# Design and Implementation of a web-based Editor for Data Pipelines

BACHELOR THESIS

**Maximilian Ackermann**

Submitted on 13 November 2023

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Philip Heltweg M.Sc.
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

# Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 13 November 2023

# License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see https://creativecommons.org/licenses/by/4.0/

Erlangen, 13 November 2023

# Abstract

This goal for this thesis is to contribute to the JValue Hub. The thesis is covering the design and development of the new ad-hoc runtime feature for the JValue Pipeline Editor. During the process, pipeline and code editor market were assessed and a new layout for the JValue editor was created. The new added functionality of an ad-hoc runtime was added to the pipeline editor. Major decisions that were taken during the implementation of this feature are presented and evaluated.

# Contents


**1 Introduction** **1**
- 1.1 JValue Project . . . 2
  - 1.1.1 Jayvee . . . 2
  - 1.1.2 JValue Hub . . . 2

**2 Common interfaces of editors** **5**
- 2.1 Editor categories . . . 5
  - 2.1.1 Pipeline editors . . . 6
  - 2.1.2 Code editors . . . 6
- 2.2 Common layout for code editors . . . 7

**3 Scope and requirements** **9**
- 3.1 User story . . . 10
- 3.2 Scope definition . . . 10
  - 3.2.1 In scope . . . 10
  - 3.2.2 Out of scope . . . 10
- 3.3 Requirements . . . 10
- 3.4 Non-functional requirements . . . 11
- 3.5 Functional requirements . . . 12
  - 3.5.1 Ad-hoc runtime . . . 13
  - 3.5.2 Runtime information . . . 13

**4 JValue Hub Architecture** **15**
- 4.1 Hub-web . . . 15
- 4.2 Hub-backend . . . 15
  - 4.2.1 Project, repository and versioning . . . 16
- 4.3 Pipeline service . . . 17
- 4.4 Runtime service . . . 17
- 4.5 File service . . . 17
- 4.6 Important frameworks and libraries . . . 17
  - 4.6.1 React and Redux . . . 17
  - 4.6.2 NestJS and typeORM . . . 18

4.7 Communication . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
4.8 Data storage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

**5 development and implementation 21**
5.1 Layout and design . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
5.1.1 Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
5.1.2 States of the pipeline editor . . . . . . . . . . . . . . . . . . 22
5.2 Implementation into existing architecture . . . . . . . . . . . . . . . 25
5.2.1 Additional libraries and framesworks . . . . . . . . . . . . 26
5.2.2 Runtime environment . . . . . . . . . . . . . . . . . . . . . 26
5.2.3 New architecture . . . . . . . . . . . . . . . . . . . . . . . 26
5.2.4 Repository, version and pipeline relation . . . . . . . . . . 27
5.2.5 Editor page . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.2.6 Sink storage . . . . . . . . . . . . . . . . . . . . . . . . . . 30

**6 Evaluation of requirements 33**
6.1 Demo and code review . . . . . . . . . . . . . . . . . . . . . . . . . 33
6.2 End-to-End-Testing . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
6.3 Frontend testing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
6.4 Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

**7 Conclusion and outlook 37**

**Appendices 39**
A Test scenarios . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
A.1 Testing existing functionality . . . . . . . . . . . . . . . . . 41
A.2 Testing new functionality . . . . . . . . . . . . . . . . . . . 41
B Editor overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

**References 47**

# List of Figures


4.1 Repository pipeline relation - fact check!!! . . . . . . . . . . . . . 16
4.2 Simplified class diagram . . . . . . . . . . . . . . . . . . . . . . . . 18
4.3 Pipeline API . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
4.4 Communication between services of a single pipeline run . . . . . 20

5.1 Mock layout with console . . . . . . . . . . . . . . . . . . . . . . . 21
5.4 State for the pipeline editor page . . . . . . . . . . . . . . . . . . . 22
5.2 Final design editor page . . . . . . . . . . . . . . . . . . . . . . . . 23
5.3 Final design runtime information . . . . . . . . . . . . . . . . . . . 24
5.5 Mock up: Editor page state S1 and S4 . . . . . . . . . . . . . . . . 24
5.6 Mock up: Editor page state S1 and S5 . . . . . . . . . . . . . . . . 24
5.7 Mock up: Editor page state S3 and S4 . . . . . . . . . . . . . . . . 24
5.8 Mock up: Editor page state S2 and S6 . . . . . . . . . . . . . . . . 24
5.9 New simplified class diagram . . . . . . . . . . . . . . . . . . . . . 25
5.10 Implementation for pipelines module including adhocPipelines . . 25
5.11 Option 1-4 for pipeline information entity integration . . . . . . . 28
5.12 Repository ad-hoc pipeline relation . . . . . . . . . . . . . . . . . . 29
5.13 Pipeline inheritance . . . . . . . . . . . . . . . . . . . . . . . . . . 30

# List of Tables

3.1 Non-functional requirements . . . . . . . . . . . . . . . . . . . . . . . . 11
3.2 Functional requirements . . . . . . . . . . . . . . . . . . . . . . . . . 12

6.1 Functional requirements . . . . . . . . . . . . . . . . . . . . . . . . . 34

1 Overview of test scenarios for existing features . . . . . . . . . . . . 41
2 Overview of test scenarios for new features . . . . . . . . . . . . . . 41
3 IDE overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
4 ODE overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

# Acronyms

**ETL** Extract, Transform and Load

**E2E** End to End

**IDE** Integrated Development Environment

**NF.R.** Non-functional requirement

**F.R.** Functional requirement

**DOM** Document Object Model

# 1 Introduction

This bachelor thesis covers the approach and documentation of my implementation of new features to a web-based code editor for data pipelines. The goal of this work was to contribute to the JValue Project, a research project from the Professorship for Open Source Software at Friedrich-Alexander-University in Erlangen.

The overall target of this thesis is to further improve and implement new features to the already existing code editor of the JValue Hub. The JValue Hub is the collaboration platform of the JValue project, which focuses on building a new programming language for data pipelines while also improving the collaboration between data engineers and research scientists. A more detailed overview of the project will be provided in chapter 1.1.

The long term goal for the JValue Hub is to create a dedicated collaboration platform specifically designed for data engineering that can be used from data engineering experts but also from experts within a respective research field. As of now there is now common standard or tailored solution for working large data sets, but rather existing platforms for software engineering get repurposed. The most dominant platform for sharing data engineering projects at the moment is GitHub. A platform specifically designed for software engineering. This means that essential functionality is missing for data engineering and collaborating on data projects, e.g. the direct access to output data and thereby effectively presenting the projects. In order to create a new and competitive platform that gets both equally adopted by the community of data engineers and research scientists, it is thereby important to provide well-built and to both user groups well-known functionality. [1]

In collaboration with a key member of the JValue team, we decided that an ad-hoc runtime for the pipeline editor would be the best first step to improve the capabilities of the existing editor and the JValue Hub respectively. This feature would include receiving runtime information and the pipeline model output from the ad-hoc run, and thereby introduce the capability to debug a pipeline model

[1] Heltweg and Riehle, 2023

from the editor page.

At the beginning of this thesis, the JValue project will be briefly introduced and some necessary context for this thesis is provided. The following chapter summarizes the benchmarking that was performed during the research on web editors. For this benchmark, common code and pipeline editors on the market were assessed and a common default layout familiar to the end user was derived.

Later on, the requirements of the new features are introduced, followed by the most relevant aspects of the current JValue Hub architecture and consequently highlights of the implementation are discussed. In the final chapter, the implementation and what has been achieved during this thesis is evaluated and a possible roadmap for future features that would enhance the capabilities of the online pipeline editor of the JValue Hub is provided.

## 1.1 JValue Project

The JValue Project is an ongoing research project of the Professorship of Open-Source Software at Friedrich-Alexander-University in Erlangen. The overall goal is to provide (data) scientists easy access to open data and enable them to share, discuss and collaborate. In order to achieve shareable and consistent ETL (Extract, Transform and Load) Pipelines, the project team is developing a new modeling language called jayvee.
In addition to the language, the team is creating the JValue Hub. This hub offers the ability to create, share and contribute to data pipeline projects written with jayvee. It allows the user to directly run their data pipeline model within the hub.

### 1.1.1 Jayvee

The Jayvee language is a domain-specific language which is currently developed at the professorship for open source software at Friedrich-Alexander-University in Erlangen. The language is used to model ETL data pipelines. The targeted user varies from experienced developer to research scientists across different expertises. The language itself is currently developed and follows its own syntax. Pipeline models written in jayvee are either developed on a local machine or web based via the collaboration platform JValue Hub.

### 1.1.2 JValue Hub

The JValue Hub is a collaboration platform for developing data pipeline models that are built with the jayvee language. The hub itself separates into two main components. The first component is the hub itself, including several user facing

functionalities such as creating new pipeline models, forking and contributing to existing project of other users. Another functionality is the possibility to edit projects in the web editor directly. As of release v0.5.0-alpha, the web editor supports basic syntax highlighting. The pipeline code can be executed from the hub itself. The second component includes cloud features of the Hub such as the model and file storage, the runtime services to execute data pipeline models and the versioning of projects for the user and between them.

# 2 Common interfaces of editors

While the typical software engineer and data scientist does not necessarily spend the majority of his or her time programming with an editor, it still is one of the most important tools that can have a crucial effect on the result of your work. Whereas the editor itself is not the main contributor to your work, a poorly designed editor or an interface that requires time for accommodation can increase the risk of decelerating the process. Whereas, an editor with proper functionality and a well-equipped toolbox of functionality can enhance your work significantly.

Based on the foundations of the jayvee language and its current state of development, the assumption was made that for now, the typical JValue Hub user already has preliminary skills in programming and data science in general. Consequently, the typical user is already familiar with existing code and pipeline editors.

To get a better understanding of the current industry standards and reduce my own bias towards an optimal editor, an industry benchmark among different software providers was performed and taken into consideration. Comparing different editors and deriving common characteristics helped to design relevant features and a familiar UI layout. During my research I focused on well established code and pipeline editors such as VS Code by Microsoft, IntelliJ by JetBrains or Apache StreamPipes.

## 2.1 Editor categories

Editors can be separated into multiple categories. First, they can be split into code and pipeline editors, and second into offline and online editors. Going further, editors could be separated into code editors and integrated development environments short IDEs, but in the context of this thesis, a focus on IDEs was decided, with no further classification.

### 2.1.1 Pipeline editors

During the initial explorative research on 'ETL pipelines' and 'ETL pipeline editors', similarities across a majority of editors and ETL pipeline solution providers could quickly be derived. Common editors to model ETL pipelines are often based on low code solutions. Low code ETL pipeline solutions are commonly build by adding elements to a graph which is centered in the middle of the screen, with information about components and elements being on the side of the screen or in a menu on the top. Low code solutions often use an existing language to generate code based on user input. In the case of Apache StreamPipes it is built with Java. Similar to Jayvee, Apache Streampipes includes prebuilt extractors, transformers and loaders which can be configured via the GUI. In order to add functionality and own components to the editor, the user needs to create and add a new file, e.g. a new Java file, to the project. The coding of this element is then performed in a separate code editor, which in most cases is not integrated in the pipeline editor.

Since javee is by design not a low code solution, I focused on conventional code editors instead.

### 2.1.2 Code editors

As explained in chapter 2.1, code editors can be separated into different categories. The biggest difference is between regular code editors and Integrated Development Environments or IDEs. IDEs often come with a wide variety of functionality. Predominantly, they can be separated from code editors by the possibility to execute and compile code from within the editor itself. While regular code editors by themselves do not offer more pronounced functionalities than editing files and syntax highlighting, I will focus on IDEs for my comparison.

IDEs can as well be sub-categorized. It is differentiated between Online Development environment (ODE) which can be used from the browser and regular IDEs which are run locally. As locally installed IDEs provide more functions and have a broader user base, I will include them in my comparison, even though the JValue Pipeline Model Editor is developed to be an ODE.

To compare different editors and get a good understanding of the current industry standard, I decided to compare the most popular IDEs and ODEs from the "IDE" and "ODE index" [1] published on GitHub. The data is based on the total google search results for a given editor in a certain time period. For each editor, the latest build as of October 2023 has been taken into consideration. The detailed overview of editors and their characteristics can be seen in appendix B.

[1] GitHub, 2023

## 2.2 Common layout for code editors

In comparison, 18 out of 20 editors provided a customizable interface, of which 16 were shipped with a similar default layout: In the center the code editor is located, more information like a file explorer are on the left or right side of the center window. Additional information or input options such as a terminal or runtime information are found on the bottom. Menu items and extra buttons are often located above the code window. All the 16 editors used similar symbols to visualize commands, such executing the code from the editor or stopping the execution. The two of the other four editors did place the console on the right side of the window. The remaining two editors were either developed for a different use case (Sublime Text is primarily a text editor with customizable coding capabilities) or no longer developed (Koding).

# 3 Scope and requirements

One of the goals for the JValue Hub is to offer a collaboration platform for researchers and data scientists to collaborate on date pipeline models. Multiple steps are necessary to set up your local IDE for developing with the jayvee language. Some of them require a certain understanding on how to configure your own IDE. Besides experienced data engineers, another target user for the jayvee language are scientists from different research areas who need to explore large data sets. The second user does not necessarily have the skill set to set up a local version of jayvee neither to build large data pipelines on their own. This group might only need to make minor changes or updates to an existing pipeline model. In order to enable this user to work with the jayvee language and provide the necessary tools to use larger data pipeline models, the current online editor requires additional features to enhance the development experience and truly enable collaboration.

Currently, the JValue Hub pipeline editor is missing some essential development features to develop sophisticated and large jayvee models online. One of them is an ad-hoc runtime that enables the user to run their model from within the editor. Another important aspect of data engineering that is missing, is evaluating the output that is generated by the pipeline model. Both functionalities would enable the user to continuously develop while debugging without having to switch between contexts.

To enhance the development experience, both functionalities were added to the JValue Hub in the course of my thesis. The goal was to provide the JValue Hub user with the ability to run their code and get instant feedback that supports debugging or improving an existing model without leaving the editor and having to switch between pages on the hub. Therefore, the Ad-Hoc runtime will be implemented which enables the user to execute the code as well as provides him with additional information in the form of the console output from the jayvee run, additional runtime information and the output sink that is produced by the jayvee model.

As the code editor did not provide any additional functionality other than editing

your model and having syntax highlighting, the current layout and design was extended to integrate into the existing interface.

## 3.1 User story

In agile projects, it is common practice to formulate new features in the form of user stories. A user story helps to define and communicate the scope for the project or work. It also helps as a base if fundamental decisions have to be made. They are typically written in one specific format: As a "Role", I want "what", so "that".

Based on the practice of formulating user stories, the following user story was used:

As a JValue Hub User, I want to run my model from within the editor page and receive the console output as feedback so that I can verify if my model is working and debug directly if necessary.

## 3.2 Scope definition

The thesis was implemented in an agile project set-up. To remain in time and achieve respective deliverables, it was crucial to define a clear scope for the project. This scope helped to define if new requirements would be added or should be part of future research outside this thesis.

### 3.2.1 In scope

The definition of the target implementation requires changes to layout and implementation of the frontend layout that is related to the code editor. To link the frontend layout with functionality from the backend, it is also in scope to include every change to initiate and display the console output from the ad-hoc run.

### 3.2.2 Out of scope

As the output data that is received from the jayvee model is not generated by the JValue Hub but the jayvee language instead, I decided to define any changes to the output data as out of scope for this thesis.

## 3.3 Requirements

To measure the quality of my implementation and to verify if everything works as intended, certain requirements had to be defined beforehand. Additionally,

I had to add or changed certain requirements during the development process when new functionality or change requests came up during my exchanges with stakeholders.

Requirements can be split into two categories. Functional and non-functional requirements. Non-functional requirements group requirements do not influence the behavior of the implementation, but rather ensure code quality and consistency within the project. Functional requirements define expected behavior of the final implementation.

The next two chapters describe the functional and non-functional requirements that were set for this thesis in more detail.

## 3.4 Non-functional requirements

Non-functional requirements (NF.R.) are shared across the entire development process and therefore not listed individually per development story. The NF.R.s are defined with main focus on the quality of the implementation itself and how it can be integrated into an existing code base of the JValue Hub. Table 3.1 provides an overview of all non-functional requirements.

| Req. ID | Description |
|---|---|
| NF.R.1 | Use existing architecture |
| NF.R.2 | Follow current implementation |
| NF.R.3 | Reusable and clean code |
| NF.R.4 | Follow design principles and general layout |
| NF.R.5 | No side effects on existing implementation |

**Table 3.1:** Non-functional requirements

#### NF.R.1 - Use existing architecture

The goal of this requirement is to reduce the amount of code that needs to be maintained throughout the lifetime of the project. One way to achieve this is by reusing an existing component instead of building a new one with a single purpose only.

#### NF.R.2 - Follow current implementation

One key aspect of maintainable code is to be persistent with implementation throughout the code base. This supports the readability of the code itself and thereby makes maintaining the code easier.

**NF.R.3 - Reusable and clean code**

There should not be any repetitive code within the implementation itself. If code repeats itself, best practice would be to create an own component that handles the repeating part of the code. This makes refactoring easier when changes are required later on.

**NF.R.4 - Follow design principles and general layout**

The entire interface and layout should be in line with the JValue hub design. This ensures a persistent look and feel for the user throughout the entire application.

**NF.R.5 - No side effects**

The implementation does not interfere or change any existing implementation to the extent that an already designed element of the JValue Hub needs to be changed.

## 3.5 Functional requirements

To manage work packages efficiently, the implementation was split into two major development stories. Consequently, there were two sets of functional requirements (F.R.), one for each development story. The first story implemented all changes to the backend and frontend that are necessary to trigger an ad-hoc run. The second story included displaying the console output, additional runtime information and adding functionality to download the output sink from the ad-hoc run in the user interface.

| Req. ID | Development story | Description |
|---|---|---|
| F.R.1.1 | Ad-hoc runtime | Meaningful layout |
| F.R.1.2 | Ad-hoc runtime | Information on the latest run |
| F.R.1.3 | Ad-hoc runtime | Execute runtime from editor |
| F.R.1.4 | Ad-hoc runtime | The entire ad-hoc run is stored permanently |
| F.R.2.1 | Console output | Layout for console output |
| F.R.2.2 | Console output | Display console output |
| F.R.2.3 | Console output | Previous run result on page load |
| F.R.2.4 | Console output | Storage of console output from jayvee model |
| F.R.2.5 | Console output | Additional run result information in extra tab |
| F.R.2.6 | Console output | Download of output sink |

**Table 3.2:** Functional requirements

### 3.5.1 Ad-hoc runtime

This development story included the ad-hoc execution of the current model that is displayed in the editor including all necessary changes in the front- and backend. The following requirements were defined for this story.

#### F.R.1.1 - Meaningful layout

Buttons with self explaining icons are added to the interface of the editor. The ad-hoc run will later be started with these buttons. New users should be able to understand the functionality of the interface intuitively.

#### F.R.1.2 - Information on the latest run

The current state of the ad-hoc run is displayed in the frontend. As pipeline models for large datasets can take longer to execute, visual feedback about the running state is displayed in the frontend.

#### F.R.1.3 - Execute runtime from editor

The user is able to initiate an ad-hoc run from the editor page itself.

#### F.R.1.4 - The entire ad-hoc run is stored permanently

To potentially access the information and output sink of the ad-hoc run later on, the run should be stored permanently in the database.

### 3.5.2 Runtime information

The console output and runtime information of the ad-hoc run are displayed in a dedicated window within the page. I defined the following requirements for this story.

#### F.R.2.1 - Layout for console output

A new layout for displaying the console output from the ad-hoc run is implemented. The layout should be in line with industry standards and be intuitive.

#### F.R.2.2 - Display console output

The output from the jayvee model of the ad-hoc run is displayed in the output window.

#### F.R.2.3 - Previous run result on page load

On reloading the page or when the editor is accessed after re-logging, the console output from the previous ad-hoc run is displayed in the console output window.

#### F.R.2.4 - Storage of console output from jayvee model

The console output from the jayvee model is permanently stored with the run result from the ad-hoc run. This requirement is essential for the development story.

#### F.R.2.5 - Additional run result information in extra tab

Additional information on run results are displayed in an additional tab within the console box.

#### F.R.2.6 - Download of output sink

The output sink that resulted from the ad-hoc run is accessible within the runtime information tab. The user is able to download the file.

# 4 JValue Hub Architecture

The following chapters will provide an overview of the existing architecture of the JValue hub. Herein, I will explain the individual services and their functionality. This thesis includes the latest changes up to release v0.5.0-alpha of the JValue hub.

The application can be split into the frontend implementation called the "hub-web" as well as the following services which are consumed by hub-backend: the pipeline service, the runtime service and the file service. The majority of code is written in TypeScript and facilitates the Node.js framework. The frontend is implemented using ReactJS. Services communicate via HTTP requests that are described with the OpenAPI specification. First, I will explain the individual functionality of each service in the following chapters. Thereafter, an overview of the most important frameworks and libraries that are used by the JValue Hub is being provided.

## 4.1 Hub-web

The frontend application of the JValue Hub, the hub-web, is developed with TypeScript and uses the React library. React enables the use of reusable components. To ensure consistent design, the JValue Hub implemented a JValue Design System, which provides guidelines and a collection of assets for use in the JValue frontend. In the hub-web API, endpoints from the hub-backend are described with the help of Redux. The endpoints are used to trigger events and retrieve information from the JValue file storage.

## 4.2 Hub-backend

The hub-backend handles all requests originating from hub-web, as well as communication between other services. All data that is directly processed from the hub is stored and handled in the hub-backend. The backend handles requests related to the project, the corresponding repository and all user related topics

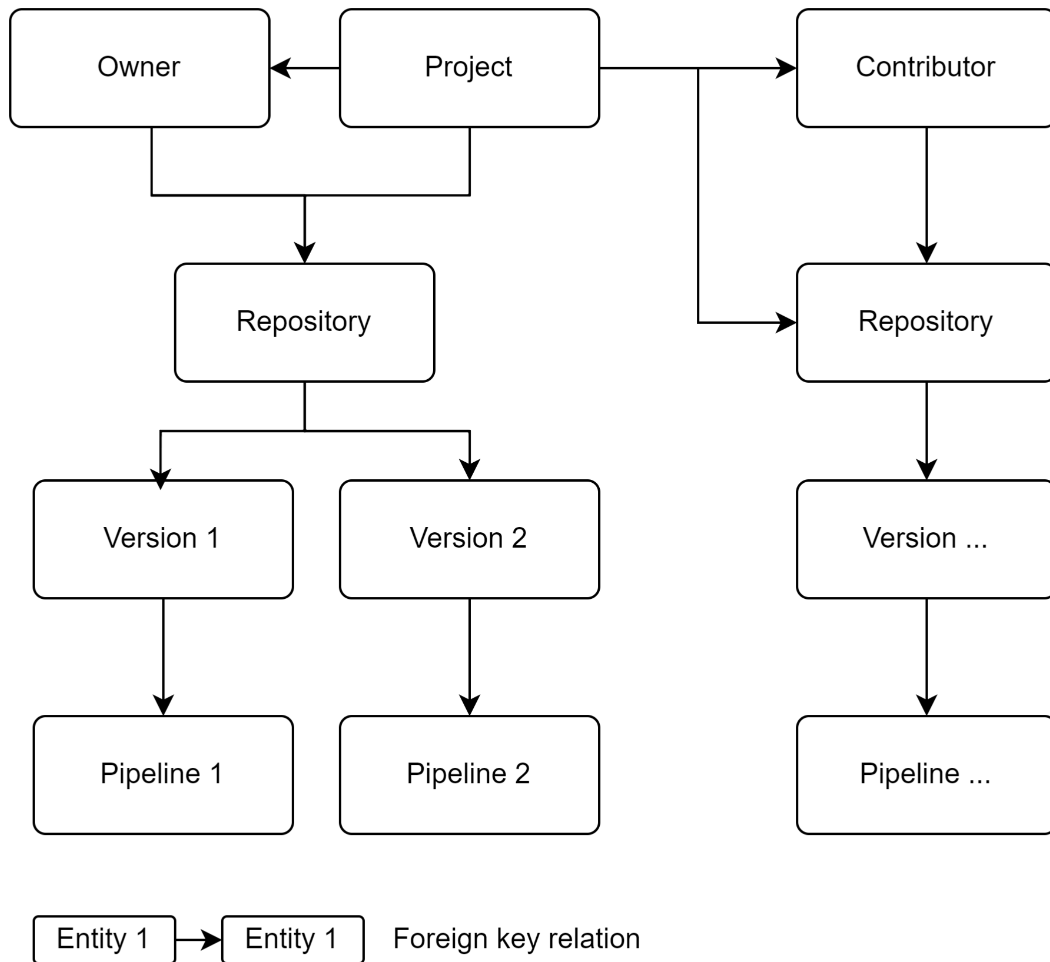


**Figure 4.1:** Repository pipeline relation - fact check!!!

such as authentication and creation. Requests related to pipeline management, runtime and file storage are passed on to the individual service. The hub-backend is build with the NestJS framework, and HTTP endpoints are described and made available to the frontend and other services.

### 4.2.1 Project, repository and versioning

The JValue Hub implements a git service for handling the versioning of projects that are developed in the hub. By forking and contributing to an existing project, multiple repositories can exist for a single project. Within a repository, the JValue model is stored and versioned. This information is then used to link a pipeline to a specific version of a repository. The relation between the entities can be seen in figure 4.1.

## 4.3 Pipeline service

The pipeline service handles requests for the data pipeline itself. A pipeline represents a container for a specific version of a jayvee model. The pipeline can be executed and scheduled for runs. The current state of the jayvee model and additional metadata is stored in a pipeline entity. For handling the actual runtime and retrieving information about past runs, the pipeline service communicates with the runtime service.

## 4.4 Runtime service

The runtime service handles requests for pipeline model runs and individual runs of a jayvee model. The runtime calls the jayvee interpreter and initiates the run for the model. The output is passed on and stored by the file service. The simple runtime also actively calls the pipeline service to update the run status of individual runs.

## 4.5 File service

The file service receives requests originating from the runtime service when executing a jayvee model in order to store the resulting output. It also handles all requests from the backend that require the storage of permanent data. Currently, all jayvee models store the output that is generated. The resulting file sink is handled by the file service. This service is also used to store and retrieve data sinks and provide it to the user.

## 4.6 Important frameworks and libraries

### 4.6.1 React and Redux

The open-source library React is used to create dynamic User interfaces. Its component-based architecture supports modular and reusable components. The actual Document Object Model (DOM) that is used to render the website is updated by the virtual DOM of React. This reduces re-rendering of the whole web page and ensures a responsive and performant usage.

Redux is an open-source state management library for JavaScript applications. It enables the developer to handle data and state changes. The application state is stored in a single, immutable object called store. State modifications are driven by dispatching actions. Functions that update the state of an application are

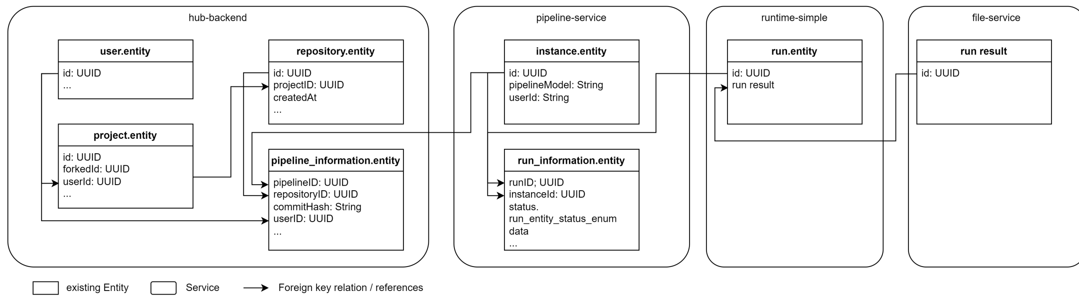


**Figure 4.2:** Simplified class diagram

called reducers. For the JValue Hub, Redux is mainly used to describe and handle all API consumption by the frontend.

React components can connect to the Redux store and access and update the application state. In addition, React components can trigger reducers in Redux. The interaction between React and Redux enable components to reflect the state changes from Redux and ensure a consistent state across the application.

### 4.6.2 NestJS and typeORM

NestJS is a Node.js framework designed to build scalable and maintainable server-side applications. NestJS enables a modular and component-based architecture and thereby emphasizes on code reusability. With its built-in features, it simplifies the creation of web APIs and microservices. NestJS's has a strong orientation on TypeScript.

TypeORM is an Object-Relational Mapping (ORM) library for TypeScript and JavaScript, which simplifies database integration in Node.js applications. TypeORM translates classes and object structures from Typescript into relational database schemas such as PostgreSQL. Without any SQL queries, the developer can query and perform data manipulations in an object-oriented and type safe manner.

## 4.7 Communication

The communication between services is done via HTTP-requests. The endpoints are described with NestJS which implements the OpenAPI standard for Restful APIs. In figure 4.3 an example of the pipeline API with all involved classes is shown. As this thesis mainly focuses on implementing new features to call the runtime, only the use case for creating a pipeline is shown in the communications diagram in figure 4.4. The diagram shows the interaction between the hub-backend, the pipeline-service, the runtime and file service when a run is triggered by the end user.

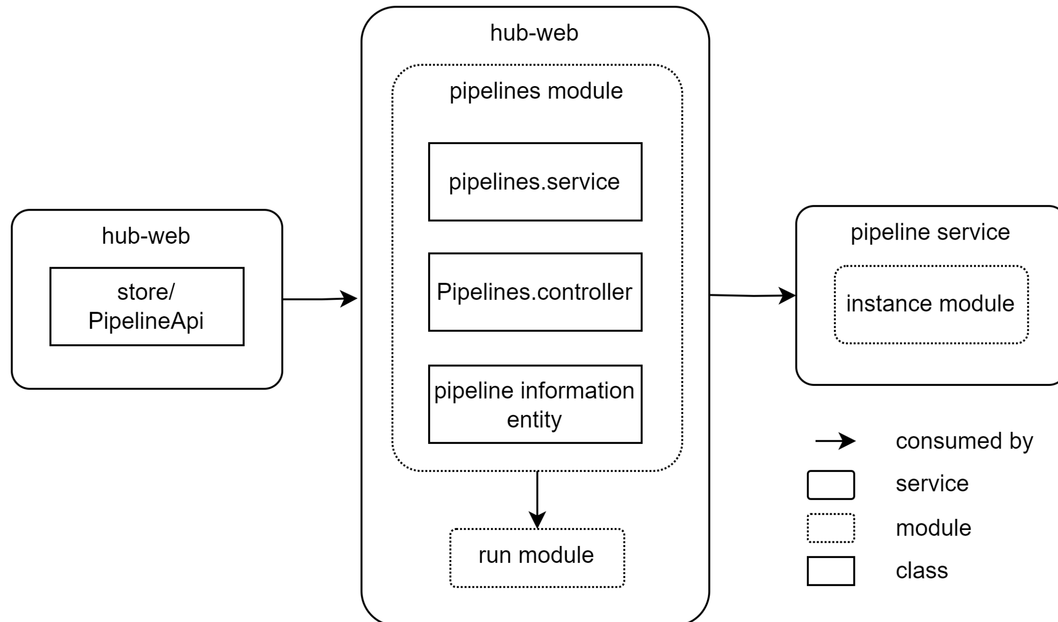


**Figure 4.3:** Pipeline API

## 4.8 Data storage

The data of the JValue Hub is stored in a Postgres database. As described in section 4.6 the TypeORM library is used to describe database schemas based on the implementation in Typescript. The table structure of the database is thereby an exact representation of figure 4.2. Information entities are created to store relevant information, e.g. foreign keys, within a service. This allows for faster queries for a specific entity and reduces inter service communication.

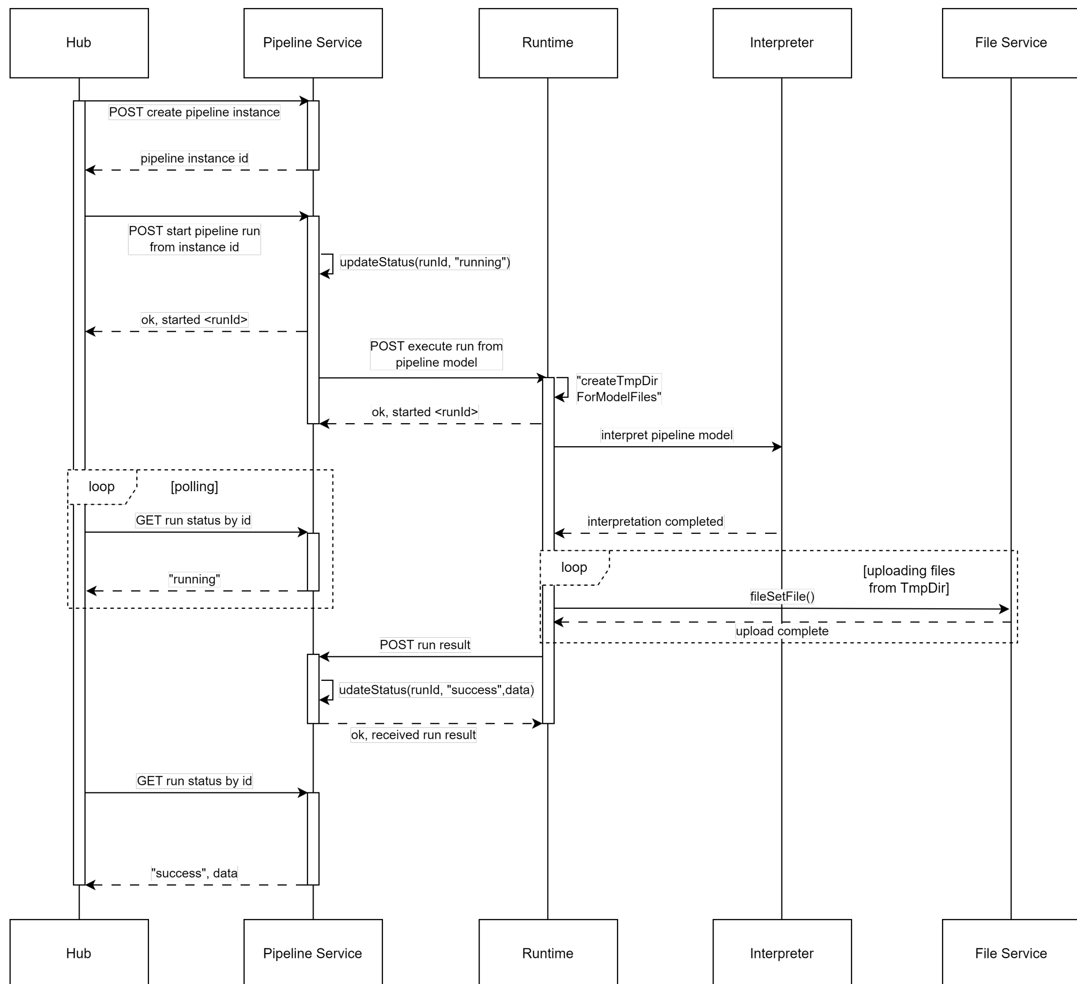


**Figure 4.4:** Communication between services of a single pipeline run

# 5 development and implementation

In the following chapters, I will describe my implementation, elaborate on certain decisions and my way forward.

## 5.1 Layout and design

Information about runtime or console is located at the bottom of the screen. Among the top ten editors, most editors provide a customizable interface. This feature will not be implemented. The current focus of the JValue project and this thesis is to generate new functionality features. Implementing the most common default layout to the editor would allow adding new functionality to the existing editor layout and still provide a familiar user experience to developers. A visualization of the interface can be seen in figure 5.1.

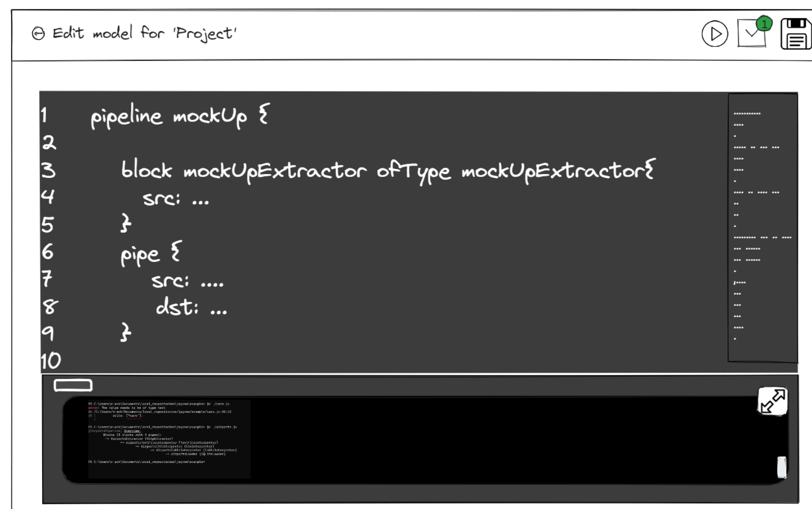


**Figure 5.1:** Mock layout with console

### 5.1.1 Design

For the final design of the layout implementation, no new components were introduced. The new editor page was built by reusing existing components from either the JValue Design System or external components that have already been introduced within the project, such as icons from the react-md library. This allows to maintain the current appearance and ensures a continuous experience throughout the usage of the JValue Hub. The final design and implementation is shown in figure 5.2 and 5.3

### 5.1.2 States of the pipeline editor

To provide visual feedback to the user, I decided to integrate the current state of the ad-hoc run into the interface. Based on the different states, the interface would slightly change to let the user know if a pipeline is still running and if the result should be available in the output box of the layout. A simple representation of the different states can be seen in the state machine in figure 5.4. The corresponding mock-ups can be seen in figure 5.5 to 5.8. I decided to use common symbols and colors instead of written titles for the buttons. Well selected symbols allow users to use the JValue Hub even though the implemented languages might not be their native language. In terms of accessibility, I decided that it should be sufficient to use symbols. Additional color coding is only optional and redundant information. It is worth mentioning that there is no implementation of common standards for accessibility in web design.

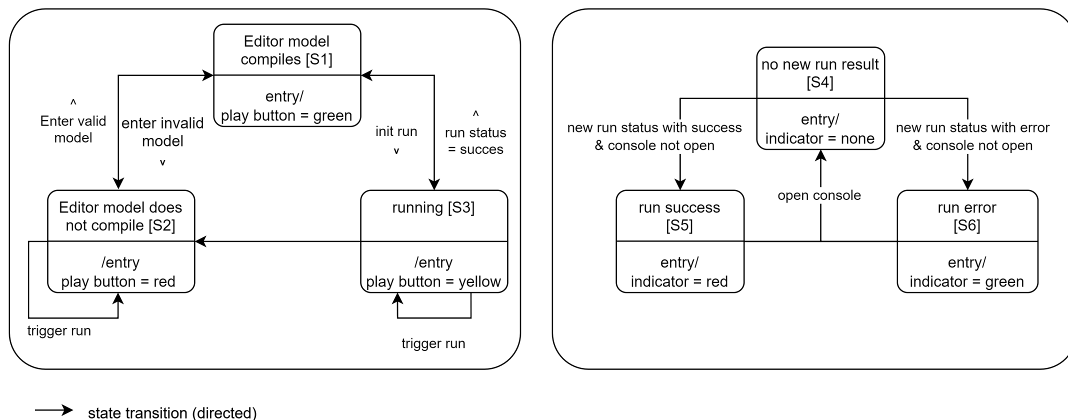


**Figure 5.4:** State for the pipeline editor page

JValue Hub

MM

Edit model for 'Test'

```
1  // SPDX-FileCopyrightText: 2023 Friedrich-Alexander-Universitat Erlangen-Nurnberg
2  //
3  // SPDX-License-Identifier: AGPL-3.0-only
4
5  // Example 1: Cars
6  // Learning goals:
7  // - Understand the core concepts pipeline, block, and pipe
8  // - Understand the general structure of a pipeline
9
10 // 1. This Jayvee model describes a pipeline
11 // from a CSV file in the web
12 // to a SQLite file sink.
13 pipeline CarsPipeline {
14
15     // 2. We describe the structure of the pipeline,
16     // usually at the top of the pipeline.
17     // by connecting blocks via pipes.
18
19     // 3. Verbose syntax of a pipe
20     // connecting the block CarsExtractor
21     // with the block CarsTextFileInterpreter.
22     pipe {
23         from: CarsExtractor;
24         to: CarsTextFileInterpreter;
25     }
26
27     // 4. The output of the "from" block is hereby used
28     // as input for the "to" block.
29
30     // 5. More convenient syntax of a pipe
31     CarsTextFileInterpreter -> CarsCSVInterpreter;
32
33     // 6. Pipes can be further chained,
34     // leading to an overview of the pipeline.
35     CarsCSVInterpreter
36         -> NameHeaderWriter
37         -> CarsTableInterpreter
38         -> CarsSQLLoader;
39     CarsTableInterpreter
40         -> CarsPostgresLoader;
41
42
43     // 7. Below the pipes, we usually define the blocks
44     // that are connected by the pipes.
45
46     // 8. Blocks instantiate a blocktype by using the oftype keyword.
47     // The blocktype defines the available properties that the block
48     // can use to specify the intended behavior of the block
49     block CarsExtractor oftype HttpExtractor {
50
51         // 9. Properties are assigned to concrete values.
```

TODAY AT 10:30 PM    RUNINFORMATION

```
[CarsPipeline] Overview:
        Runtime Parameters (6):
                HUB_DB_HOST: localhost
                HUB_DB_PORT: 5432
                HUB_DB_USERNAME: postgres
                HUB_DB_PASSWORD: postgres
                HUB_DB_DATABASE: pipeline_ce4a2159-f6ba-4fc3-9f5a-18f50e7ea694
                HUB_DB_TABLE: runresult_ca7c09eae8004be697b374719c056820
        Blocks (7 blocks with 4 pipes):
         -> CarsExtractor (HttpExtractor)
                 -> CarsTextFileInterpreter (TextFileInterpreter)
                         -> CarsCSVInterpreter (CSVInterpreter)
                                 -> NameHeaderWriter (CellWriter)
                                         -> CarsTableInterpreter (TableInterpreter)
                                                 -> CarsSQLLoader (SQLiteLoader)

                                                 -> CarsPostgresLoader (PostgresLoader)

[CarsExtractor] Fetching raw data from https://gist.githubusercontent.com/noamross/e5d3e859aa0c794be10b/raw/b999fb4425b54c63cab088c0ce2c0d6ce961a563/cars.csv
[CarsExtractor] Successfully fetched raw data
[CarsExtractor] Execution duration: 326 ms.
[CarsTextFileInterpreter] Decoding file content using encoding "utf-8"
[CarsTextFileInterpreter] Splitting lines using line break /\r?\n/
[CarsTextFileInterpreter] Lines were split successfully, the resulting text file has 33 lines
[CarsTextFileInterpreter] Execution duration: 2 ms.
[CarsCSVInterpreter] Parsing raw data as CSV using delimiter ","
[CarsCSVInterpreter] Parsing raw data as CSV-sheet successful
```

**Figure 5.2:** Final design editor page

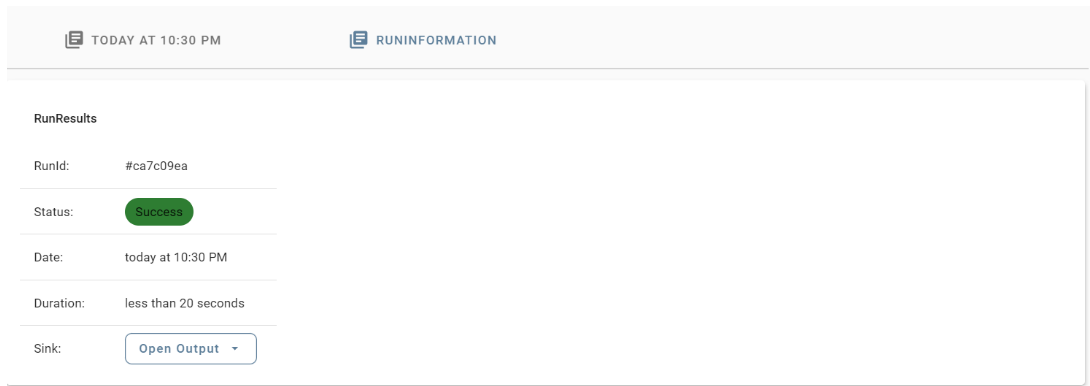


**Figure 5.3:** Final design runtime information

**Figure 5.5:** Mock up: Editor page state S1 and S4

**Figure 5.6:** Mock up: Editor page state S1 and S5

**Figure 5.7:** Mock up: Editor page state S3 and S4

**Figure 5.8:** Mock up: Editor page state S2 and S6

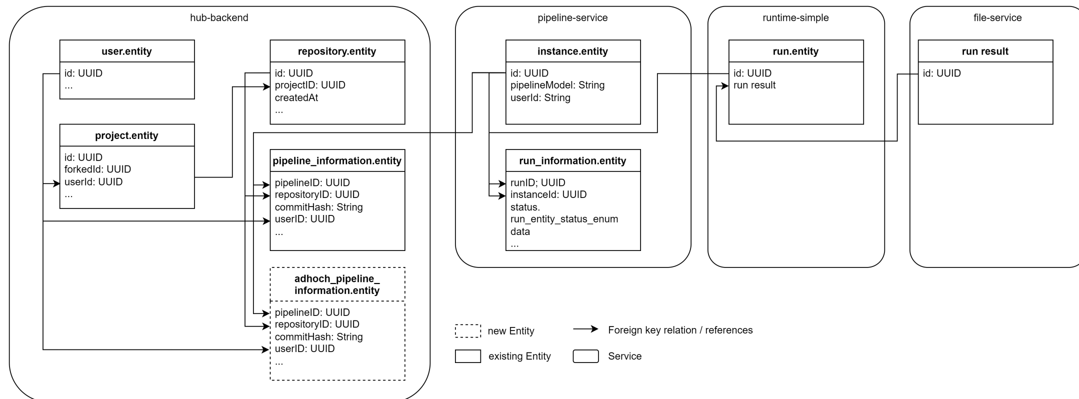


**Figure 5.9:** New simplified class diagram

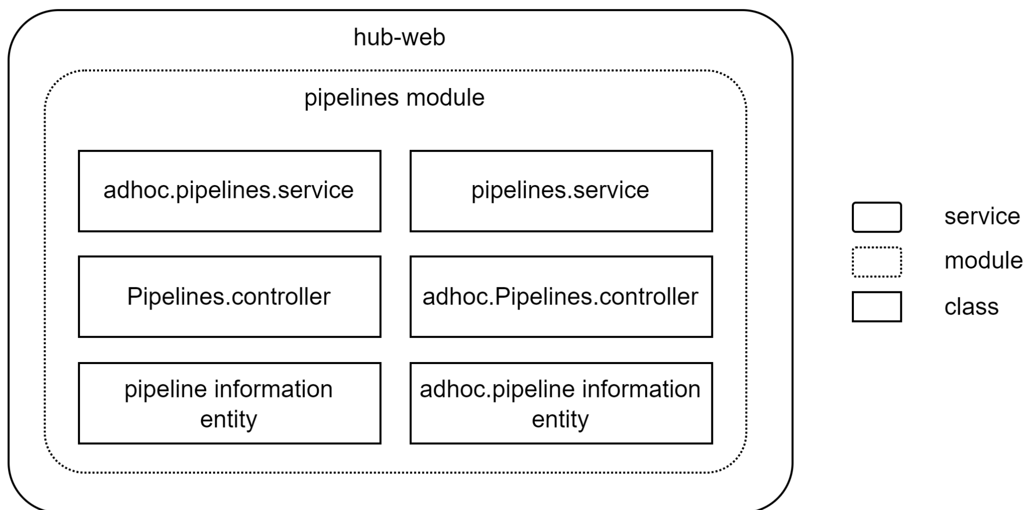


**Figure 5.10:** Implementation for pipelines module including adhocPipelines

## 5.2 Implementation into existing architecture

Like in chapter 3.4, non-functional requirements described, it should be achieved to minimize changes to the existing logic within the JValue Hub code base and create an implementation that minimizes the effort to refactor any component or service later in the development process. Respectively, the existing architecture was expanded as shown in figure 5.10 and 5.9. In the following chapters, I will explain different options for some major decisions that lead to the final implementation.

### 5.2.1 Additional libraries and framesworks

During the implementation of the ad-hoc runtime and console output functionality, no additional libraries or frameworks were added to the project. By primarily reusing existing libraries, frameworks and components, no additional external dependencies were required. Consequently, no bill of materials was added to this thesis.

### 5.2.2 Runtime environment

The first major decision during the implementation of the ad-hoc runtime feature was the location of where to run the jayvee model. In general, there were two options to be considered: The model could be run on client or server side. Running on server side would take up additional resources on the server. However, this option would allow the JValue hub full control over the output and information that are generated by running a pipeline model. This way, it would be easier to implement additional features in the future. Information that is gathered during the execution of a jayvee model could be analyzed to further improve the jayvee language. Common errors could be identified and lead to an overall better user experience.

The benefit of running the model on client side would be less resources required by the JValue Hub server and simultaneously allow the user to keep data on premise while still using the online editor. The disadvantage of this option is that the jayvee interpreter can not be run within the browser. To mitigate this issue, a local version of the jayvee interpreter would be required. This option was abandoned, as it would require an additional setup by the user before using the online editor and could easily increase complexity when dealing with different versions of the jayvee interpreter. By running the interpreter as a cloud service, it can be guaranteed that the syntax highlighting and interpreter of the pipeline editor always match.

As the benefits of option one clearly predominated its downside, the ad-hoc runtime was implemented on server side.

### 5.2.3 New architecture

After analyzing the existing architecture of the JValue Hub and the interaction between the services, four possible implementation options were discussed. Options 1 to 4 are described in figure 5.11

The first option was to duplicate the existing logic for the creation of a run entity with only minor adaptations. This implementation would allow for full control of the new ad-hoc classes. All new requirements could be implemented within that specific class without any side effects towards existing functionality. The

downside of this approach would be the complex effort required for refactoring if any shared logic would need to be changed.

As the pipeline service only transmits the requests between hub-backend and runtime and does not store any necessary information to differentiate between an ad-hoc run and a regular run, the second option was to implement an additional AdhocRunEntity. This still would allow for flexibility to change certain aspects of the ad-hoc run implementation specifically while reducing the code that needs to be maintained by the pipeline service component. In case of any fundamental changes, this option still requires rework of both the ad-hoc run and the run entity logic.

The third option was to add a new ad-hoc Pipeline Information Entity that would store information about which pipeline belongs to which version of a repository. This is possible as the pipeline service, as well as runtime service, do not store nor require any information about the caller and the context from where it was called. Thereby, all existing logic for triggering a run can be reused, and no changes are required for this implementation. In addition, in case any of the services needs to be refactored, only one component will need to be changed. As this approach reuses existing architecture, there is no information on the type of run available at the pipeline and runtime service.

Option four is similar to option three. The idea behind this solution is to add a boolean value to the pipeline information entity that flags a certain pipeline as an ad-hoc pipeline or regular pipeline. This implementation would require no changes in the pipeline or runtime service, however would require refactoring of existing implementation of any pipeline-information entity implementation.

After reviewing all alternatives, option three was implemented to add the functionality of triggering an ad-hoc pipeline run. This implementation did not require any changes to the existing code, while on the same side reduces the effort for any refactoring in the future to a minimum. The downside of this approach is the lack of flexibility on the resulting runtime, e.g. it is not possible to not store the target sink of a pipeline run. After discussion and thorough evaluation together with stakeholders, it was decided that that keeping the sink from the model was the preferred behavior, as it might be needed in future features. In addition to that, the extra storage space is not considered to be an issue and therefore the easier maintainability outweighed the extra storage required.

### 5.2.4 Repository, version and pipeline relation

The updated relation between repositories, versions and pipelines can be seen in figure 5.12. While maintaining the 1:1 relation between version and pipeline, a 1:n relation between version and ad-hoc pipeline was introduced. This was implemented as the pipeline stores the corresponding jayvee model. As ad-hoc

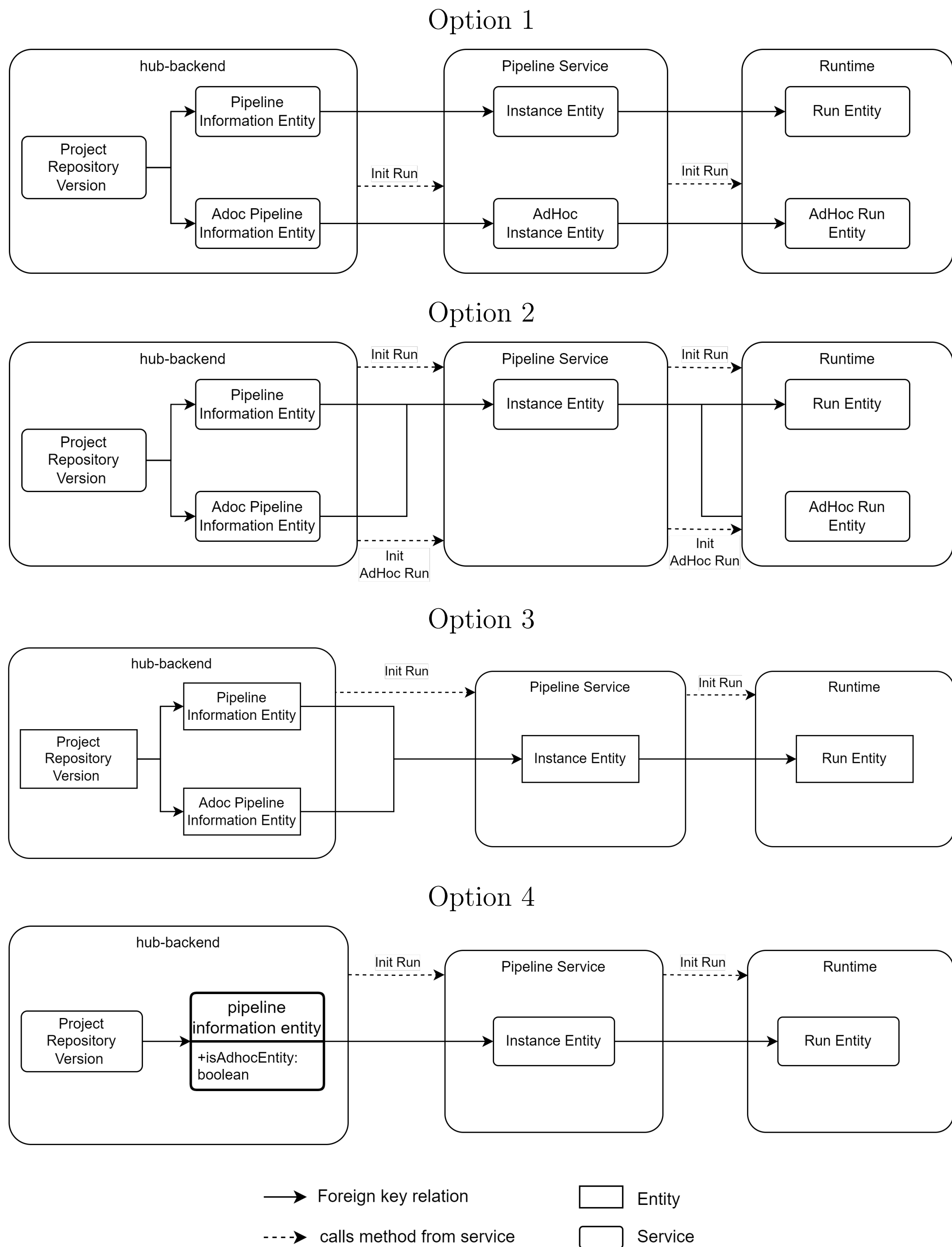


**Figure 5.11:** Option 1-4 for pipeline information entity integration

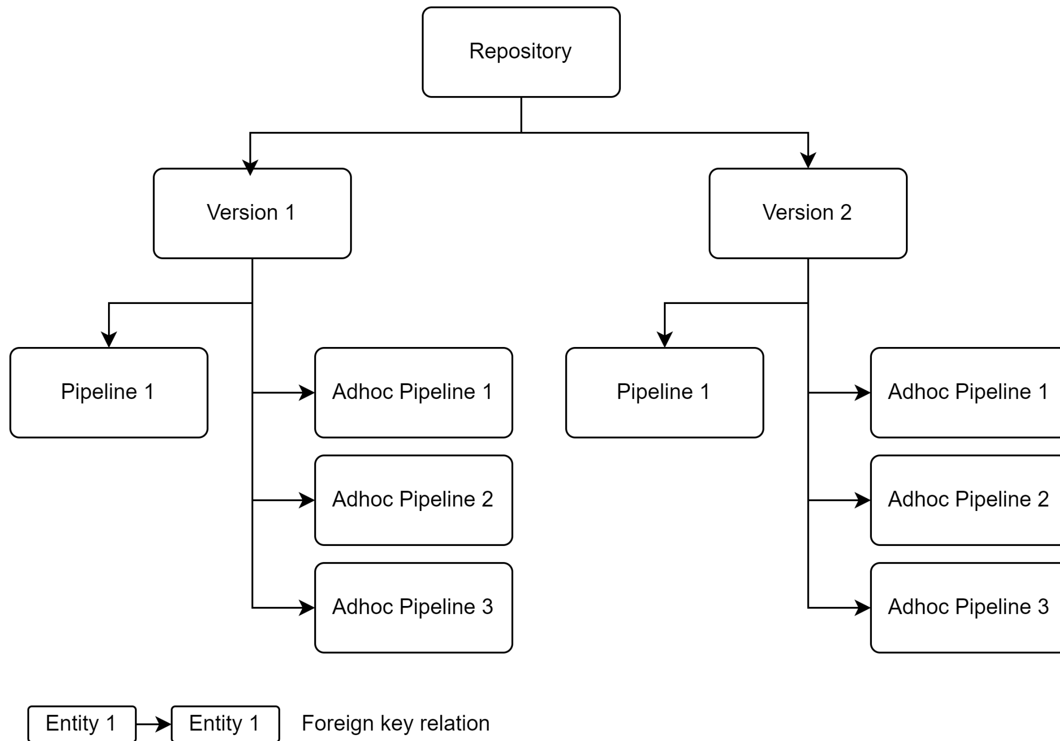


**Figure 5.12:** Repository ad-hoc pipeline relation

runs can be executed on uncommitted versions of the jayvee model, this way multiple ad-hoc runs can be stored within the same version. By keeping the relation between ad-hoc pipeline and version, indexes can be used to enhance query performance over the lifetime of the project. In addition, this approach would allow for easier clean-up of unused ad-hoc pipelines.

#### Adaption to class hierarchy

Option three was implemented by moving shared logic of the pipeline and ad-hoc pipeline service and controller in the hub-backend into a base-pipeline class. By only extending the base-pipeline classes if additional logic is needed for either the pipeline or ad-hoc pipeline, the code gets easier to maintain in the future. A visualization of the inheritance logic can be seen in figure 5.13.

### 5.2.5 Editor page

To reduce the complexity of individual components and pages within the application, the implementation for the code editor was moved from a component mounted in an overlay of the project page into an individual pipeline model editor page. In React, each page has its own state, therefore, two options on how to

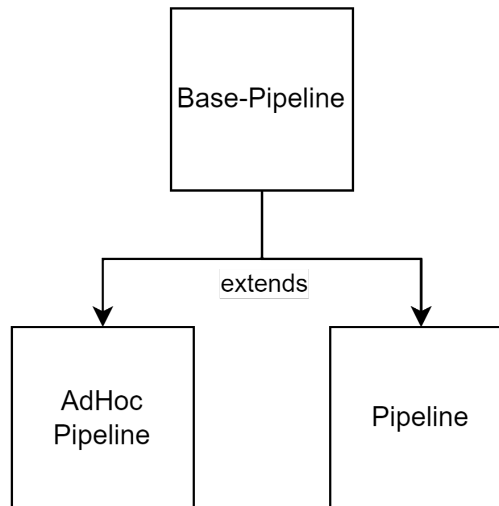


**Figure 5.13:** Pipeline inheritance

initialize the editor page were assessed.

One option would be to use the current location object of the React app to pass on information between pages. This would reduce queries that are executed in the backend and lead to an overall faster rendering when the page is first visited. The other option would be to freshly initialize the webpage by passing the project ID to the page component and query the remaining data directly from the backend. This option would ensure the latest data to be pulled from the database, and the possibility to access the editor page from different contexts without the need to prefetch any data. The downside of this option would be more queries to the backend, as well as reduced responsiveness if the server is running slowly.

I followed the second option as this would reduce the necessary state management and allow an easier development of the page. This approach is also more in line with best practices for React development, and thereby the most suited option.

### 5.2.6 Sink storage

There were three possible approaches on handling the jayvee model results from the ad-hoc run.

Option one would be not store the result. This would reduce the amount of data that is produced by the JValue Hub. This approach, however, would not allow providing any feedback to the user other than the console output and limit the development of other new features with regard to visualizing and analyzing the output sink within the editor.

Option two would be to keep the latest model output in memory during user sessions. As we can not influence the amount of data that is processed by a single jayvee model, this approach might quickly lead to memory issues on either the server or client side. Another downside would be that, once the connection is lost, the model needs to be executed again.

Option three would be to keep the results the same as in a normal run and store it permanently. This would allow keeping the output across multiple sessions and evaluate the latest run whenever needed at the cost of additional file storage required.

I decided on the third approach as this needed the least amount of changes to the existing code. In addition, there will be supplementary features implemented for the code editor with regard to the model output. Though this is against the agile approach of only developing what is needed for the current feature, this is the most efficient option.

# 6 Evaluation of requirements

To verify my implementation and check if all requirements have been met, different approaches were followed. Both, functionality and final implementation were reviewed by key stakeholders of the JValue team. To test the functionality beyond a demo to the JValue core team, I tested different use and edge cases for the front- and backend, as well as implemented additional End-to-End tests to the existing test suite. Going through multiple instances of evaluation not only improves the quality of the end result but also ensures a high success rate in fulfilling all requirements that have been set. An overview of all requirements, i.e. which evaluation method verified which requirement and if any limitation to the fulfillment of the requirement were identified, can be found in table 6.1. Any limitation to the implementation are outlined in chapter 6.4.

## 6.1 Demo and code review

The entire architecture and code was thoroughly reviewed by a JValue core member. Any feedback was discussed, and the resulting improvements were implemented accordingly. In addition, a preview of the implementation and new functionality was demonstrated to the JValue core team. In a Q&A session, the implementation and integration into the existing architecture was explained in detail. Any feedback was implemented and then verified in a smaller subgroup of the core team.

## 6.2 End-to-End-Testing

To verify backend functionality, additional End-to-End (E2E) tests were implemented. The E2E tests do follow the current standard within the existing JValue Hub implementation. However, it should be noted that the test setup only focuses on basic functionality and does not test the functionality in detail. To achieve higher test coverage, additional tests are necessary.

| Req. ID | description | Coverage by | limitation |
|---|---|---|---|
| NF.R.1 | Use existing architecture | code review | - |
| NF.R.2 | Follow current implementation | code review | Yes |
| NF.R.3 | Reusable and clean code | code review | - |
| NF.R.4 | Follow design principles and general layout | demo | Yes |
| NF.R.5 | No side effects | FT + E2E | Yes |
| F.R.1.1 | Meaningful layout | demo | Yes |
| F.R.1.2 | Information on the latest run is displayed | demo + FT | Yes |
| F.R.1.3 | Execute runtime from editor | demo + FT | - |
| F.R.1.4 | The entire ad-hoc run is stored permanently | demo + E2E | - |
| F.R.2.1 | Layout for console output | demo | - |
| F.R.2.2 | Display console output | demo + FT | - |
| F.R.2.3 | Previous run result on page load | demo + FT | - |
| F.R.2.4 | Storage of console output from jayvee model | demo + E2E | - |
| F.R.2.5 | Additional run results information in extra tab | demo + FT | Yes |
| F.R.2.6 | Download of output sink | demo + FT | - |

**Table 6.1:** Functional requirements

## 6.3 Frontend testing

While E2E testing primarily focuses on backend functionality and does not take into account any unforeseen user activity, thorough frontend testing closes this gap and increases the confidence in the implementation. To ensure high code coverage during frontend testing, detailed test scenarios were defined beforehand and iteratively tested during the development process to locate side effects to the existing implementation early on and fix them directly. The test scenarios can be seen in appendix A.

## 6.4 Limitations

The overview of limitations can be seen in table 6.1

### NF.R.2 Follow current implementation

After the first iteration of code review, some minor changes were implemented. At the time of submission, a final and complete review is still outstanding. Therefore, this requirement is still under evaluation.

### NF.R.3 Reusable and clean code

Considering that the newly introduced component 'AdhocRunPanel' shows similarities with the RunRow component, a joint component would be possible and increase readability. In addition, further endpoints could have been described to allow a more direct access to backend services and improve readability of the editor page implementation. However, this would have been in contrary with NF.R.2. and was therefore not done. If more functionality is added in the future, this might be a good opportunity to maintain readability with increasing complexity.

### NF.R.4 Follow design principles and general layout

The Monaco Editor component that is used for the code editor of the JValue Hub is not fully in line with the JValue Design in general. Currently, there is a custom configuration provided by the jayvee team. This configuration could be extended to perfectly embed with the JValue design.

### NF.R.5 No side effects

The implementation itself does not show any side effects with the existing functionalities of the JValue Hub. Over the lifetime of the Hub and with growing functionalities of the jayvee language, more complex pipeline models might be developed that require many iterations of development. This will increase the resources required by the editor and ad-hoc pipelines and might lead to implications on the remaining services.

### F.R.1.2 Information on the latest run

The output of the jayvee run with additional information is displayed on the editor page after the run has finished. The fundamental requirement is fulfilled. One improvement could be to implement a dynamic and continuous stream of the output during the execution of the jayvee pipeline model to the frontend to display data while it is produced.

### F.R.2.5 Additional run results information in extra tab

Additional run result information is displayed in the information panel at the bottom of the screen. The information displayed could be extended and include

additional information as well as configuration options, i.e. for the run time variables.

# 7 Conclusion and outlook

In conclusion, this thesis has revolved around the design and implementation of a web-based code editor for the JValue Hub. The primary focus has been the integration of an ad-hoc runtime as a new feature within the existing pipeline editor. Throughout this project, latest industry standards related to designing web interfaces for code editors were examined.

By adopting an agile methodology, project requirements were defined during the development process and continually evaluated through various sources. Successfully achieving the initial goal of incorporating the ad-hoc runtime was a significant milestone. However, it is clear that in order to offer an exceptional user experience when it comes to developing jayvee models from the JValue Hub, additional features and improvements to the web editor are essential.

One area of improvement involves the handling of output data. While implementing the option to access the output from the editor page is a significant improvement for the development experience of the editor, there is potential to further expand the functionality by implementing options to evaluate the output sinks from within the editor page. Also providing runtime variables through the editor itself would increase the development experience as more influence over the output would be granted.

Additionally, the following performance optimizations could improve the editor and ad-hoc runtime feature in the future. To reduce data storage, periodically sweeping saved ad-hoc pipelines would reduce the amount of data drastically. Currently, there is no possibility to stop or pause any given run. This feature would allow for better steering of the required resources, as multiple parallel runs of the same model could be prevented. It would set up the development of further features that interact with an ongoing run.

This thesis represents another foundational step in the broader journey towards developing a fully functional Online Editor for Data Pipelines in the JValue Hub. The ultimate objective is to become a competitive alternative in a market already populated by established players.

# Appendices

# A Test scenarios

## A.1 Testing existing functionality

**Table 1:** Overview of test scenarios for existing features

| Scenario | expectation |
|---|---|
| On project page | |
| create pipeline | pipeline is created for the specific version of the pipeline |
| On pipeline page | |
| create run | a new run linked to the pipeline is triggered and stored |
| "running" | "running" status is indicated |
| completed | on error, error status is indicated<br>on success, success status is indicated |
| run success complete | download of pipeline model sink provides the expected file |
| In database | |
| ProjectPage - create pipeline | new pipeline information entity is created<br>new instance is created |
| PipelinePage - createRun | new run information entity is created<br>new run entity is created |

## A.2 Testing new functionality

**Table 2:** Overview of test scenarios for new features

| Scenario | expectation |
|---|---|
| Navigate to page | |
| logged in as Project lead | show editor with model from current version<br>previous run information is displayed in console |
| is not Project lead | empty editor, with indication "not authorized" |
| Run completed | |
| with run success | jayvee model output is displayed in console<br>information on run are displayed in second tab |
| with run error | error message is indicated in console |

| Continuation of Table 2 | |
|---|---|
| Scenario | expectation |
| | jayvee model output is displayed in console<br>information on run are displayed in second tab |
| Run is triggered | |
| no run is triggered | new ad-hoc pipeline is created and run is initiated |
| while test is running | new run is started<br>success/error are displayed for new run<br>new ad-hoc pipeline is created and run is initiated |
| in database | |
| EditorPage - create ad-hoc Run | new ad-hoc pipeline information entity is created<br>new instance is created |

# B Editor overview

| IDEs | | | | | |
|---|---|---|---|---|---|
| # | Editor | code position | console position | customizable | link & comment |
| 1 | Visual Studio | centered | bottom | yes | https://visualstudio.microsoft.com/downloads/ |
| 2 | VS Code | centered | bottom | yes | https://code.visualstudio.com/docs/?dv=win |
| 3 | Eclipse | centered | bottom | yes | https://www.eclipse.org/downloads/ |
| 4 | pyCharm | centered | bottom | yes | https://www.jetbrains.com/de-de/pycharm/ |
| 5 | Android Studio | centered | bottom | yes | https://developer.android.com/studio |
| 6 | IntelliJ | centered | bottom | yes | https://www.jetbrains.com/de-de/idea/download/?section=windows |
| 7 | NetBeans | centered | bottom | yes | https://netbeans.apache.org /front/main/download/nb19/ |
| 8 | Xcode | centered | bottom | yes | https://developer.apple.com/xcode/ |
| 9 | RStudio | centered | bottom | yes | https://posit.co/download/rstudio-desktop/ |
| 10 | Sublime Text | centered | none | yes | https://www.sublimetext.com/3 |

**Table 3:** IDE overview

| ODEs | | | | | |
|---|---|---|---|---|---|
| # | Editor | code position | console position | customizable | link & comment |
| 1 | JSFiddle | centered | bottom | yes | https://jsfiddle.net/g6Lu0yro/3/ |
| 2 | PythonAnywhere | centered | bottom | yes | https://www.pythonanywhere.com/ |
| 3 | Codio | centered | bottom | yes | https://www.codio.com/<br>different use case |
| 4 | Koding | – | – | – | https://www.koding.com/login/<br>no longer maintained,<br>most prominent successor is Codeanywhere |
| 5 | DartPad | centered | right | yes | https://dartpad.dev/? |
| 6 | Repl.it | centered | right | yes | https://replit.com/ |
| 7 | Ideone | centered | bottom | no | https://ideone.com/J2uVks |
| 8 | Cloud9 AWS | centered | bottom | yes | https://aws.amazon.com/cloud9/ |
| 9 | Goorm | centered | bottom | yes | https://ide.goorm.io/ |
| 10 | Codeanywhere | centered | bottom | yes | https://codeanywhere.com/ |

**Table 4:** ODE overview

# References

GitHub. (2023). https://pypl.github.io

Heltweg, P., & Riehle, D. (2023). A systematic analysis of problems in open collaborative data engineering [Just Accepted]. *Trans. Soc. Comput.*, 24. https://doi.org/10.1145/3629040