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 5.1
Initial Version of the DAIKIRI Platform

Dissemination Level	Public
Due Date of Deliverable	Month 24, 31/12/2021
Actual Submission Date	Month 30, 11/08/2022
Work Package	WP5 — Platform
Task	T5.1
Type	Report
Approval Status	Final
Version	1.0
Number of Pages	13

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	03/07/2022	First draft created	Sören Brunk
0.1	28/07/2022	Peer reviewed	Carolin Walter
0.2	11/08/2022	Final version submitted	Carolin Walter

Author List

Organization	Name	Contact Information
USU	Sören Brunk	soeren.brunk@usu.com

Executive Summary

In order to integrate the algorithms and components developed in work-packages 2-4, we need a shared platform that allows us to run these components together, reliably and in a seamless fashion. We define several requirements a system must meet to be qualified as a base for such a platform. We look at existing systems and compare them regarding the requirements of DAIKIRI. We present Flyte, the data and execution platform we have chosen to integrate the implementations developed in previous work packages into a single coherent solution. Finally, we show how we have deployed the DAIKIRI platform in the cloud to support our use-cases, enabling reliable parallel execution, scalability and data-lineage among others.

Contents

1 Introduction

In work-packages 2-4, we have seen the development of various algorithms and components to address the goals of the DAIKIRI project. Among them are components for clustering, semantification, black-box machine learning, structured machine learning and verbalization. In order to combine these individual components into a single, coherent solution, we need a shared platform. Such a platform should enable us to seamlessly integrate these components and run them together in a reliable and scalable way.

Goal of this work-package is to work out the platform requirements, decide what the technical basis for our platform should be, provide and initialize the resources for a shared platform, and finally deploy and provide the platform to all project partners. In this deliverable, we describe our approach to solve these challenges. The actual integration of components into the platform is described in the deliverable for work-package 5.2.

In order to choose the best fitting basis for our platform, we describe how we have identified platform requirements in the following sections.

2 Platform Requirements

Based on the project goals, and in close coordination with the implementations of work-packages 2-4, we have identified a set of requirements that should be met by a system to be used as the base for the DAIKIRI platform. These requirements can be further divided into two groups: functional requirements and non-functional requirements.

2.1 Functional Requirements

Based on the goals of this work-package specifically and the project goals in general, we have identified the following functional requirements:

Integrating different algorithms
The algorithms developed in work-packages 2-4 need to be integrated into a single platform in a way that allows them to be run together and to define their data dependencies.

Support different runtimes and programming languages
In DAIKIRI, some algorithms are written in Python while others are written in Java. At minium, the platform should provide support for both languages and to integrate them. Furthermore, the solution should also be generic enough to integrate future components using different technologies.

Multi-user support
The platform should support multiple users working in parallel without interference. Multi-user support is also important for continuous integration and continuous deployment.

Support for machine learning tasks
Since we're dealing with machine learning tasks, and corresponding frameworks, the platform must be compatible with common machine learning frameworks and ideally already provide integrations.

Human-in-the-loop
Some of the tasks in DAIKIRI need to be carried out by humans interactively. For instance,

annotating data in LabEnt[1] [Zahera et al., 2022] is done by humans. The platform should support integrating such human-in-the-loop tasks.

High-level APIs for integration
Integrating the existing components and defining their data dependencies should be simple as possible through high-level APIs

Caching
Since we're dealing with long running algorithms and experiments, failures are unavoidable. Through intelligent caching, the system should help us to avoid unneeded repeated computations due to intermittent system failures or bugs in user code.

Versioning
Due to the inherently experimental nature of machine learning, especially in research it is important to be able to try new things while keeping old versions and data. To support this type of work, the platform should support code, data and executions to be versioned to allow researchers to go back to old versions.

Run in Isolation
The platform should have the capability to run different algorithms or the same algorithm in different versions independently, without interfering with each other.

Parallel execution
The platform should be able to run algorithms that have no data dependencies in parallel, ideally providing necessary resources on demand.

2.2 Non-functional Requirements

Furthermore, we have identified the following non-functional requirements:

Open-Source
The platform code must be open-source under a permissible license. It should not tie its users to a single vendor.

Fast development iteration cycle
The ability for rapid prototyping is especially important when developing new algorithms as part of a research project.

Robust Failure handling
Robustness includes failure handling such as retries and cleanup in case of system errors, as well as good error messages in case of user errors.

Continuous integration and continuous deployment
The platform should provide means to continuously test, integrate and deploy newer versions during development.

Scalability
The platform should provide computational resources at scale on demand and free unneeded resources.

[1] `https://https://github.com/dice-group/LabENT`

Observability The system should provide insights into executions via metrics and UI support.

After establishing and discussing the requirements above, it quickly became clear that a solution supporting all these requirements needs to be very flexible and extensible. Using a **workflow-engine** was suggested as the basis of our platform providing that flexibility.

3 Choosing a Workflow Engine

Based on the requirements defined above, we have identified an initial selection of workflow engines for further comparison. While a wide variety of workflows-engines for different purposes exist, many can be ruled out immediately. In an initial selection, we have only considered candidates that

1. are open-source
2. are under active development and in real use
3. support machine learning use-cases, either explicitly or implicitly due to their flexibility

The initial selection consists of the following candidates:

- Apache Airflow[2]
- Kubeflow[3]
- Metaflow[4]
- Dagster[5]
- Flyte[6]
- Prefect[7]

We then looked how well each of the candidates could fulfill our requirements. After reviewing each candidate, it became clear that Flyte could best meet our requirements. It is the only engine that supports all of our requirements with the exception of human-in-the-loop support. Most importantly, it is the only system that properly supports combining components written in Java and in Python into a single workflow due to its language independent core, different SDKs, as well as its containerization features. Human-in-the-loop support in Flyte was still in development when we chose a workflow-engine, but workarounds exist for the LabEnt integration. Due to that, it was considered an acceptable limitation for the project. In the following, we describe the basic Flyte concepts and how as well as our deployment for DAIKIRI.

[2] https://airflow.apache.org/
[3] https://airflow.apache.org/
[4] https://metaflow.org/
[5] https://dagster.io/
[6] https://flyte.org/
[7] https://www.prefect.io/

4 Flyte

Flyte is "the workflow automation platform for complex, mission-critical data and machine learning processes at scale". Its main goals are to

- Provide a reproducible, incremental, iterative and extensible workflow automation PaaS for any organization.
- Focus on user experience, reliability and correctness.
- Separation of responsibilities between platform teams and user teams.

Flyte provides the following main features, among others:

- Define tasks and arbitrarily complex workflows via Python and Java/Scala SDK.
- Plugins to integrate with popular data-processing and machine-learning libraries such as Spark, Tensorflow, PyTorch and more.
- Run tasks at scale as containers in Kubernetes.
- Strong data typing and validation.
- Reproducible versioned pipelines and outputs, isolated execution.
- Robust failure and retry handling.
- Auditable executions und data lineage.
- Scheduling and notifications.

It is a Linux Foundation Data & AI open-source project, and is used by a growing number of companies worldwide.

4.1 Flyte Core Concepts

The two main primitives used in Flyte are **tasks** and **workflows**.

4.1.1 Tasks

Tasks are the smallest unit of work in Flyte. They define a strongly typed interface with typed inputs and outputs as well as executable code in any of the supported programming language. Tasks map to a backend execution plugin enabling the use of any language, framework and technology supported by a plugin. Examples of supported plugins are Python tasks, containers, SQL queries, and WebAPI calls, among others. Tasks are declarative and versioned.

Listing 1 shows the definition of a simple task using the Python SDK. As shown in the example, tasks are regular Python functions, with the addition of the *@task* decorator. The only requirement for Flyte is that we annotate our functions with types for inputs and outputs, and that these types are compatible with the Flyte engine. Flyte supports most common Python types out of the Box. For custom types not supported, users can provide their own *type transformers*. Due to that, we can

Listing 1 Example of a Simple Flyte Task using the Python SDK

```python
import pandas as pd
import numpy as np

from flytekit import task

@task
def generate_normal_df(n:int, mean: float, sigma: float) -> pd.DataFrame:
    return pd.DataFrame({"numbers": np.random.normal(mean, sigma,size=n)})
```

use almost any existing Python code with very few additional steps as a Flyte task, which provides a lot of flexibility. While we can use almost any Python function as a task, it is recommended to use pure functions, that is, functions without side effects that only rely on their inputs to leverage things like caching. Task decorators can carry additional metadata such as resource requirements, caching or retry handling. Metadata can also be specific to certain task types, for instance, Spark configuration settings.

4.1.2 Workflows

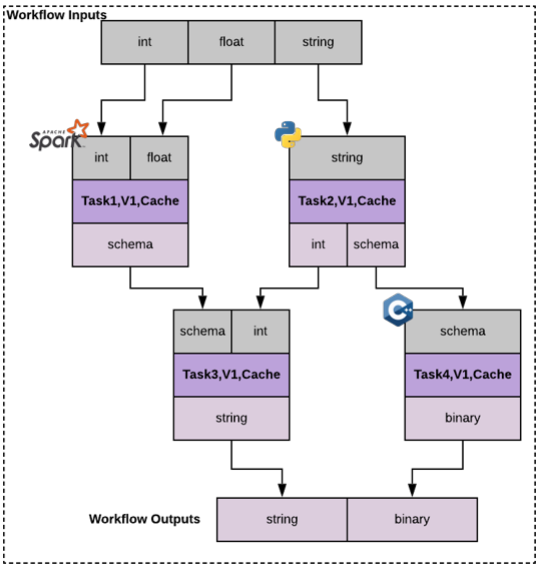

Figure 1: Flyte Workflow Example

Workflows allow composing tasks to more complex execution units by modeling data-flow through tasks. Like tasks, they define a strongly typed interface with typed inputs and outputs. Figure 1 shows a schematic example of a Flyte workflow consisting of multiple tasks and the typed interface of each task as well as the workflow itself.

In order to write workflows in a high-level programming language, Flyte provides domain specific

Listing 2 Example of a Flyte Workflow using the Python SDK

```python
import typing
import pandas as pd
import numpy as np

from flytekit import task, workflow

@task
def generate_normal_df(n:int, mean: float, sigma: float) -> pd.DataFrame:
    return pd.DataFrame({"numbers": np.random.normal(mean, sigma,size=n)})

@task
def compute_stats(df: pd.DataFrame) -> typing.Tuple[float, float]:
    return float(df["numbers"].mean()), float(df["numbers"].std())

@workflow
def wf(n: int = 200, mean: float = 0.0, sigma: float = 1.0) -> typing.Tuple[float, float]:
    return compute_stats(df=generate_normal_df(n=n, mean=mean, sigma=sigma))
```

languages in Python, Java, Scala and JavaScript with Python being the most mature implementation at the time of writing. Listing 2 shows an example workflow, combining two tasks using the Python DSL. Like tasks, workflows are defined as regular python functions with typed inputs and outputs, but decorated with *@workflow* instead. Inside a workflow, tasks or sub-workflows can be combined using regular function composition. Unlike task code though, workflows are converted to a static graph during compile time, which is then executed at runtime by the Flyte engine. Sometimes, a static graph is not enough, because i.e. the execution path depends on runtime inputs. Flyte solves this issue through *dynamic workflows* which provide means to create or expand workflows at runtime. Workflows are declarative by default, versioned and dynamically recursive. An imperative model also exists to make it easier to build high-level SDKs and DSLs on top.

4.2 Development Flow

The typical development workflow with Flyte is as follows:

1. Develop and test tasks and workflows locally.
2. Build a Docker image containing the code as well as all dependencies.
3. Register tasks and workflows with Flyte.
4. Execute workflows. Either via UI or integrate with external sytems via API.
5. Monitor workflow execution. Repeat previous steps in case of user errors.
6. Fetch and use workflow outputs for downstream processing.

Figure 2 shows how the registration process in Flyte looks like.

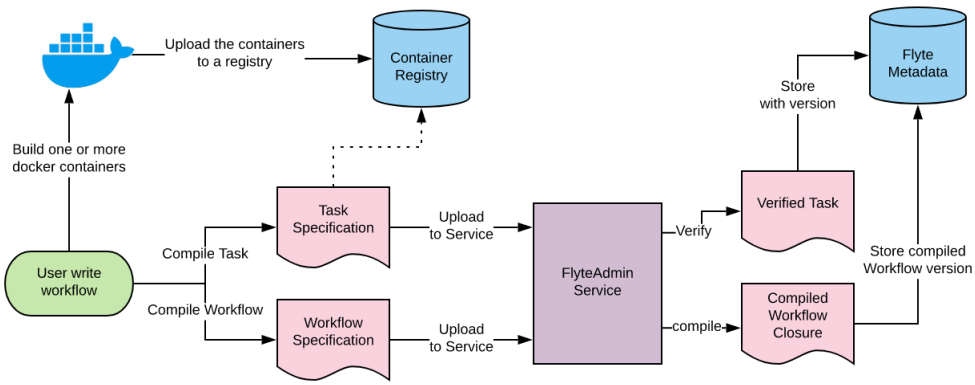

Figure 2: Flyte Registration Process

4.3 Flyte UI

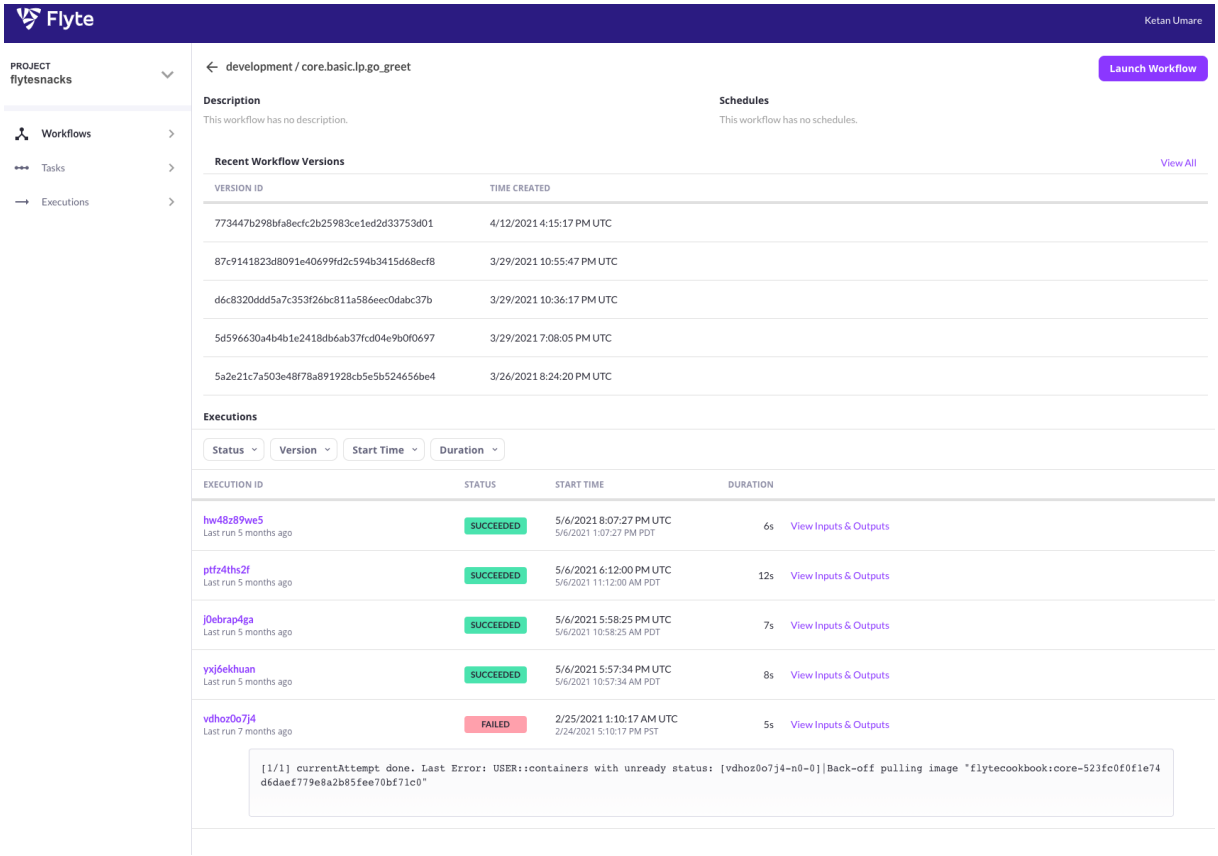

Figure 3: Workflow Executions in Flyte UI

Flyte provides a web-based UI that allows users to view registered tasks and workflows and their interface (input and output types). The UI also enables execution of workflows and monitoring of currently running tasks and workflows, as well as maintenance. In figure 3 for instance, we can see all versions and executions of a specific workflow. Different visualizations provide better visibility into workflows executions, such as workflow node graphs or gantt charts for timeline views. The strongly typed inputs and outputs are used by Flyte UI to automatically generate user interfaces for workflow

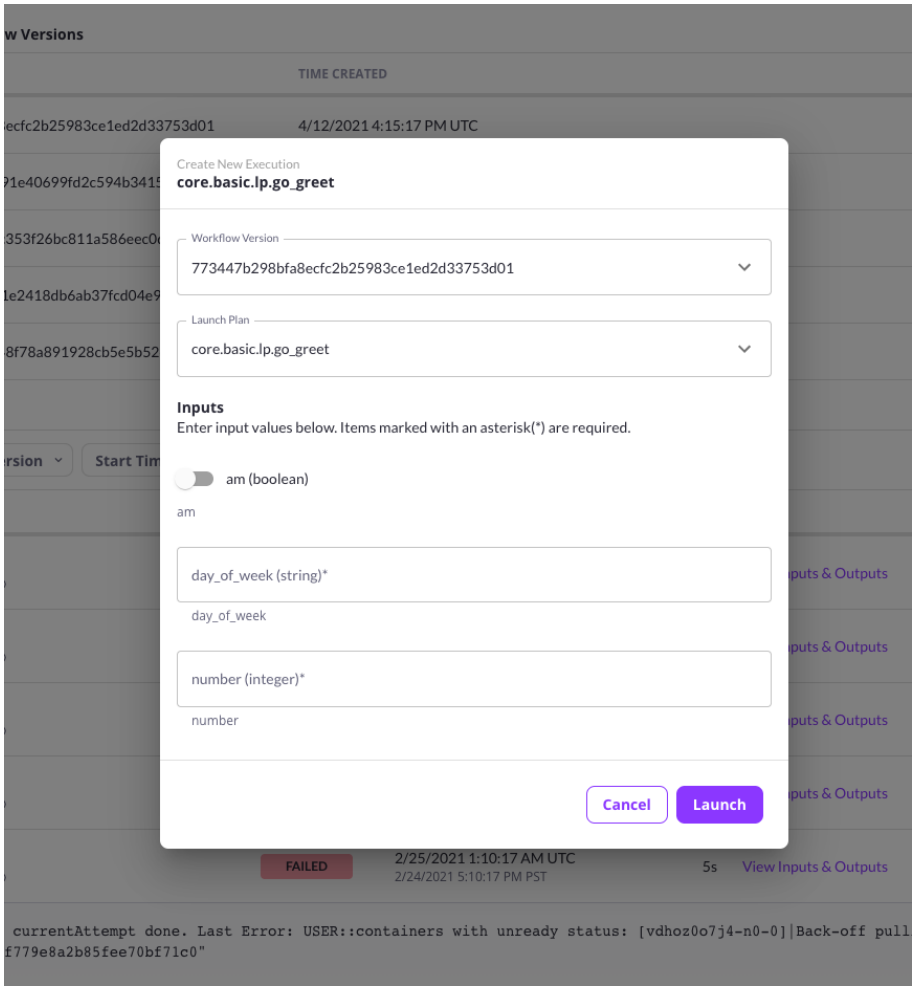

Figure 4: Create new Execution in Flyte UI

execution, matching the workflow input parameters and their types. Figure 4 shows an example with two parameters, *day_ of_ week*, and *number*.

5 Platform Deployment

Flyte supports several modes of deployment. During early development phases and for fast prototyping, it is enough to install one of the Flyte SDKs such as flytekit for Python and run tasks and workflows via local execution.

5.1 Local Execution

During development, tasks and workflows can be run locally, without the platform. Local execution is helpful for fast iteration and to run tests, but has a few restrictions. For instance, it is restricted to a single programming language (the language of the installed SDK) and it is not possible to run SQL tasks. Resources are also restricted to local hardware constraints.

5.2 Sandbox Deployment

In addition to local execution, Flyte also supports a full deployment with its sandbox mode, running in a single docker container. Sandbox is useful for testing registration and running workflows in a real Flyte cluster. It is still constrained to the local hardware though, and single-user only.

5.3 Cloud Deployment

For multi-user support, shared data and computational resources required for machine-learning processing, a dedicated Flyte cluster installation is required. Flyte depends on Kubernetes[8] as the underlying container orchestration platform. For robust operational deployments, it is recommended to use a production-ready Kubernetes cluster, as provided by most cloud providers, or an on premise cluster such as OpenShift.

5.4 DAIKIRI Platform Deployment

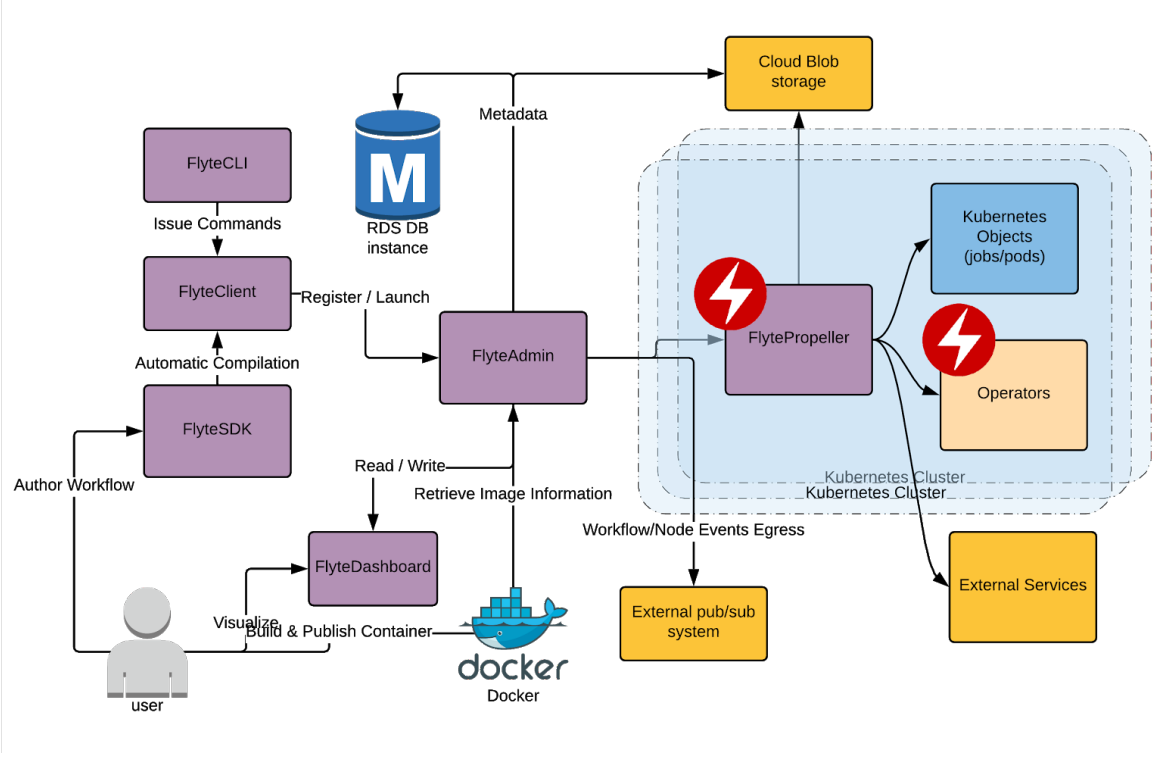

Figure 5: Flyte Architecture Overview

In order to meet the multi-user and scalability requirements of the DAIKIRI platform discussed earlier, we have deployed a dedicated Flyte cluster for DAIKIRI in the cloud, using Google Kubernetes Engine (GKE) as the underlying resource manager. Figure 5 shows the Flyte architecture that corresponds to our deployment. Workflow and Task information is stored in a PostgreSQL database while storage of artifacts is done in Google Cloud Storage. We have added authentication to enable all project partners to securely log into the platform and register and run workflows. We have also configured logging and monitoring capabilities, to add visibility into workflow executions.

[8] https://kubernetes.io/

We have configured the DAIKIRI Flyte instance to use the autoscaling abilities of Kubernetes to scale up on demand for tasks demanding more resources, while keeping resource usage low when not needed. If needed for larger tasks, the platform can provide computational resources up to 416 vCPU and 5.75 TB of memory. The Kubernetes cluster and Flyte is also configured to provide special GPU or TPU accelerators required for deep-learning tasks on demand.

Development of workflows happens in a shared git repository, open to all partners. From there, all partners can register their tasks and workflows and run them in isolation or combine them with existing tasks and workflows.

In addition to Flyte, we have deployed other platform components such as LabEnt on the same infrastructure so they can be used together.

5.5 Open-Source contributions

While deploying Flyte for DAIKIRI, we have worked with the Flyte team and contributed several improvements to Flyte, notably a helm chart to make deployment easier. Over the course of the DAIKIRI project, we have also contributed bug fixes and smaller improvements.

6 Conclusion

In this deliverable, we have presented our shared platform for DAIKIRI. We have discussed the requirements to be met by such a shared platform, the need for a workflow engine as basis for our platform as well a the selection process that led us to choose Flyte. We have presented Flyte, its core concepts, its architecture and how the development process looks like when using Flyte. Finally, we have shown how our scalable, multi-user deployment looks like to support all project partners in integrating their solutions into a common platform.

References

Hamada M Zahera, Stefan Heindorf, Stefan Balke, Jonas Haupt, Martin Voigt, Carolin Walter, Fabian Witter, and Axel-Cyrille Ngonga Ngomo. Tab2onto: Unsupervised semantification with knowledge graph embeddings. In *European Semantic Web Conference*, pages 47–51. Springer, 2022.