Cross Project Review - DataCloud

Christian Bauer

June 2, 2023

Contents

Listings

1 Introduction

In this report, I will elaborate on the process of creating a REST server for the DataCloud project that was completed during the project semester work. The topic I have chosen to focus on is creating a REST server for the DataCloud (see 3) research project and more specifically the ADAPIPE component (see 3.1). The topic has relevance to current trends and issues such as using RESTful services (see 4.1) to interact with numerous DataCloud components in a distributed infrastructure. The communication between those DataCloud components is required to be handled in a secure manner, for which the tool KeyCloak (see 4.4) is used. KeyCloak enables the DataCloud components to interact only with authenticated resources or users. To ensure that the REST server is extendable and maintainable, multiple code quality measures (see 7) are being considered. Finally, future work is mentioned that is not included in the current state of the project (see 8).

The second part Reflection and Experience of this report contains the gained experience while working on this project.

2 Goal of the Project

The goal of this project was to implement a server that is capable of handling incoming messages by processing and forwarding them to other services of the DataCloud project by providing a REST API for communication. Additionally, the server has to be capable of handling incoming monitoring data by distributed resources that are used for computing tasks. Finally, a requirement is for the server to be deployable inside a Kubernetes cluster, which was achieved by containerising it to a Docker container.

3 DataCloud

Nowadays, the amount of data that is generated and collected is bigger than ever before. Big Data leads to new challenges involving processing and managing this massive amount of information and turning it into valuable insights. The DataCloud toolbox offers a comprehensible solution for discovering, simulating, deploying and adapting Big Data pipelines. The toolbox is designed to also be executed on infrastructures with heterogeneous and untrusted resources.

> DataCloud delivers a toolbox of new languages, methods, infrastructures, and prototypes for discovering, simulating, deploying, and adapting Big Data pipelines on heterogeneous and untrusted resources. DataCloud separates the design from the run-time aspects of Big Data pipeline deployment, empowering domain experts to take an active part in their definitions [1].

The separation of the design from the run-time aspects of deployment, DataCloud empowers its users to create efficient and effective Big Data solutions. The architecture of DataCloud is shown in figure 1.

3.1 DataCloud Pipeline Components

The following sections describe the different DataCloud components that are part of the toolbox. Each of the components is also seen in figure 1.

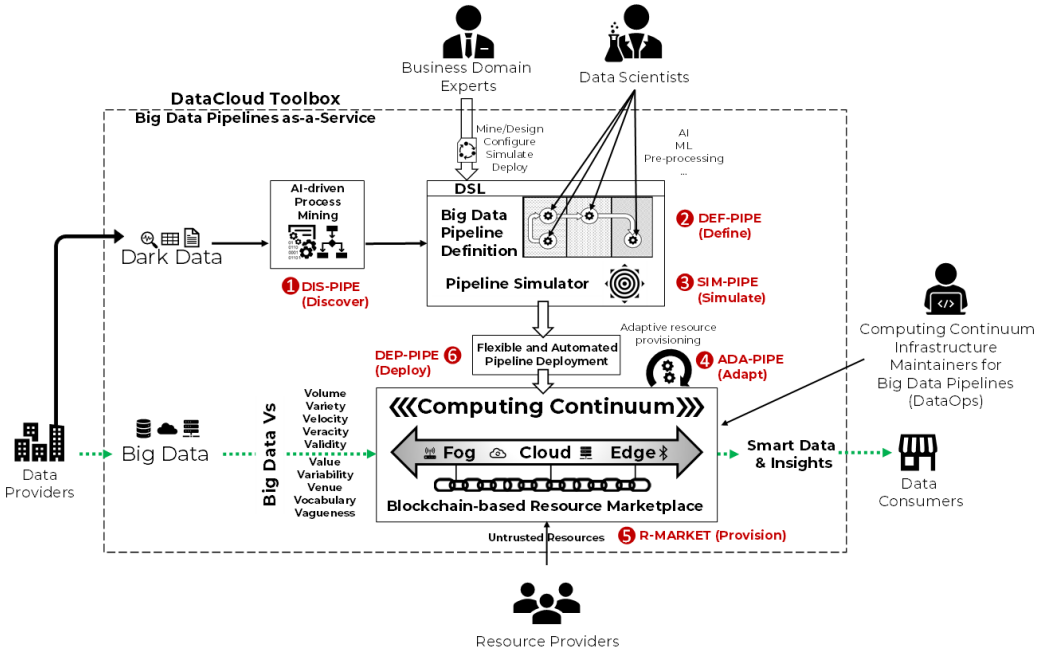

Figure 1: DataCloud Toolbox Overview [2]

DIS-PIPE

DIS-PIPE (Discover pipeline) [3] uses a set of process mining techniques and AI algorithms to analyse and learn the structure of Big Data pipelines. This is done by extracting, processing and interpreting the vast amount of event data that is collected by multiple heterogeneous data sources.

DEF-PIPE

DEF-PIPE (Definition pipeline) [4] uses domain-specific language (DSL) [5] to provide a visual design for the implementation of Big Data pipelines. This includes support for storing and loading the pipeline definitions and displaying them in a UI. For this, the pipeline structure is declared by domain experts.

SIM-PIPE

The SIM-PIPE (Simulation pipeline) [6] module generates and simulates a deployment configuration that takes hardware requirements into account and also includes additional middleware information that is required. The generated result is then used for the final deployment.

ADA-PIPE

ADA-PIPE (Adaptation pipeline) [7] is the module that we are working on. ADA-PIPE provides a data-aware algorithm for smart and adaptable provisioning of resources and services. The tool allows resource reconfiguration for improved computational performance and interoperability by monitoring and analysis of diverse resources.

The monitoring of resources and the mapped pipeline tasks allow analysing of the gathered data to be able to make more accurate predictions in the future based on machine learning algorithms that are specifically made for the prediction of time-series or sequential data.

R-MARKET

R-MARKET (Resource-MARKET) [8] deploys a decentralized resource network that is based on a hybrid of permissioned and permissionless blockchain. This blockchain federates a set of heterogeneous resources from multiple providers spread across the Computing Continuum.

DEP-PIPE

DEP-PIPE (Deploy-PIPE) [9] provides elastic and scalable deployment and orchestration of Big Data pipelines while addressing run-time aspects. The module also features real-time event detection and automated decision-making for automated deployment and orchestration.

4 Server

This section describes the used third-party tools and concepts used to implement the server of ADA-PIPE. It was agreed upon by all developers to use REST for the communication between the different pipeline components of DataCloud with specific interfaces to send messages.

4.1 REST

REST [10] (short for Representational State Transfer) was created for building web services and provides a set of constraints and properties that are used to make web services *RESTful*. RESTful web services are built to be scalable, flexible and maintainable. REST is based on the HTTP protocol and relies on client-server communication, where a client first sends a request and the server sends back a response. The response contains a representation of the requested resource, which is JSON for the ADA-PIPE module.

4.2 Flask

Flask [11] is a Python framework for creating web applications. The main benefit of Flask compared to other web application frameworks for Python like Django is that it is easy and quick to get started, yet scaling up to complex web applications is easily possible and most important a lightweight framework. Flask is often used for small to medium-sized projects and is known for its simplicity and ease of use.

In ADA-PIPE, Flask is used to implement and expose the REST server to other components (see 1) of DataCloud.

4.3 Required REST Interfaces for Collaboration

4.3.1 Pipelines

Endpoint `/pipelines`

REST Methods `GET, POST`

Description The pipelines endpoint is used to get a JSON file that contains a list of all pipelines that are currently running in the DataCloud system with the `GET` method. To upload a new pipeline and its tasks, the `POST` method can be used. The process after posting the pipeline is described in section 4.5.2. Once a pipeline is successfully uploaded using the `POST` method, it can also be retrieved with the `GET` method as long as the pipeline is running inside the system.

4.3.2 Requirements

Endpoint `/requirements/<pipeline_id>`

REST Methods `GET`

Description This endpoint returns the requirements of each job of the pipeline that is specified with its `pipeline_id`. The requirements include parameters such as the maximum resource limits of a task that are not to be exceeded or the docker image and its location. Also included in the requirements are the dependencies of each task to tasks that have to be executed beforehand.

4.3.3 Resources

Endpoint `/resources`

REST Methods `GET`

Description This endpoint returns a list of all resource providers and their worker pools to the requesting client. The worker pools contain metadata, such as their name, the name of each worker, the number of workers and their resource capabilities. The resource capabilities include values such as the number of virtual CPU cores, and the memory and storage capacity.

4.3.4 Schedules

Endpoint `/schedules/<pipeline_id>`

REST Methods `GET`

Description This endpoint returns a schedule of tasks of a pipeline that corresponds to the identifier `pipeline_id`. The tasks are returned in a list that is ordered based on the order they should be deployed on the resources. The tasks contain metadata such as the resource they need to be deployed on, the resource limits that are not to be exceeded for the task as well as the adapted resource utilisation that was predicted via machine learning.

4.4 KeyCloak Authentication

KeyCloak [12] is an open-source identity and access management tool. DataCloud has decided to use this tool to ensure authenticated communication between all services as well as its users. In DataCloud, the possibly sensible data requires to be handled securely, making KeyCloak a feasible tool for user federation and access control. KeyCloak is based on strong standard protocols and provides support for many common communication protocols such as OpenID Connect and OAuth 2.0.

In DataCloud, the responsibility of providing authentication tokens is done by a KeyCloak server instance. In order to be able to communicate with other DataCloud microservices, ADA-PIPE first has to retrieve a KeyCloak authentication token from the server instance. This authentication token is similar to a session key often found for web services and is only valid for a predefined time frame. This authentication token must be sent with every request to other microservices and every microservice inside of DataCloud is required to verify the validity of the sent token.

In ADA-PIPE, before acquiring the authentication token for our service, an instance of the *KeycloakOpenID* has to be created with the proper credentials, such as *server URL, client id, realm name* and *client secret key*. Since especially the client's secret key has to be concealed, a function is used to load the key into memory so that it can not be easily read.

4.5 Processing of Incoming Data

In this section, the processing of the incoming data will be discussed. The incoming data in this scenario is a Big Data task pipeline that should be mapped and then deployed to the resources in the Computing Continuum. The Big Data task pipeline is defined in a custom DSL [13] that is sent from DEF-PIPE to ADA-PIPE and is then preprocessed before analysing and scheduling the tasks.

4.5.1 Requirement File Structure

As mentioned, the incoming file structure is a custom DSL. It is structured similarly to a JSON file with a few distinctions, but the key-value pattern and the nested value sections are similar. At the root level, the keyword `Pipeline` is always followed by the name of the pipeline. At the next nesting level, the keywords are as follows:

- `communicationMedium`, which denotes what type of service the pipeline tasks are, for example, that they consist of a set of web service tasks.
- `environmentParameters`, which is a set of parameters that are required to interact with the environment such as Kubernetes and other services.
- `steps`, which is the task pipeline itself that consists of all the tasks. This also includes the dependencies of tasks to other tasks and is usually shown by the order in the tasks mentioned in the pipeline as well as a nested keyword `previous` that denotes all services that have to be running before the task this keyword resides in. Each step also describes the hardware requirements and how scalable a task has to be.

4.5.2 Requirement File Preprocessing

Since the incoming data is provided in a DSL scheme, and the next DataCloud module DEP-PIPE is responsible for the deployment of the tasks onto resources expects the file format in a JSON scheme, we parse the incoming data and convert the file as a JSON file. For this, we receive the data via REST and extract the message body and parse it. After parsing the JSON message body, the extracted data is stored in Python objects of the classes `JobDataContainer` and `PipelineDataContainer`. The `PipelineDataContainer` class is used to store the data of the entire incoming pipeline, its structure, order and its metadata. It holds the jobs/tasks as `JobDataContainer` objects that each holds the metadata and properties of the jobs. The jobs in the pipeline are stored in a list, that is sorted ascending based on the dependencies upon other tasks.

5 Monitoring

In this section, the approach and tools used to monitor the system are described. First, the monitoring tool Netdata is described that is used to monitor the computing resources of the infrastructure. Next, the tool Prometheus is mentioned, which is used to scrape monitoring data of all resources that have Netdata installed.

5.1 Netdata

Netdata [14] is an open-source tool that collects real-time metrics, including CPU usage, disk activity, bandwidth usage and furthermore. The reason we chose Netdata for monitoring is that it is a lightweight tool mostly written in C, Python and Javascript and it requires minimal resources, which is necessary when monitoring edge devices. One of its major features is that it runs without interrupting any of the applications running on the same device. This is achieved by only using idle CPU cycles while running.

Netdata provides an in-browser dashboard to analyse each metric in real-time with the help of visual representation.

Netdata has a vast amount of support for other tools in order to gather data. One of the supported tools is Prometheus, which is used to scrape the monitored data from all resources that have Netdata installed.

5.2 Prometheus

Prometheus [15] is an open-source application used for monitoring and alerting events and is designed to run across various platforms in a scalable and easily deployable manner. Same as Netdata it records in real-time, and stores the gathered metrics in a time series database by using a *HTTP pull model*. It also allows real-time alerting via a rule-defining configuration and also has a flexible query language called *PromQL*, that enables the retrieval and processing of the gathered data. Prometheus has great integration with other tools such as Netdata. In the project, it is used inside a monitoring master node that is continuously scraping all resources that have Netdata installed for monitoring data. In case a predefined rule is broken at a monitored resource, such as CPU over-allocation, Prometheus triggers an event that notifies ADA-PIPE in order to be able to take measurements.

6 Project Planning

The internal project planning was done with Skype Business [16] and Trello [17]. Skype is used as a video communication tool to discuss the current progress of the project and the future steps to take. Trello is a project management tool that uses boards and cards to organize tasks among all members. We organized our Trello board similar to a Kanban board and assigned the tasks according to the previous discussions on Skype.

For project steps that are important for other stakeholders of DataCloud, the ADA-PIPE GitHub project board is used similarly to the Trello board. This is done to ensure others see the current tasks of the team members as well as the ones that are already done or are in the backlog. Also, they can add tasks to the project board, such as bugs to be fixed or missing API functionality.

7 Code Quality

7.1 Code Refactoring

Code refactoring of pre-existing code was necessary in order to be able to use the components with other modules of the project. The reason for this is that many functionalities were written in Python scripts that were not written to be included with other modules at that stage.

> Code refactoring is defined as the process of restructuring computer code without changing or adding to its external behavior and functionality [18].

Therefore, the functionality of the scripts was analysed and then put in separate methods. Hard-coded sections of the scripts were put into variables or abstracted were possible. Since most of the hardcoded sections were specific file paths that were manually changed depending on which functionality was tested, those hardcoded sections were exported to configuration YAML files to be able to add further configuration files to the project.

7.2 Type Hinting and Linting

Python supports *type hinting*[1] since version 3.5. This enables type annotation for functions, variables and other components. Note that type hinting is not enforced and can only be used to declare a type of a code section, but even then, it is not checked nor analysed by the Python runtime. Yet, correctly using type hinting enhances the code readability and even makes the method documentation section of each parameter and its type unnecessary in trivial cases. Also, it enables other users to easily use the methods and classes since all its parameters and fields are annotated with the corresponding type.

As an example of the usage of type hinting in the project, the `PipelineState` class constructor is shown in the following listing:

```
class PipelineState():

    def __init__(
                self,
                initial_pipeline_state: Dict[str, PipelineDataContainer] = None,
                add_dummy_data: bool = False
                ) -> None:
        if initial_pipeline_state is None:
            initial_pipeline_state = {}
        self.__pipelines: Dict[str, PipelineDataContainer] =
    initial_pipeline_state

        if add_dummy_data is True:
            self.__init_with_dummy_data()
```

Listing 1: PipelineState Constructor

A major benefit of using type hinting not mentioned above is that *Linters* use the provided type hints to enable type checks before runtime.

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

8 Future Work

In this section, the future work of sections mentioned that are planned to be done in the ADAPIPE project but were not done thus far, or only to some degree.

8.1 Extensive Documentation

Documentation is available to some extent in the current version of the project, yet lacks documentation for many sections, such as an overview of the overall system, and the installation guide for new users. For this, we will use GitHub wiki functionality that enables us to provide documentation for the project next to the GitHub repository.

8.2 Swagger

Swagger [19] is an open-source software framework used for designing, building, and documenting RESTful APIs. It provides functionality to describe the structure of APIs, including the inputs and outputs. Additionally, it generates interactive documentation and enables APIs to be more readable and accessible, making it easier to understand how to interact with an API. The use of Swagger has become a standard for documenting and describing RESTful APIs, making it easier for developers to build and consume these APIs. While Swagger is already in use, it only covers parts of the REST API and project. It is planned to further increase the usage of Swagger to document the API endpoints.

8.3 Server Load Testing

Server load testing is the process of evaluating the performance of a server under a simulated heavy load. This simulation involves a high number of users or requests to the server so the performance under stress can be analysed. Server load testing has the purpose of identifying performance bottlenecks or limitations before deploying the server to a production environment. The results of server load testing can be used to optimize the server's configuration, identify and fix any scalability issues, and improve the overall performance of the server.

9 Reflection and Experience

This section contains the reflections and experiences gathered throughout the project. They are split into technical, organisational and personal as best as possible, though they might intersect to some degree.

9.1 Technical

9.1.1 FastAPI instead of Flask

Because I had previous experience with *Python Flask* I chose to use that framework to implement the API endpoints for the REST server. A major benefit I thought Flask had over competitors such as Django is its lightweight dependencies and size. The desire to build a REST server as small as possible with no extra *bloat* kept me from looking elsewhere for solutions. While the result turned out fine, while implementing the server I quickly saw the shortcomings of using a framework that does not add any (or few) functionalities not defined by the developer. A problem that soon took some time to solve was that I had to generate a JSON to Python (and vice versa) parser for each API endpoint, simply because we consumed messages as JSON but worked with Python objects internally. And each change in the API object structure did take up additional time to keep the server functional. Another feature that was requested at approximately three-quarters of the project duration was to enable *Swagger* [19] documentation to the API endpoints. Given the more complex nature of the server and the cooperation with other teams, the *minimalism* of Flask slowed the development of features down, since every addition had to be done with some extra steps.

I then encountered the framework *FastAPI* [20] during my spare time by watching YouTube videos. I quickly realised that the problems that were encountered with Flask already had clever solutions in FastAPI. The conversion from JSON to Python would have been done automatically by providing a Python class template that defines all fields that are expected by an API endpoint. This also would have omitted error-prone parsing and conversion of fields. Similarly, the conversion from Python classes to JSON is also done automatically by FastAPI. Also, Swagger documentation is automatically done by FastAPI for each API endpoint based on the defined Python classes, so there is the benefit of having clean Python classes, that are used to convert from and to JSON and also create Swagger documentation automatically. On top of not having to worry about those things since they are provided out of the box, FastAPI does have many features that enable developers to develop features for a server fast and securely and cleanly. At least that's what the YouTube video made FastAPI seem to be, the right tool for the job.

9.1.2 Python is Object-Oriented but with Extra Steps

The programming language Python was mainly used for this project and whenever it seemed beneficial, an object-oriented approach was used. This includes using libraries such as `dataclass` that provide a decorator for Python classes, and with it, adds functionality such as automatically creating a constructor. This approach has the advantage of making data classes smaller in size, therefore making them more readable as well. Python data classes strongly rely on *type hinting*, therefore the type of a class variable has to be declared. While it is not possible to ensure the type in Python, it still has the advantage of providing information regarding the expected data type and many *linters* exist that do type checking and warn about using the wrong data type.

9.2 Organisational

9.2.1 Priorities, Priorities, Priorities

Setting the right priorities isn't easy. Even harder is to extract which tasks do have a higher priority based on meetings and communicating with colleagues. This resulted in a colleague mentioning a task that needs to be completed but had no urgency at the stage of the project, which I then started working on. This task did consume about a working week and given the supposed urgency of it, other tasks could not be finished in time. Only then it became apparent that there was a miscommunication happening and the task did not have any urgency at all.

This did result in me adapting my communication style and directly asking how urgent a mentioned task is, and if I have to complete two competing tasks, which has the higher priority. Doing so also has the benefit of discussing the benefits of completing which tasks first based on the needs of colleagues as well, giving oneself a clearer picture of the responsibilities and tasks of other colleagues, since they often mention why some tasks are required to be completed earlier.

9.2.2 Have fixed Meeting Time Windows

In some open-ended meetings, there is a stage where no additional insights will be gained by continuing it. Therefore it is beneficial to have fixed time windows for meetings that will not, or only slightly be extended. In a company I worked at, there was also a person moderating the conversation, which have the benefit of staying on topic and ending meetings that did not produce further insights. Of course, having a meeting moderator is not possible in most meetings, and communicating that the meeting served its purpose and should be discontinued requires some tact since it might get misunderstood as being rude.

9.3 Personal

9.3.1 Not Invented Here (NIH) Syndrome

Various online sources describe the *Not Invented Here (NIH)* syndrome. But it would be against the syndrome itself to use them instead of explaining it myself. The NIH syndrome describes the tendency to refuse the usage, buying etc, of products and tools from external sources. In development, this refers to the belief that building the functionality is inherently better suited and faster to implement for the project than the already existing tools.

In this project, I fell for this belief when searching for a solution for monitoring the distributed resources used for the computation of tasks. The idea was to implement a monitoring client that tracks the resource and sends only the required amount of information to the REST server. The major benefit of this approach was the very low message size that had to be sent to the server, which in the case of hundreds of resources will make a huge difference. While implementing the client, it became apparent that developing the monitoring tool would require plenty more time to be completed only to have the same functionality that other monitoring tools already provide. But this did not even take error handling and exceptions into account, which are likely to occur in a heterogeneous and large-scale system. And in the case of this project, not focusing more time on the monitoring part and relying on proven tools did allow me to progress in other directions.

9.3.2 Less Is More

While implementing the features for the REST server I often found myself implementing methods and functionality that I *might* use in the future. This resulted in a scattering of unused methods

and functionality over multiple files and classes just to have that *better-be-save* feature, which for almost all cases did turn out to not be used once. Most of those functions did not take up much of my time to implement, yet continuing this behaviour of course does add up over time. Another problem that started to evolve with adding more and more of those methods was that the files and their contents naturally became larger, which led to taking longer times to comprehend the contents of a file and sometimes even to the question *"Why does this function exist?"*.

Having to encounter functions that were not even used once over and over again has led to the discovery that *better be safe than sorry* is not always applicable. Now I try to think twice about if I will actually use a method before implementing it, and force myself to (almost) only add methods when I refactor code segments once their functionality is clear.

9.3.3 It has to be Good, not Perfect

In general, I tend to do more work than was necessary to complete a task, always trying to do it perfectly. This is apparent in the above sections of this chapter. Perfectionism in my opinion does have its time and place, but in this project, it sometimes was misplaced when focusing on parts that were not that important, hence did not have a high priority. Throughout this project, I gradually did force myself to apply the $80 : 20$ rule, making sure the required functionality is provided and only then when necessary reworked some parts in order to improve the usability.

9.3.4 It Turned Out Fine

One very positive reflection section was kept as a final note. I was worried about having to implement a REST server for multiple independent companies/services and how it will turn out and reflect on my abilities. The other people involved do have more experience either in software development or research than me, so it was *natural* for me to feel the need to show that I am also capable to further the progress of the project and not hinder it by inexperience. This is one of the reasons why the above-mentioned experiences and issues did manifest more than they did for smaller projects done for University courses.

After finalising my part of the project and looking back I was able to provide good insights in some stages regarding secure communication and later implementation of a working REST server. While I now would have used other frameworks and tools for some parts, it did turn out fine. And wanting to improve on a part previously thought to be a good solution also shows that some lessons were learned during the project.

References

[1] R. Dumitru, "About the Project — Project — DataCloud Project." https://datacloudproject.eu/project/about-the-project.

[2] R. Dumitru, "DataCloud Toolbox." https://datacloud-project.github.io/toolbox/.

[3] S. Agostinelli, J. Rossi, A. Marrella, and D. Benvenuti, "DIS-PIPE." DataCloud, Feb. 2023. https://github.com/DataCloud-project/DIS-PIPE.

[4] V. Mitrovic, A. Layegh, B. Elvesæter, and S. Tahmasebi, "DEF-PIPE Frontend." DataCloud, Feb. 2022. https://github.com/DataCloud-project/DEF-PIPE-Frontend.

[5] JetBrains, "What are Domain-Specific Languages (DSL) — MPS by JetBrains." https://www.jetbrains.com/mps/concepts/domain-specific-languages/.

[6] N. Nikolov, A. Pultier, A. Thomas, and B. Elvesæter, "SIM-PIPE." https://github.com/DataCloud-project/SIM-PIPE.

[7] N. Mehran and C. Bauer, "ADA-PIPE." DataCloud, Feb. 2023. https://github.com/DataCloud-project/ADA-PIPE.

[8] S. Sengupta and A. Djari, "R-MARKET." https://github.com/DataCloud-project/R-MARKET.

[9] G. Ledakis, I. Plakas, B. Elvesæter, and K. Theodosiou, "DEP-PIPE." https://github.com/DataCloud-project/DEP-PIPE-Pipeline-Deployment-Controller.

[10] RedHat, "What is a REST API?." https://www.redhat.com/en/topics/api/what-is-a-rest-api.

[11] "Welcome to Flask — Flask Documentation (2.2.x)." https://flask.palletsprojects.com/en/2.2.x/.

[12] KeyCloak, "Documentation - Keycloak." https://www.keycloak.org/documentation.

[13] N. Mehran and C. Bauer, "DSL-DEF-PIPE Example." DataCloud, Feb. 2023. https://github.com/DataCloud-project/ADA-PIPE/blob/04ed7959883afbf5ba536a5688d7e61e30ff4be4/ImportFrom-DEF-PIPE/tellu.dsl.

[14] Netdata, "Getting started — Learn Netdata." https://learn.netdata.cloud/docs/getting-started/, Mar. 2023.

[15] Prometheus, "Overview — Prometheus." https://prometheus.io/docs/introduction/overview/.

[16] Microsoft, "Skype — Stay connected with free video calls worldwide." https://www.skype.com/en/.

[17] Atlassian, "Manage Your Team's Projects From Anywhere — Trello." https://trello.com/.

[18] S. Watts and C. Kidd, "What is Code Refactoring? How Refactoring Resolves Technical Debt." https://www.bmc.com/blogs/code-refactoring-explained/, Mar. 2018.

[19] Swagger, "API Documentation & Design Tools for Teams — Swagger." https://swagger.io/.

[20] "FastAPI." https://fastapi.tiangolo.com/lo/.