paul C Clements. Early aspects at icse: workshop in aspect-oriented requirements engineering and architecture design. In *Proceedings of the 2006 international workshop on Early aspects at ICSE*, pages 1–2, 2006.

IEEE Computer Society. Software Engineering Standards Committee and IEEE-SA Standards Board. *IEEE recommended practice for software requirements specifications*, volume 830. IEEE, 1998.

ISC Committee et al. Ieee standard glossary of software engineering terminology (ieee std 610.12-1990). los alamitos. *CA IEEE Comput. Soc*, 1990.

Alan M Davis. *Software requirements: objects, functions, and states*. Prentice-Hall, Inc., 1993.

Caglar Demir. Introduction of constraints scalable implementation. 2021. URL https://daikiri-projekt.de/wp-content/uploads/2022/05/DAIKIRI_D2.2.pdf.

Martin Glinz. On non-functional requirements. pages 21–26, 2007.

Ivar Jacobson. *The unified software development process*. Pearson Education India, 1999.

G Kotonya and I Sommerville. Requirements engineering (processes and techniques) john wiley & sons ltd: England. 1998.

John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions on software engineering*, 18(6):483–497, 1992.

Cornelius Ncube. *A requirements engineering method for COTS-based systems development*. PhD thesis, City University London, 2000.

S Robertson and J Robertson. chapter "volere requirements specification template", 1999.

Carolin Walter and Fabian Witter. D4.3: Verfahren zur erklärung von black-box ansätzen. 2022. URL https://https://daikiri-projekt.de/deliverables/.

Hamada Zahera and Stefan Heindorf. D3.3: Semantifizierungsservice für industriedaten. 2021. URL https://daikiri-projekt.de/wp-content/uploads/2022/05/DAIKIRI_D3.2_D3.3.pdf.

Hamada Zahera, Stefan Heindorf, Axel-Cyrille Ngonga Ngomo, Semih Ymusak, Mehmet Buğrahan Duran, and Martin Voigt. Clustering of high-dimensional data. 2020. URL https://daikiri-projekt.de/wp-content/uploads/2022/05/DAIKIRI_D3.1.pdf.