NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

MARTIM BIXIRÃO FERREIRA ANDERSEN

BSc in Computer Science

A NO-CODE INTERFACE FOR DATA EXTRACTION FROM HETEROGENEOUS DATA SOURCES

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
October, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

A NO-CODE INTERFACE FOR DATA EXTRACTION FROM HETEROGENEOUS DATA SOURCES

MARTIM BIXIRÃO FERREIRA ANDERSEN

BSc in Computer Science

Adviser: Bruno Moscão
Software engineer, Skills Workflow

Co-adviser: Sérgio Marco Duarte
Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
October, 2023

A No-Code Interface for Data Extraction from Heterogeneous Data Sources

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.17) [1].

Acknowledgements

Once, Hellen Keller said: "Alone we can do so little; together we can do so much.". I believe the success of this master's thesis was only possible because of the help of many people.

First, I would like to express my sincere gratitude to my adviser, Bruno Moscão, as well as to João Paulo Vieira, for their continuous support, patience, and guidance throughout this work, and for their motivation and enthusiasm during our bi-weekly meetings. I am also grateful to Pedro Fernandes for assisting me during the first semester and the initial stages of implementation. Moreover, I would like to thank Hélder Barreiros for suggesting the use of a tool (JSON Editor Online [2]) to facilitate the development of the prototype. Finally, I would like to acknowledge the support during the first phase of this master's thesis from other colleagues at Skills Workflow, namely Eduardo Pinto and Fernando Romano.

Secondly, I would like to thank my co-adviser, Professor Sérgio Marco Duarte, for his valuable advice regarding the development of this document. Additionally, I would like to express my gratitude to Professor João Lourenço for providing the NOVAthesis LaTeX template [1].

Finally, I want to extend my heartfelt thanks to my family and friends for their unconditional support and encouragement during this work.

Abstract

Skills Workflow is a company that sells a platform enabling companies to build custom micro-applications. These are called workspaces and are built using a set of widgets (charts, tables, calendars, kanbans, Gantts, among others). Additionally, Skills Workflow, provides a marketplace of pre-built micro-applications for various domains, like outsourcing, marketing, project management, and more. But, even with all the workspaces provided, customers usually need specific solutions that are not available in the marketplace.

Ideally, customers lacking programming skills should have the capability to extract information into their own workspaces. Often, this information originates from heterogeneous data sources. This thesis presents a no-code interface that enables other users to supply the data utilized by customers in constructing their workspaces. This interface relies on a set of features (back-end) that can perform the extraction efficiently and with effectiveness - at high speed and in a scalable way.

The solution comprises an interface with different boxes that can be dragged, dropped, and interconnected. Each box opens a specific component, which can be easily configured to extract, aggregate, filter, or store data. There is also the possibility of defining a recurring execution schedule. The interface is complemented by a set of services that compile and execute the tasks defined by the user.

Keywords: No-code, Low-code, Data extraction, Heterogeneous data sources

Resumo

A Skills Workflow possui uma plataforma que permite às empresas construir micro-aplicações customizadas. Estas são chamadas de ”workspaces” e são construídas recorrendo a um conjunto de ”widgets” (gráficos, tabelas, calendários, kanbans, gantts, entre outros). Para além disso, oferece um ”marketplace” com microaplicacões pré- construídas para diversos domínios, como ”outsourcing”, ”marketing”, gestão de projetos, entre outros. Frequentemente, os clientes têm necessidade de soluções específicas que não estão disponíveis no ”marketplace”.

Idealmente, os clientes sem conhecimentos de programação deveriam ter a capacidade de extrair dados para os seus próprios ”workspaces”. Muitas vezes, esses dados vêm de fontes de dados heterogéneas. Esta tese apresenta uma interface sem código que permite que outros utilizadores forneçam os dados utilizados pelos clientes na construção dos seus ”workspaces”. Por detrás desta interface (”back-end”), as funções implementadas são capazes de realizar a extração de forma eficiente e eficaz - em alta velocidade e de forma escalável.

A solução consiste numa interface com diferentes caixas que podem ser arrastadas, soltas e interligadas. Cada caixa abre um componente específico, que pode ser facilmente configurado para extrair, agregar, filtrar ou armazenar dados. Existe também a possibilidade de definir um horário de execução recorrente. A interface é complementada por um conjunto de serviços que compilam e executam as tarefas definidas pelo utilizador.

Palavras-chave: Interface sem código, Interface com pouco código, Extração de dados, Fontes de dados heterogéneas

Contents

List of Figures

Acronyms

AST Abstract Syntax Tree *(pp. 26, 33, 62)*

BERT Bi-directional Encoder Representations from Transformers *(p. 8)*

ETL Extract-Transform-Load *(pp. 13, 14, 30, 31)*

NLP Natural language processing *(pp. 7–9)*

OEM Object Exchange Model *(pp. 18, 19)*

RAM random-access memory *(p. 17)*

RDBMS relational database management system *(p. 10)*

SSMS Microsoft SQL Server Management Studio *(pp. ix, 10, 11, 40)*

UI user interface *(pp. ix, 1, 2, 4, 16, 28, 30, 35–37, 39, 50, 51, 60, 61, 63)*

1 Introduction

1.1 Context and motivation

The goal of this thesis is to develop a solution for companies to be able to extract data from different data sources without needing people with programming knowledge. In the future, the interface used for this task can be adapted to be used by companies that want to build their own workspaces[1] without needing to hire software engineers.

This solution consists of a no-code UI[2] that abstracts users of techniques to extract information from multiple data sources, at high speed and in a scalable way.

This work is revolutionary in the sense that companies no longer need to face the challenges of finding the right pre-built solutions in the marketplace. Instead, they can utilize the Skills Workflow's platform to build their own customized workspaces. In fact, anyone can use the platform, regardless of their programming knowledge.

Certainly, a no-code UI can be very beneficial for companies, because it can enable faster development. Besides, the development can be cheaper because there is no need to hire software engineers do develop workspaces. This interface can help not only people without programming expertise, but also programmers, since the latter may focus their time on other tasks.

1.2 Problem definition

Nowadays, organizations extract data from a wide variety of data sources and, consequently, they need to use different techniques to do it. Besides, the inherent challenges of this task are even greater when we intend to solve it with a no-code UI.

To extract, aggregate, filter, and store data we need to write queries, use various libraries, make endpoint calls, and write several lines of code. This thesis aims to provide a no-code UI that abstracts users of these techniques. One of the goals of this work is to

[1] A workspace is a set of widgets (charts, tables, calendars, kanbans, gantts, among others) that presents information the user requests.

[2] A no-code UI is an intuitive interface that can communicate with the back-end of an application without forcing the user to code instructions.

replace the process of writing queries by utilizing a visual query builder with a no-code UI. This involves creating an intermediate representation of a query that can be translated to any query language.

The extraction of large quantities of data brings some challenges if we want to do it with as little latency as possible.

GraphQL is a query language used in the Skills Workflow's platform and has a big limitation: it is not able to aggregate data from multiple records or rows into a single value, like making a sum or calculating a maximum. Moreover, its syntax is not universal and varies according to its GraphQL server implementation.

It is important to mention that we can consider three types of user of this product: the Super-creator, the Creator and the End-user. The Super-creator is responsible for extracting, aggregating, filtering, and storing data. This is all then presented to the Creator, who creates the workspaces. Finally, the End-user can consume the information of these workspaces.

To be clear, the Super-creator can be someone paid by Skills Workflow to do the job. Alternatively, consider a scenario where a company's team composed of ten people intends to build its own workspace. One member of the team can perform the role of Super-creator. In fact, even a Creator can perform that if he thinks some information is missing on his side.

We intend to offer a user-friendly interface to the Super-creator to help him with his task. That is, if he wants to make, for example, an aggregation, he should do it by simply building a diagram with boxes. Nevertheless, he still needs to know some details about the data source, like the query language it uses. This justifies the presence of "low-code" in the keywords' list of this thesis, besides "no-code". On the other hand, the Creator does not need to have programming skills at all. To perform his tasks, he uses a no-code UI.

1.3 Main contributions

Regarding my contribution to the Skills Workflow's platform, I believe that the proposed prototype is of great value to the company as it is a great step forward in the direction of a unique solution on the marketplace that enables anyone to build their own workspaces. There is no other existing solution that provides exactly everything this one does when it comes to extracting data from heterogeneous data sources with a no-code approach.

The solution contains a no-code UI with six graphical components. One component allows the design of a diagram with boxes with buttons that, when clicked, open other components. These components contain input forms, buttons, drop-down menus, and other elements that enable the user to configure the options for specific tasks, such as extracting, transforming, and storing data from heterogeneous data sources. Furthermore, there are two services responsible for compiling, executing, and scheduling tasks. There is also a way of dealing with the extraction of large quantities of data, as it is explained in more detail in subsection 3.6.3.

Put together, all these components and services contribute to the uniqueness and innovation of this work.

1.4 Document outline

This document is divided into six main parts: Introduction, State of the Art, Concept, Implementation, Validation, and Future Work and Conclusions.

Firstly, the Introduction chapter presents background information on the topics being discussed and explains why they are relevant. Then, the problem is defined and the following section states the innovations this product provides.

Secondly, the State of the Art is divided into several sections. In this chapter, we find definitions that help to understand the challenges of this project. We also present many approaches to solve the problems previously stated. The final section of this chapter includes projects with similarities to this one.

Next, the Concept chapter provides an overview of the solution concept from a broad perspective. Here, we give a brief explanation of the main components of the interface, as well as the services involved in the back-end.

The Implementation chapter describes the implementation details of the solution. The major steps of the development are layed out in a step-by-step order, enabling the reader to track the progress and understand the choices made along the way.

The Validation chapter offers both qualitative and quantitative assessments of the prototype, determining whether it is of high quality.

Finally, the Future Work and Conclusions chapter presents the possible improvements to be made in the future and the conclusions of this thesis.

2

State of the art

This chapter is divided into several sections. The first one gives an introduction to GraphQL, as it is mentioned many times in the document. The second part is related to natural language processing, which is included in the document because it is a field of artificial intelligence that needs to be understood in order to add intelligent functionalities to the visual query builder. The following section of the state of the art is related to the no-code UI. More specifically, it presents the topic of transforming natural language into query language. It also talks about some well-known visual query builders, as well as the challenges of implementing one. Next, there is a section for data extraction from heterogeneous data sources, which contains an explanation to why data sources are called *heterogeneous* and why they exist in the first place. It also includes information about the most common used techniques when dealing with this type of data sources. The section that comes next shows some approaches to solve the problem of large dataset extraction. The last section of the state of the art is the related work, which presents projects similar to this one.

Initially, we believed that complementing this tool with functionalities powered by artificial intelligence, like creating queries from natural language, would be a good idea. However, after some research, we realized that this is not a trivial task. So, we decided to focus on a prototype without these functionalities. Nevertheless, this is a topic that is worth mentioning in this document, as it can be explored in the future. We kept the research on this chapter and not on the last one because it is relatively extensive and contains several definitions.

2.1 GraphQL

GraphQL, short for Graph Query Language, is an open-source data query and manipulation language for APIs. It was developed by Facebook in 2015 and released to the public in 2021 [3].

The company Skills Workflow opted to use GraphQL in their projects and now, it faces some new challenges, but also several perks that did not exist if it chose REST, for example.

2.1.1 Advantages

When fetching data using a REST API, a common issue is over-fetching and under-fetching data. Sometimes, one request may return more data than the user needs (over-fetching). On the other hand, the user may be obligated to make more than one request to get the data he wants. GraphQL solves this problem, because the user can specify exactly the fields he wants [3]. The Figure 2.1 shows an example of a request - name and email of the user whose id is 1 - and its output when using a REST API (left side) and a GraphQL API (right side). We can see that, on the left side, the response includes unwanted fields, opposed to what happens on the right side - only the name and the email are returned.

```
# REST API

# Request
GET /users/1

# Response
{
  "id": 1,
  "name": "Martim Andersen",
  "email": "martimandersen@gmail.com",
  "password": "password1234",
  "birthDate": "28/07/2000",
  "address": "Lisboa, Portugal"
}
```

```
# GraphQL API

# Request
query {
  user(id: 1) {
    name
    email
  }
}

# Response
{
  "data": {
    "user": {
      "name": "Martim Andersen",
      "email": "martimandersen@gmail.com"
    }
  }
}
```

Figure 2.1: Over-fetching in REST API vs in GraphQL. Scenario where name and email of the user whose id is 1 is requested.

GraphQL provides an intuitive syntax [3]. The Figure 2.2 shows how much more complex the syntax of a query written in SQL can be, compared to its equivalent GraphQL query. More specifically, the query retrieves information about users, including their name, email and their posts.

Another advantage of using GraphQL is that developing applications can become easier. For example, when something is changed in the front-end of an application, usually the data needed increases or becomes more specific. This means that, when using REST, more endpoints need to be added in the back-end. This does not happen when GraphQL is used [4].

Furthermore, GraphQL does not restrict applications to use a specific programming language or storage system. This results in a consistent interface for product development and a versatile platform for building tools [3].

```sql
# SQL

SELECT users.name, users.email, posts.title, posts.content
FROM users
JOIN posts ON users.id = posts.author_id;
```

```graphql
# GraphQL

query {
  users {
    name
    email
    posts {
      title
      content
    }
  }
}
```

Figure 2.2: Complexity of syntax for a query written in SQL vs in GraphQL.

2.1.2 Limitations

One problem with GraphQL is that it lacks built-in aggregation support. For instance, it doesn't natively offer functionality for simple count-by operations. These operations need to be implemented in specific resolvers on the GraphQL server.

Similarly, the availability of filtering depends on the GraphQL server's implementation. Moreover, unlike SQL where all query filters utilize the same standardized syntax, the filtering syntax in GraphQL is not uniform [5]. For instance, let's consider a scenario where we have several users stored in a database and we want to retrieve the information of a certain user working in a given department. As we can see in the Figure 2.3, the syntax for the same query is different. The query on the left uses the syntax supported by the Skills Workflow's GraphQL server, whereas the one on the right uses the Apollo Client's [6] filtering syntax.

```graphql
query {
    users (where: { department: { eq: "Development" } }) {
        name
        id
        avatar
        department
    }
}
```

```graphql
query {
    users (department: "Development") {
        name
        id
        avatar
        department
    }
}
```

Figure 2.3: Example of one query with two different filtering syntaxes.

In the context of this thesis, where we are required to extract data from multiple sources, including accessing various GraphQL servers, this can be problematic. We need to know the syntax supported by each one to be able to filter the data. Plus, we need to be able to offer support for aggregations. Naturally, the user of this no-code interface must be abstracted from all these concerns.

2.2 Natural language processing

Natural language processing (NLP) is a field of artificial intelligence. It is related to complex language-related tasks. For example, machine translation[1] is one application of NLP. Other applications are question-answering, summarization, paraphrasing and email spam detection. The analysis of human language to solve these tasks brings several challenges. Therefore, to solve these advanced problems, NLP requires designing and putting into practice models, systems, and algorithms [7].

NLP can be divided into two parts: natural language understanding/linguistics and natural language generation. The first one includes phonology, morphology, pragmatics, syntax and semantics. The second one involves the generation of sentences by a computer [8].

2.2.1 Challenges

The demand for new tasks that need to be solved using NLP has been growing daily and consequentially so are its inherent challenges. For example, one word can be used in different contexts and have different meanings. On the other hand, there are different words that express the same idea. Sarcasm, irony and sentences with double meaning sometimes can be easily interpreted by us, humans, but not by a computer. Furthermore, there are several expressions that are specific to a certain culture, region or domain that can make the job of language models harder. Finally, misspelled or misused words can also comprise a problem [8].

Moreover, it is crucial that the accuracy is not affected when we try to build lighter models. There are several methods for model compression such as pruning, quantization, low-rank approximation, knowledge distillation and neural architecture search [9].

It is also important to solve these tasks while consuming as little energy as possible [9].

2.2.2 Language models

Naive Bayes classifiers are probabilistic models that predict the tag[2] of a text. They can be trained very quickly and have an easy implementation. Plus, their performance is relatively good. They are used in some tasks in NLP, such as segmentation, translation, spam filtering and emotion detection. There is a big limitation here: if a certain word that was not included in the training data and is present in the test data, the probability would be zero [8].

Hidden Markov Models are statistical models used to infer the next hidden state, by observing the previous states. These models have many applications, like speech recognition, word prediction, bioinformatics, etc [8].

[1] The process of translation of text from one language to another using artificial intelligence without human intervention.

[2] Label given to a word or expression that represents its meaning. For example, a noun, verb, organization, location, positive, neutral, etc.

In recent years (late 2000s and early 2010s), the popularity of neural networks has been increasing. This is due to the evolution of computers and their ability to manage larger quantities of data. The emergence of several open-source libraries such as Tensorflow, Pytorch is also a decisive factor. Neural networks revolutionized NLP. They are based on the human brain, more specifically on the neurons, which compute and produce information between each other. With these models, the word embedding (representation of a word) is made using vectors. Neural networks can infer new states.

Bi-directional Encoder Representations from Transformers (BERT) is a transformer-based model trained on a huge amount of unlabeled text from BookCorpus[3] and English Wikipedia. The transformer architecture - a neural network architecture - was introduced in 2017 by Google researchers. This architecture considers the relative weights of various input components while producing a prediction. This mechanism is called "self-attention". Recurrent neural networks are examples of conventional neural network architectures that process input sequentially. On the other hand, with the transformer architecture, the model can process the full input at once. This makes it ideal for NLP jobs, where the meaning of the words matters more than their order [10]. It is called "bidirectional" because BERT offers the ability to derive the meaning of a word by looking at both the previous and the following words. This model can be fine-tuned and then used in tasks of NLP, like question answering, sentiment analysis, text classification, sentence embedding, ambiguity interpretation, among others. BERT has a big limitation: it was trained on a maximum sequence length of 512 tokens[4]. So, during fine-tuning, the model can only handle inputs with a maximum length of 512 tokens. This can cause truncation which possibly leads to the loss of context and meaning. Variations of BERT like RoBERTa, ALBERT, and T5 have a longer maximum sequence length [8].

GPT (Generative Pre-training Transformer) was developed by OpenAI and released in 2018. It combines the Transformer architecture with unsupervised pre-training. This model, before fine-tuned, was pre-trained on a huge dataset with 117 million parameters[5]. In the following year, with the introduction of GPT-2 - GPT's successor - the maximum input length of 512 tokens increased to 1024. Plus, the number of learnable parameters became 1.5 billion. And in 2020, with GPT-3 it increased to 2049 tokens and to 175 billion parameters. This last one was trained on a dataset of 570 GB of text data. This model can perform NLP tasks effectively under zero-shot and few-shot settings (without and with little fine-tuning, respectively). However, there is a concern mainly for GPT-2 and GPT-3: sometimes they can generate highly convincing, but misleading information. This is because they were trained on a big amount of data that comes from the internet, where there is some false information [7, 9].

[3]Collection of free novel books written by unpublished authors, containing 11,038 books of 16 different sub-genres.

[4]Sequence of characters that represents a single meaningful unit of text. Usually, a token corresponds to a word.

[5]Number of values that are learnt during training used to make predictions. The values can be the weights and biases of the model's neural network architecture.

2.3 No-code UI

This section is important because one of the key points of this prototype is its interface which should be as intuitive as possible for people with no programming skills.

2.3.1 Natural language to query language: OpenAI Codex

2.3.1.1 Definition of Codex

Codex is the more specialized descendant of the general-purpose language model GPT-3. While GPT-3 can perform a wide variety of NLP tasks, Codex excels at producing code from an input of natural language. Besides, given an input code it can explain it in natural language. It is also able to translate code from one programming language to another. Codex is specialized in Python, but is also very capable to use other languages such as C#, JavaScript, Go, Perl, PHP, Ruby, Swift, TypeScript, SQL, and even Shell. In addition to the same training made for GPT-3, Codex was trained on more 159 GB of Python code from 54 million GitHub repositories. GitHub Copilot[6] uses the OpenAI Codex [11].

2.3.1.2 Models from OpenAI

Although the models below are built on the GPT-3 architecture, they have been pre-trained using various datasets, enabling them to produce text in a specific style or domain.

Code-davinci-002 and code-cushman-001 are the two models currently available that belong to the Codex series and perform code generation tasks. The first one is the most capable one and specializes in generating code from inputs of natural language. It is also good at completing code. The second one is a bit less capable. On the other hand, it is slightly faster, which can make it better for real-time applications [12].

Text-davinci-003, text-curie-001, text-babbage-001, and text-ada-001 are models of the GPT-3 series developed by OpenAI that are used on text generation tasks. The first one is the most capable of this series and can perform, often better, any task the other models do. It can also make code completions. Text-davinci-003 is good at creative writing. The second one is a bit less capable than the first one but it is cheaper and faster. It is good for technical writing. As for the third one, it is cheap, fast and great for simple tasks, like writing code documentation. The last one is the best for the most simple tasks, like writing answers to questions. It performs them with the lowest cost and very quickly [12].

It is important to mention that these models from OpenAI are not available for free. Hence, there are alternatives presented below.

2.3.1.3 Fine-tuning and pre-training

Fine-tuning is the process of training a pre-trained model on a specific task, such as code generation. Pre-training refers to the initial training of a model on a dataset, before

[6]This tool is installed on an IDE and suggests lines of code in real-time.

fine-tuning it for a specific task [13].

These processes are used to improve the performance on a specific task. Pre-training is useful to reduce the amount of data needed for fine-tuning [13].

We can fine-tune a model using our own dataset of natural language inputs and corresponding outputs [12].

2.3.2 Natural language to query language: TensorFlow and PyTorch

As it was mentioned before, there are free alternatives for Codex: it is possible to use an open-source machine learning library that provides free pre-trained models, like Hugging Face, TensorFlow or PyTorch. However, with the first one, we need to pay to deploy the fine-tuned models in a production environment.

None of these libraries have models that are specialized in transforming natural languages into GraphQL queries. Nevertheless, they have some that are specialized in transforming natural language into SQL queries. So, we can fine-tune a pre-trained TensorFlow or PyTorch model on a dataset of natural language inputs and corresponding GraphQL queries. This process is called transfer learning.

In summary, TensorFlow is a deep learning framework, that has a large community and several tools and libraries, making it suitable for large-scale projects and production deployment. The developers of TensorFlow chose to prioritize having more options for customization over user experience. On the other hand, PyTorch has a more intuitive API. It is more flexible and more suitable for small to medium-scale projects and research purposes [14].

Finally, it is important to say that paid models like the ones developed by OpenAI are usually more capable than free models from TensorFlow and PyTorch. This is mainly because they tend to be trained on more data. Nevertheless, to understand the performance of each model, it is best to test it with our dataset.

2.3.3 Visual query builder

2.3.3.1 Microsoft SQL Server Management Studio

SSMS was first released in 2005 and it helps developers by offering an intuitive graphical interface with several tools to manage Microsoft SQL Server databases, which, in turn, is a relational database management system (RDBMS) [15].

One of the tools included in SSMS is the visual query builder, which presents 4 panes: the Diagram, the Criteria, the SQL and the Results panes. The Diagram pane shows rectangles with the selected (the ones being queried) tables or table-valued objects[7]. Inside each rectangle are the columns of a table or table-valued object. The lines that connect rectangles represent joins. The Criteria pane specifies the available options like which columns should be in the output, how the grouping should be made, what filters

[7] Database objects that return a table as a result of a query.

should be added, among others. The SQL pane contains the equivalent query in SQL for the diagram built in the Diagram pane. This query can be manually edited. Finally, the Results pane simply show the output for the query [15]. These four panes are presented in the Figure 2.4.

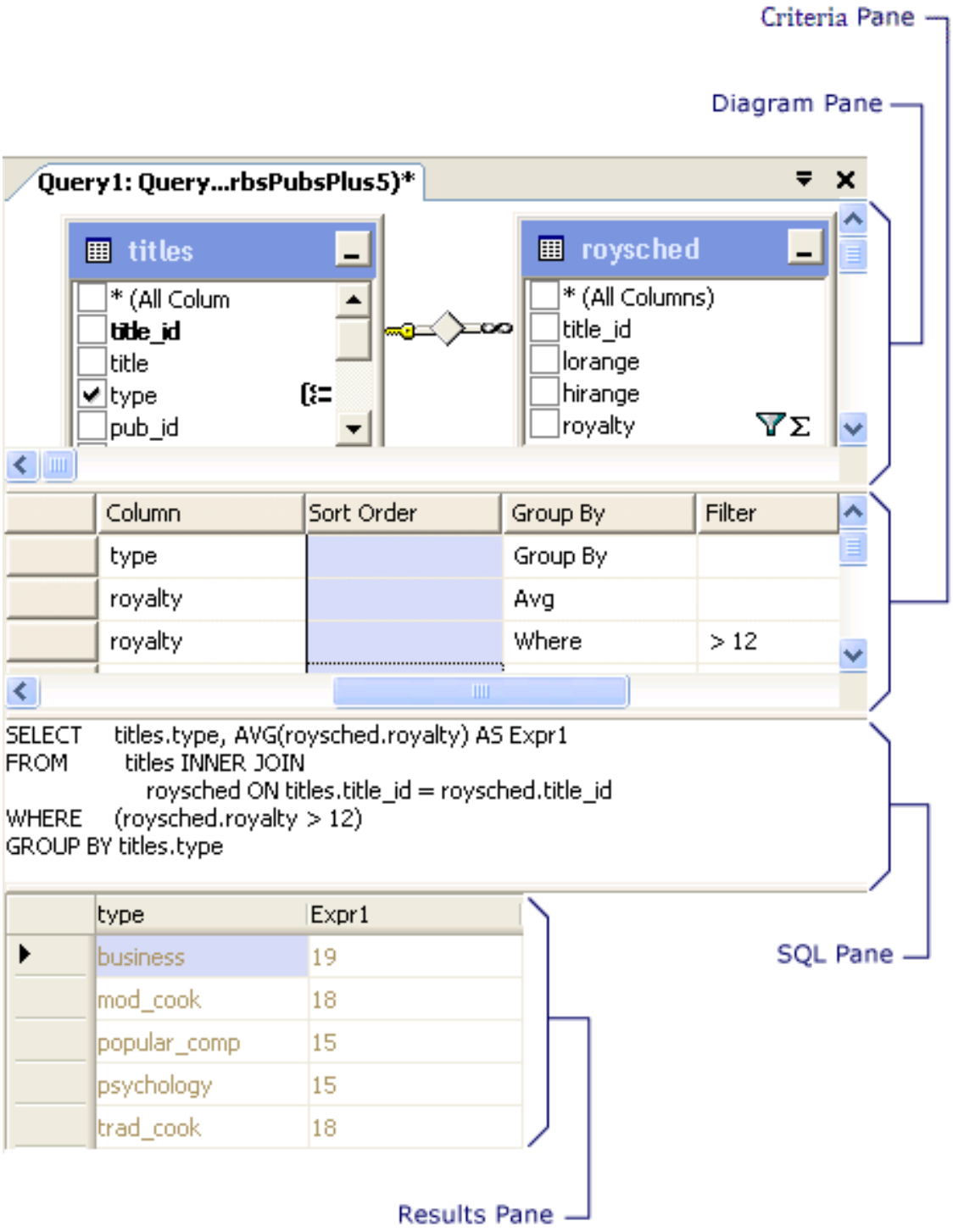


Figure 2.4: The four panes of SSMS (Adapted from [15]).

2.3.3.2 GraphiQL

GraphiQL is a tool with a user-friendly interface to explore the schema, test queries, and inspect the corresponding responses. It is mainly used by developers because it is not really a no-code visual query builder. To write a query, the developer needs to have knowledge about GraphQL, since GraphiQL only provides auto-complete on fields, gives some simple hints and highlights errors [4]. The Figure 2.5 shows the interface of GraphiQL.

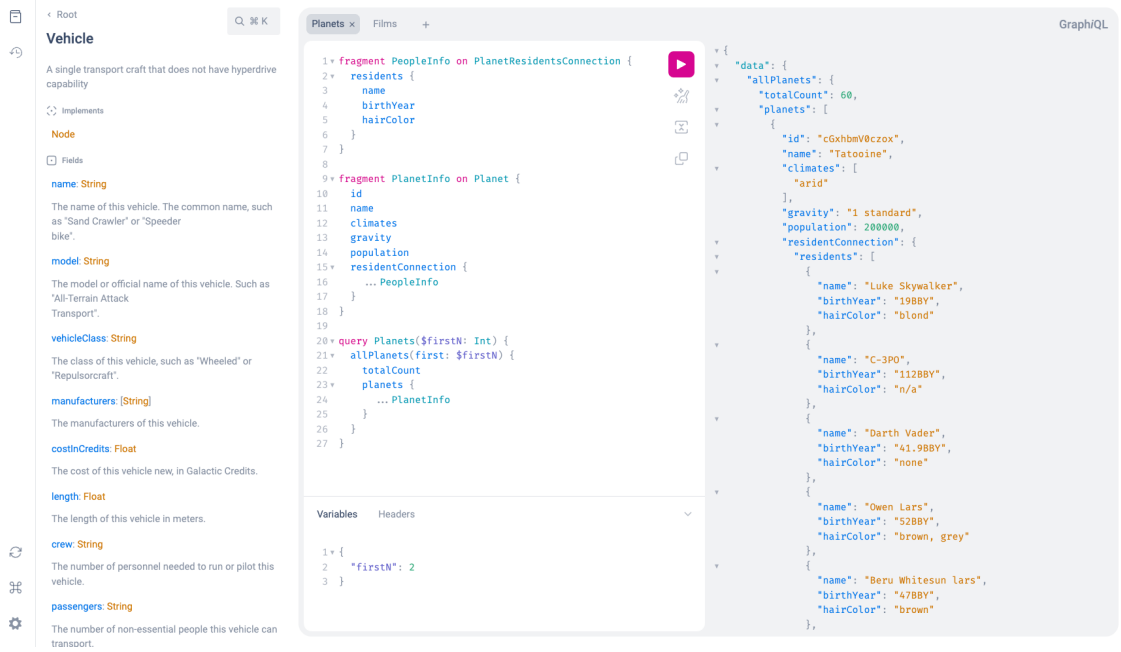


Figure 2.5: A graphical interactive in-browser GraphQL IDE (Adapted from [16]).

2.3.3.3 Visual Query Builder challenges

Firstly, it is necessary to be aware of the syntax similarities on all query languages. These make a generic no-code query representation possible. For instance, to construct a query, it is always required to specify the selected tables or fields. In addition, a query may involve filters, logical conditions, and aggregations. Thus, the main challenge is to create an intermediate representation that is generic enough to be used for all query languages.

Another big challenge of building a graphical component for constructing queries is the inclusion of GraphQL generation. GraphQL is relatively new. It was publicly released in 2015 so it is normal that it has less tools to help to work with it than, for example, SQL does, which was released in 1979.

Compared to SQL, GraphQL is more flexible and more powerful, in the sense that it can handle more types of data sources. GraphQL can work with NoSQL databases (with a dynamic schema[8]), cloud services, search engines, file systems, etc. SQL usually

[8]This allows the structure of the data to change at runtime.

works with relational databases, where data is structured in tables. Thus, the relationships between data stored are very clear and it is less challenging for visual query builders to display them. Furthermore, the syntax of SQL queries is universal[9], while various details on the GraphQL queries are defined by the server implementation.

2.4 Data extraction from heterogeneous data sources

This section marks a new topic and it is important to first introduce the concept of heterogeneity in data sources, before presenting some approaches to deal with it.

Nowadays, information systems usually request data from multiple origins. Essentially, different data sources can have different technologies of storing data, such as relational databases, NoSQL databases, data lakes, cloud storage services, among others. Consequently, the way data is accessed from the exterior can vary - through SQL, GraphQL, API REST, API SOAP, and more.

So, efficiently extracting information from heterogeneous sources can be quite challenging and, throughout the years, there have been proposed many solutions regarding this topic, such as the Extract-Transform-Load (ETL) approach and federated systems. Another approach is building a mediation system based on mediators and wrappers [17].

2.4.1 First approach: extract-transform-load

ETL is a data integration process and, as the name indicates, it has three phases: *extract*, *transform*, and *load*. The first one consists in extracting data from different sources, such as databases, flat files (csv, xls, txt, etc), APIs, among others. The second phase is responsible for transforming the extracted data into a single format (JSON, for example). This can involve cleaning[10], mapping[11] and enriching[12] the data. Finally, the third phase consists of loading the transformed data into a target system, which can be a data warehouse[13], a data lake[14], a database, or a file system [18].

There are some limitations for some ETL tools. Firstly, it can be complex to set up and maintain an ETL system. For example, the *transform* phase may require the intervention of an expert to do some manual cleansing of the data. Secondly, this process can occur on a regular basis - each day, each hour, etc. So, it may happen that data shown is not up-to-date.

[9] It is true that some relational database systems have their own variations of SQL. However, the basic tasks like querying and modifying data remain the same.

[10] Remove inconsistencies in the data, like duplicate values or incorrect data types.

[11] Align the data from the source with the structure and format of the target schema, like renaming, splitting or combining columns, and converting data types.

[12] Add more information or context to the data.

[13] Repository that stores large amounts of structured data that comes from several sources. This data is usually destined for analytical processing.

[14] Repository that stores large amounts of structured and unstructured data that comes from several sources. The data is stored in its raw format and it is usually used in machine learning.

Furthermore, extracting, transforming and loading data can introduce latency[15]. Another limitation is the cost of implementation and maintenance if you choose a tool that is not open-source. There are also some issues regarding flexibility. For example, very few ETL tools can reliably process unstructured data. Besides, some tools may have some difficulties handling simultaneous data extraction from multiple sources [18].

In summary, this approach is suitable for systems that do not have many data sources and their corresponding data models are not too complex. Plus, it is a good option if data is not usually updated, which means that the ETL process does not need to be done too often.

2.4.2 Second approach: federated systems

A federated system is a system that aims to present data as if there is a single data source with one data model, where, in reality, there are many heterogeneous data sources. This system provides the users a single virtual view of data from multiple sources, without copying or moving data. When a user queries the federated system, the query is transformed into several queries, according to the underlying data sources. Then, each one is sent to the corresponding data source. In return, the system gets the results and, finally, it combines them into a single view, which is called *virtual* because it is not a physical structure - simply the output for the user.

This system usually has a three-tier architecture: the presentation, federation and foundation layers. The presentation layer refers to the interface the users of the system are going to interact with. Next, the federation layer provides a unified way of accessing data, like a single query language. Finally, the foundation layer refers to the data sources, which are normally complemented with wrappers [19].

The Figure 2.6 presents the architecture of a federated system.

Some federated systems support some optimizations, such as caching and data batching. Caching is a very important technique used by federated systems to store frequently used data locally. By avoiding the repeated retrieval of data from its sources, we reduce latency and reduce the load on the data sources. Data batching means bundling multiple separate requests into a single request, which can reduce the number of network round trips and, in consequence, improve performance. Plus, a federated system can be a good option if we have a large number of requests and if these are complex. This is due to the parallelization job made by the system.

However, there is a limitation regarding this approach: the system is as fast as the slowest response of all the data sources. That is, we can only get a final view when the system gets in return all the outputs from all the data sources involved. Instead, if we had all our data in a data warehouse (ETL approach), for example, we would not have this drawback.

[15] Nevertheless, it is possible to choose to extract either all or just a part of the data from each data source. Hence, one could opt to extract less regularly certain data, to avoid some overhead.

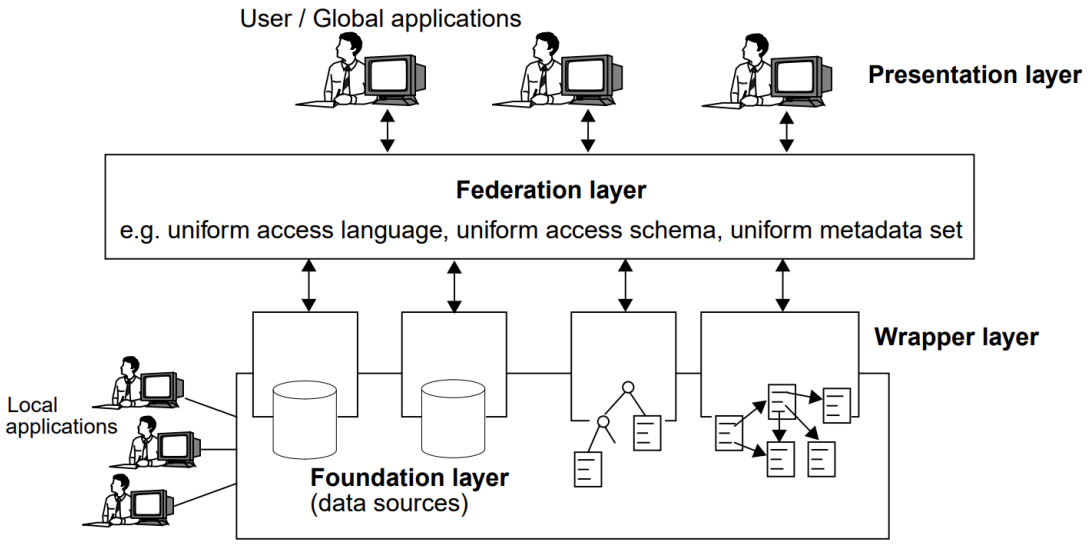


Figure 2.6: Architecture of a federated system (Adapted from [19]).

In summary, this solution is suited if we have multiple data sources with complex schemas. Besides, it can save us a lot of work related to cleaning and mapping data. Moreover, it is also a good option when we are dealing with large amounts of data and need a scalable (more data or even more data sources) solution.

It is important to mention Apollo Federation, which is an open-source GraphQL federation system that is used by several big companies, such as Adobe, Glassdoor, Netflix and Walmart. Apollo Federation enables the connection of various GraphQL APIs to a gateway server. This allows the access to data through one single port. It is also relevant to mention that it is possible to call a REST API with Apollo Federation [20].

2.4.3 Concepts of mediators and wrappers

This subsection provides essential definitions related to the third approach for handling heterogeneous data sources, which will be presented in the following subsection.

A mediator is a component that controls the flow of data between the user application and the different data sources. It abstracts both the representation and access to these sources by presenting a unified view of the data [17].

On the other hand, a wrapper is a component attached to a data source. This component performs a mapping of each local data model and transforms it into a global one - used by the other wrappers as well. Wrappers are also responsible for submitting queries on their corresponding data sources, as well as returning the results to the mediator [17].

The Figure 2.7 presents an architecture based on a mediator and wrappers.

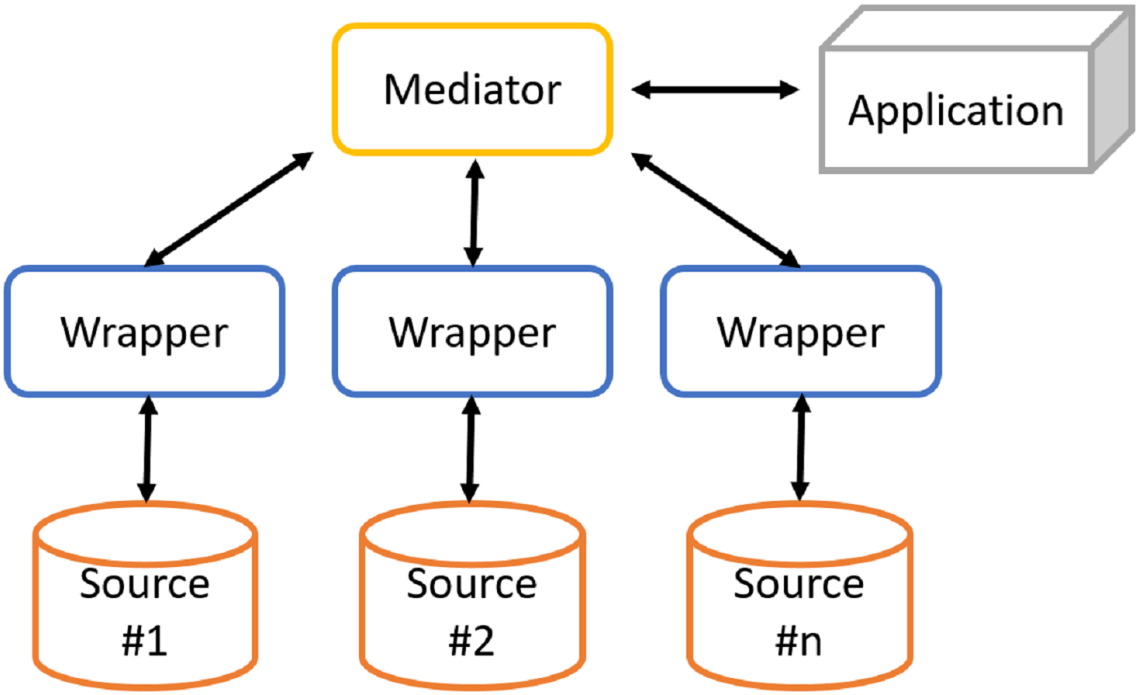


Figure 2.7: Architecture of a generic mediation system (Adapted from [17]).

2.4.4 Third approach: custom solution

Often times, organizations choose to build a custom solution when faced with heterogeneous data sources. This solution usually involves mediators and wrappers, which were introduced in the previous subsection. This process may not be automatic, but the user is abstracted from all the internal computations [17].

Now, I will briefly present the steps necessary to be taken for this approach. Firstly, the user provides the application a query as input. It is worth noting that this query can be constructed using a no-code UI. However, the system must translate it into a query with a certain language, like GraphQL or SQL.

Then, the system extracts the query's fields, using either custom code or an existing library.

The following step is mapping all the extracted entities to their corresponding data sources. This can be done using a data catalog, which is defined in the subsection below.

After this stage, each source's wrapper receives a sub-query (extracted from the user query) and transforms it into the language used in the corresponding local source. More specifically, there needs to be specific functions implemented manually that extract the arguments of the sub-query and creates the new one.

Finally, the results from all wrappers are merged to form a single output, ready to be presented to the user.

2.4.4.1 Concept of data catalog

A data catalog helps organizations discover, understand, and manage their data that comes from heterogeneous sources. It does so by storing metadata, like identification, location and content of the sources. It also maintains definitions include the meaning and structure of the data (data types, columns, attributes, etc). Besides, it saves information about the relationships and dependencies between the different data sources [21].

There are data catalog tools that offer an automatic implementation. For example, Apache Atlas [22] can be a good option for this and it has the advantage of being open-source. On the other hand, it is also possible to do it manually, using a dictionary, a hash table or a metadata repository.

2.5 Large dataset extraction

Large dataset extraction is the process of fetching big amounts of data. Naturally, the users want the data to be returned as fast as possible. There are several techniques to adopt when a lot of data is requested at once, such as caching, pagination, or even compression.

Firstly, caching is a technique that involves storing some data in a separate layer, where requests are answered faster than in the original storage location. This separate layered requires specific and usually expensive hardware, like random-access memory (RAM). As we can see in the Figure 2.8, there are several caching strategies with different layer locations [23]. In the context of this thesis, we are more interested in the database caching strategy. There are many algorithms we can implement, such as FIFO (First In, First Out), LRU (Least Recently Used), among others. The first one removes the oldest entry in the cache. The second one removes the least recently used entry in the cache.

Layer	Client-Side	DNS	Web	App	Database
Use Case	Accelerate retrieval of web content from websites (browser or device)	Domain to IP Resolution	Accelerate retrieval of web content from web/app servers. Manage Web Sessions (server side)	Accelerate application performance and data access	Reduce latency associated with database query requests
Technologies	HTTP Cache Headers, Browsers	DNS Servers	HTTP Cache Headers, CDNs, Reverse Proxies, Web Accelerators, Key/Value Stores	Key/Value data stores, Local caches	Database buffers, Key/Value data stores
Solutions	Browser Specific	Amazon Route 53	Amazon CloudFront, ElastiCache for Redis, ElastiCache for Memcached, Partner Solutions	Application Frameworks, ElastiCache for Redis, ElastiCache for Memcached, Partner Solutions	ElastiCache for Redis, ElastiCache for Memcached

Figure 2.8: Different caching strategies (Adapted from [23]).

Regarding pagination, it consists in splitting the data in portions, or pages. This solution is valid when the user does not need to see all the data at once. Thus, we avoid returning a big chunk of data in one go, reducing overhead on the server, network, or even client-side processing. Of course, to ensure a good user experience, we must provide clear navigation controls, as well as an intuitive ordered presentation of the data. Let's consider a scenario where we "deceive" the user by presenting an interface that forces

him to choose to fetch less data. For instance, consider a user wants to see all the git commits made in a year. Instead of presenting all commits at once, it is possible to split that information in commits per month. Then, the user is forced to choose a specific month.

Compression, on the other hand, reduces the size of the requested data by following a certain protocol and, consequently, minimizing the latency for serving requests. There are different compression algorithms, such as lossless compression and lossy compression. The first one preserves the data. ZIP is a popular lossless compression algorithm. On the other hand, the second one does not ensure data integrity. Examples of lossy compression include JPEG for images and MP3 for audio. Although lossy compression techniques reduce the size of the data more than lossless compression techniques, some important information may be lost in the process. For example, in the case of images, the quality of the image may be reduced significantly [24]. In the context of this thesis, we may find useful compressing (using a lossless strategy, like gzip [25], for example) the JSON data resulting from user requests. It is a trade-off to be studied between the size of the data and the time and resources it takes to compress and decompress it.

Another possible technique is to pre-fetch some data and, if necessary, make aggregations and other computations *a priori*. This is done following a chosen schedule and according to the users' needs.

2.6 Related Work

In this section, two related projects are presented. The first one serves as a base model for several others, despite many years have passed since its release to the public. The second one was developed more recently, but its context has more similarities with this thesis.

2.6.1 The Tsimmis project

The Tsimmis project [26] was developed in the decade of 1990 and its name stands for "The Stanford-IBM Manager of Multiple Information Sources". Both Stanford and the IBM Almaden Research Center collaborated on Tsimmis. In summary, the goal of this project is to integrate heterogeneous data sources, which may have structured and unstructured data. The Figure 2.9 presents the architecture of Tsimmis.

The *Translator*, attached to each data source, is the same as a wrapper and has the job of translating queries coming from the *Mediator* into a language that the local data source can interpret. It also translates the local data to a common model, called Object Exchange Model (OEM). In simple terms, it uses labels to represent an object. For example, "the following object represents a Fahrenheit temperature of 80 degrees: <temp-in-Fahrenheit, int, 80>".

The *Mediators* receive a query and send it to the wrappers of the corresponding sources. This project aims to have simple mediators that do not perform complex tasks on the

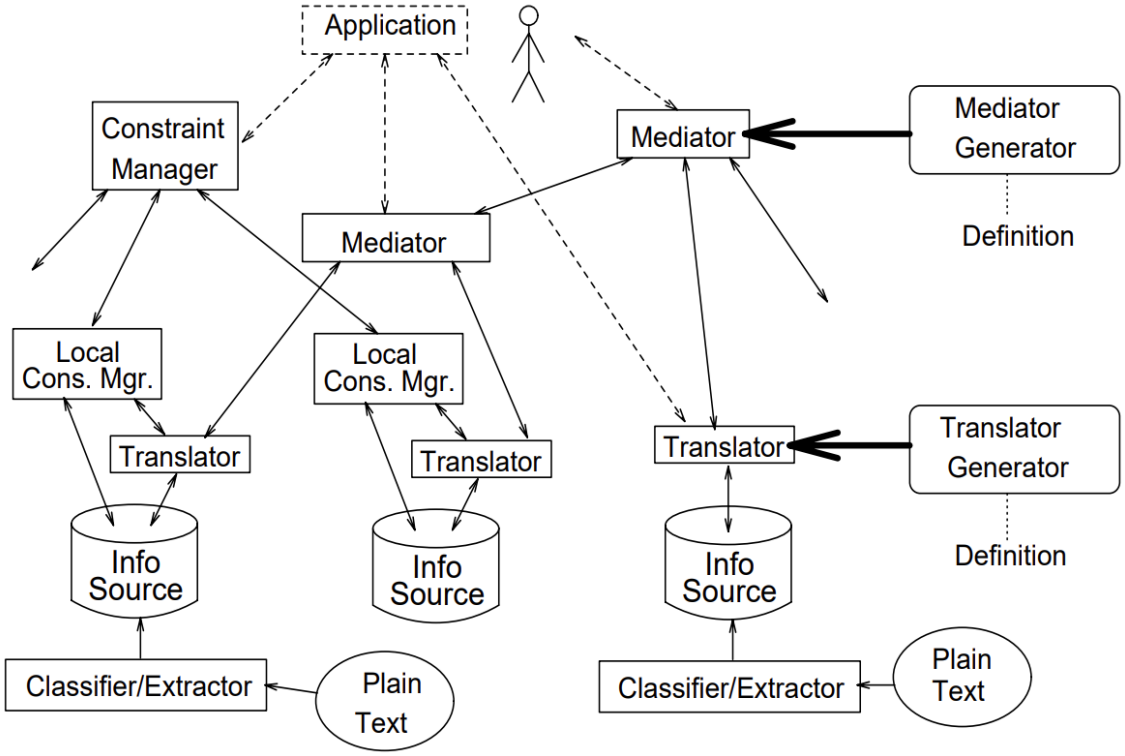


Figure 2.9: Architecture of the Tsimmis project (Adapted from [26]).

query they received before forwarding it. For example, mediators that do not eliminate duplicate information. So, these mediators can be semi-automatically generated, just like the translators can. It is important to mention that there can be more than one mediator in a system, which increases robustness (in case one fails), efficiency (some can be optimized for certain data sources), flexibility (incorporation of new sources without affecting the system) and scalability (handle more data). Furthermore, each mediator only needs to know the sources of the data it is responsible for. This means that there is no need to exist a global view of all the data of all data sources.

The Tsimmis project created a query language for OEM objects, called OEM-QL. This is the language that is used as input for the mediators and translators. The user must understand this language to write a query.

The components *Constraint Manager* and *Local Constraint Managers* ensure that the data from the heterogeneous sources is represented by the translators in a consistent manner, even if they have different local formats.

Finally, this project has a component called *Classifier/Extractor*. This component's job is to extract key properties from the sources that are totally unstructured (plain files, for example), before it is translated and sent to the mediators.

2.6.2 Lion: Listen Online. Using GraphQL as a mediator for data integration and ingestion

Lion [5] (or "Listen Online") is a system that tries to solve some problems that are discussed in this thesis. Briefly, there are several GraphQL servers available and this system attempts to simulate a unified view of them. It uses the GraphQL specification to provide integration to queries over web services. Besides, it benefits from open-source libraries and it implements a visual query builder destined for people who do not have programming skills.

Lion uses GraphQL to represent the web services involved. Hence, users can query them using only this query language. Besides all the data returned from the results of these queries is usually in the form of JSON.

This project presents a graphical component to help users build queries. It has a tree layout, which is a rooted digraph, or directed graph, with no cycles, as we can see in the Figure 2.10.

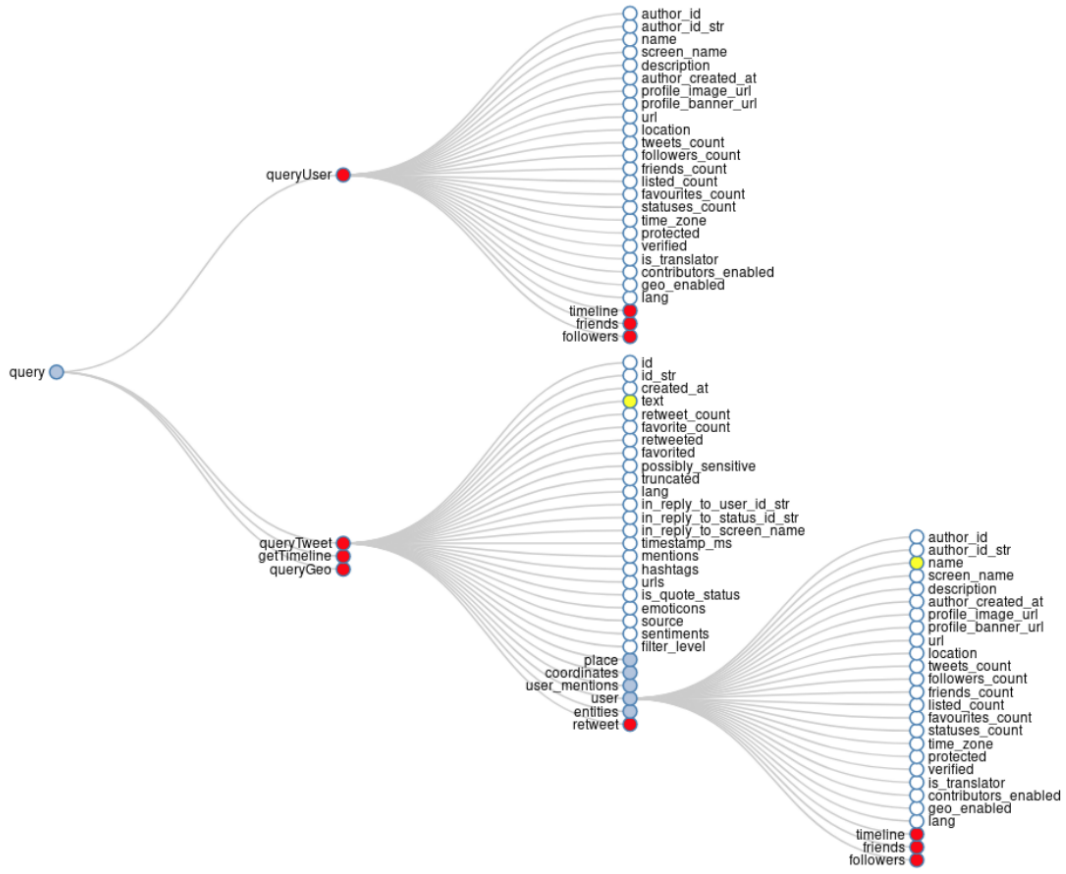


Figure 2.10: Example of the graph generated by a built query (Adapted from [5]).

There is also a component called form-generator, which helps users defining parameters for their queries. This is pictured in the Figure 2.11.

Figure 2.11: Form-generator component (Adapted from [5]).

Furthermore, if a user intends to make a query on top of the result of another query, he can by using another graphical component provided by Lion, which has an appealing interface with coloring and labeling. It also offers the possibility of interaction with the user. This is illustrated in the Figure 2.12, where it is created a composition of functions and words are extracted from a set of tweets.

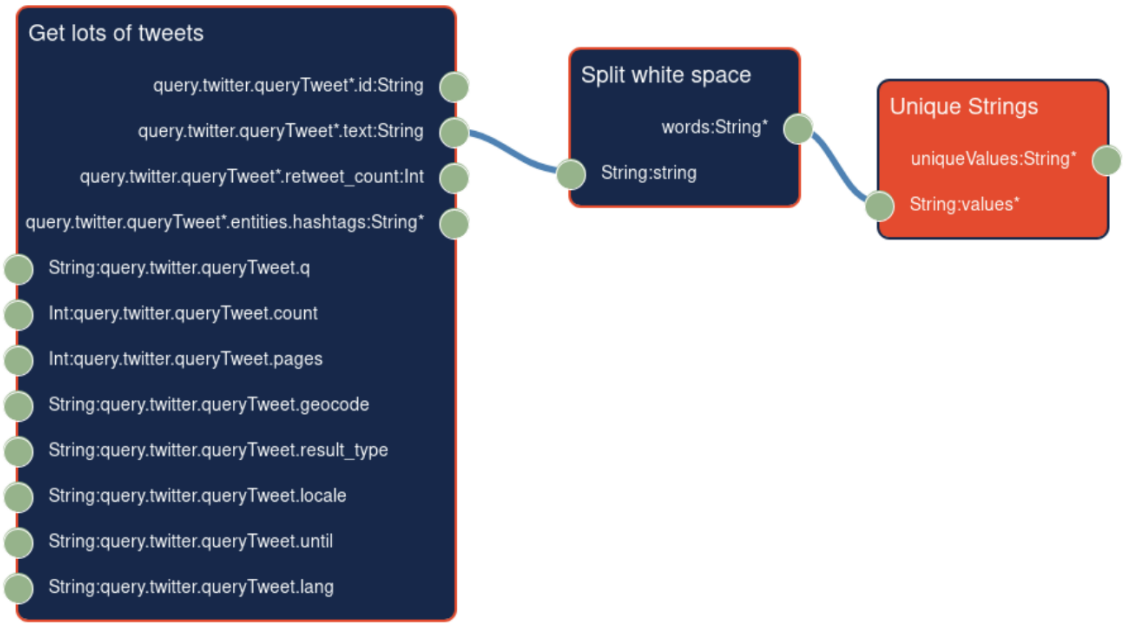


Figure 2.12: Example of the visual representation of a built query (Adapted from [5]).

This system presents an algorithm to transform the visual query made by the user into an actual GraphQL query. Besides, it also has an algorithm to check the validity of a query: as it traverses the digraph, it checks if there is at least one subtree containing a selected scalar attribute.

Lion stores the structure of the queries made so that the user has the possibility of selecting them again in the future. Furthermore, the user can set a schedule to run queries (every minute, hour, day, etc).

One interesting feature of Lion is using the output retrieved from a query as the input for another query. Besides, this project also aims to solve some limitations of GraphQL. For example, as it was said before, this query language is not able to perform aggregations. Lion also addresses another GraphQL limitation: this query language does not support filtering and selection. In SQL this is usually done by using the "WHERE" clause, which is not included in the grammar of GraphQL. This is all solved by applying custom functions on top of the output of another query. These functions can be divided into three categories: transformative, dimensional reductive and dimensional expansion. The first one includes sorting, limiting, filtering, among others. The second type of functions usually include aggregations and are mostly used to calculate maximums, minimums and

perform counting of values. The last type are used to, for example, split a string based off of white spaces.

Finally, Lion is able to handle authentication and authorization over web services when requests are made.

In summary, both projects are great to better understand the approaches for data extraction from heterogeneous data sources shown in the state of the art of this document. Lion contains a graphical interface with some features that can be useful for the prototype of this thesis, like the tree layout formed to build a query.

3

Concept

This chapter describes from a high-level point of view the concept of the solution. It is divided into five main sections: the ”No-code UI”, the ”Compiler service”, the ”Scheduling service”, the ”Document service”, the ”Global View”and the ”Solutions for the thesis' problems”. The first one includes several subsections corresponding to the components that make up the interface. The second and the third ones describe the services that are responsible for compiling and scheduling. The fourth section describes the Document service, while the fifth one presents a global view of the prototype complemented with diagrams. The final section states the chosen solutions for the challenges of the thesis, which are discussed in chapter 2.

3.1 No-code UI

This section introduces several no-code components available to users. The interface developed in this thesis is specifically designed for Super-creators. Its objective is to assist users in extracting, filtering, aggregating, and saving data that can subsequently be utilized by Creators to construct workspaces.

3.1.1 Diagram Editor

This interface features a Diagram Editor, which is a component enabling users to construct a tree within a designated pane. Within the component's side panel, there are various boxes. They can be dragged, dropped, and interconnected, and serve various purposes when clicked. The key components integral to this interface comprise the Visual Query Builder, the Filter Editor, the Aggregation Editor, the Schema Editor, and the Schedule Editor. All of them are no-code components and their details are presented below.

3.1.2 Visual Query Builder

The Visual Query Builder is designed to bridge the gap between users with no programming skills and the construction of queries. This component provides an interface that does not depend on the data source. Hence, it represents queries in a unique and generic

way - a tree-like structure. Visually, it is very similar to the syntax of a basic GraphQL query. Nevertheless, it is not the same because filters in GraphQL require a syntax that can be complex for users with no programming skills, especially when it involves filters. Thus, the Visual Query Builder includes a Filter Editor component that facilitates the creation of filters in a more intuitive manner.

Behind the scenes, in the back-end, this component produces queries in a single format, that can later be parsed and be translated to different languages, like SQL and GraphQL, for example.

There are some key aspects common across various query languages that enable a universal query representation. These were mentioned in subsubsection 2.3.3.3.

3.1.3 Filter Editor

Oftentimes, queries involve filters. These filters are used to narrow down the results of a query by applying specific conditions. In some query languages the filtering syntax can become quite complex. Thus, a no-code graphical component is required.

The interface of this component should allow users to create conditions. Each one is composed of a property, an operator, and a value. Firstly, the property can be selected from a drop-down menu containing all the properties within the query. Then, the operator (”equals”, ”contains”, among others) can be chosen in a similar manner. Finally, the value can be entered in a text field. Additionally, between conditions, users can add logical operators, such as ”and” and ”or”.

To simplify the implementation process, there are some libraries that provide pre-built React [27] components for this purpose.

3.1.4 Aggregation Editor

Aggregations are also very common in queries. Common aggregations are, for example, ”count”, ”sum”, ”average”, and ”maximum”. In the context of this thesis, the goal is to provide at least one type of aggregation in a no-code component. Other types can be added by extension in the future. Obviously, the addition of new aggregations requires some changes not only in the back-end, but also in the front-end. There are JavaScript libraries that implement aggregations, like the Lodash library. This way, as we add new aggregations, we simply add new calls to these libraries. On the other hand, the front-end updates are more specific to the type of aggregation being added and the input parameters it requires.

For instance, if we add the ”count by” aggregation, the visual editor simply needs to allow users to choose the property in the query to count by. This can be achieved with a drop-down menu that contains all the query’s properties.

3.1.5 Schema Editor

After the user has constructed a query, defined filters, and added aggregations, the next step is to save the data. To accomplish this, the user needs to create a schema by defining a set of properties and their types, along with the relationships between them.

Creating a schema can be a complex task if done by writing code. The Figure 3.1 shows the required JSON code for creating a specific schema. Expecting users of the Skills Workflow's platform to be proficient in writing this code is unrealistic; even software engineers might find it confusing initially. Thus, a no-code component is essential.

3.1.6 Schedule Editor

This component is responsible for defining a schedule for the execution of a task. The user can choose between a one-time execution or a recurring one. In the latter case, the user should be able to easily define the frequency of the execution.

3.2 Compiler service

When adding a query, an aggregation, a filter, and a schema, the user, without even noticing, is constructing a syntax tree, which is defined in the subsection below.

3.2.1 Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a hierarchical representation of the syntactic structure of a program or code snippet. This kind of tree is used mainly in programming language processing, compilers, interpreters, and other tools for code analysis and transformation. Moreover, its nodes represent the high-level structure of code while abstracting away the low-level details. Compiling an AST involves creating a parser for a hierarchical tree that contains essential information [28].

3.2.2 Concept

The Compiler service is responsible for compiling the user-defined syntax tree in the Diagram Editor.

Initially, this service receives the syntax tree in JSON format from the front-end. Based on the nodes present in the tree, it generates and writes specific code to a file. Finally, if a schedule is defined, this file is sent to the Scheduling service, which is responsible for running the query. Otherwise, it is ran immediately. Either way, the file is executed using Node.js [29].

The Compiler service includes other parsers besides the one that reads the syntax tree. For instance, if the user creates a query with the Visual Query Builder, and wants to generate GraphQL, the front-end produces a specific JSON that fits into a node of the syntax tree. Then, the Compiler service parses the main tree and identifies a node that is

```json
{
  "name": "countByAbsenceTypeName",
  "pluralName": "countByAbsenceTypeName",
  "jsonSchema": {
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "type": "object",
    "description": "Schema description",
    "required": [
      "count"
    ],
    "properties": {
      "count": {
        "type": "array",
        "items": {
          "absenceTypes": {
            "type": "object",
            "properties": {
              "absenceTypeName": {
                "type": "string"
              },
              "count": {
                "type": "integer"
              }
            },
            "required": [
              "absenceTypeName",
              "count"
            ]
          }
        }
      }
    }
  }
}
```

Figure 3.1: JSON code required to create a schema.

another tree itself. Hence, the GraphQL parser is called and GraphQL code is generated. Further implementation details about this service are described in chapter 4.

3.3 Scheduling service

The need for a Scheduling service arises from the fact that users may sometimes want to run tasks at recurring intervals. If the only available service for executing tasks was the Compiler service, the latter would need to be actively waiting for the next execution, preventing it from serving other requests. Therefore, the Compiler service is responsible only for running tasks immediately, or, if they have a schedule attached, for sending them to be executed in the Scheduling service. In summary, the main goal of this service is to allow storing data at recurring intervals in the final data source that can be accessed by Super-creators and, in particular, Creators.

3.4 Document service

This service was already implemented and fully functional before the start of this thesis. The Compiler and Scheduling services communicate with this service to retrieve and store data. It is also responsible for storing data according to a schema defined in the Schema Editor. The Document service supports all types of data and schemas. It uses a NoSQL database: CosmosDB [30].

3.5 Global view

This section presents some diagrams to illustrate the global view of the prototype.

Firstly, the Figure 3.2 shows a diagram with a sketch of the architecture of the solution. It is divided in three columns, each representing one user of the platform. The column on the center mentions a UI that is not yet implemented. This UI communicates with the Package service, responsible for retrieving packages where workspaces are stored. The column on the right shows the role of the End-user, which is simply visualizing and interacting with workspaces in a web page. On the left column, the Super-creator interacts with the no-code UI to construct a syntax tree to request, transform, and store data. Then, the Compiler service and Scheduling services generate Javascript code and execute it, communicating with the Document service.

Secondly, the Figure 3.3 illustrates what components are comprised in the no-code UI and how they interact with each other. This UI includes a Diagram Editor, a Visual Query Builder, a Filter Editor, an Aggregation Editor, a Schema Editor, and a Schedule Editor. These components are responsible for generating a syntax tree in JSON format, which is then sent to the Compiler service.

The Figure 3.4 depicts the process of compiling and executing a syntax tree. From the UI, arrives a syntax tree that, in this case, contains three nodes and a schedule. This

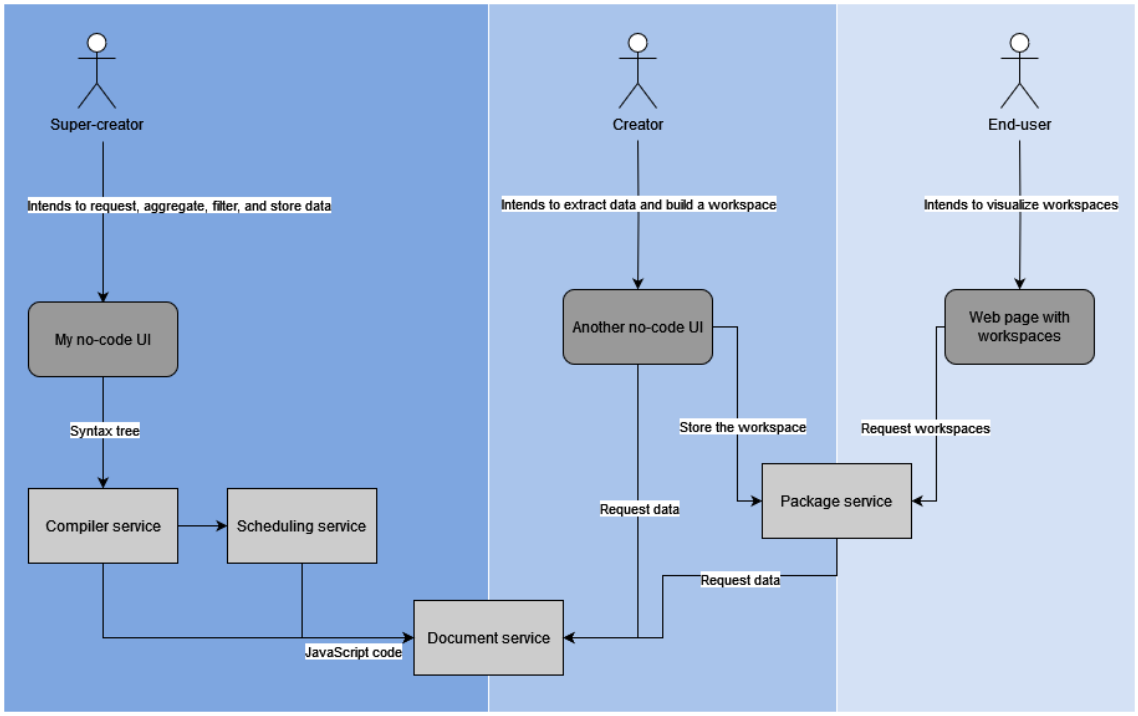


Figure 3.2: Global architecture of the system.

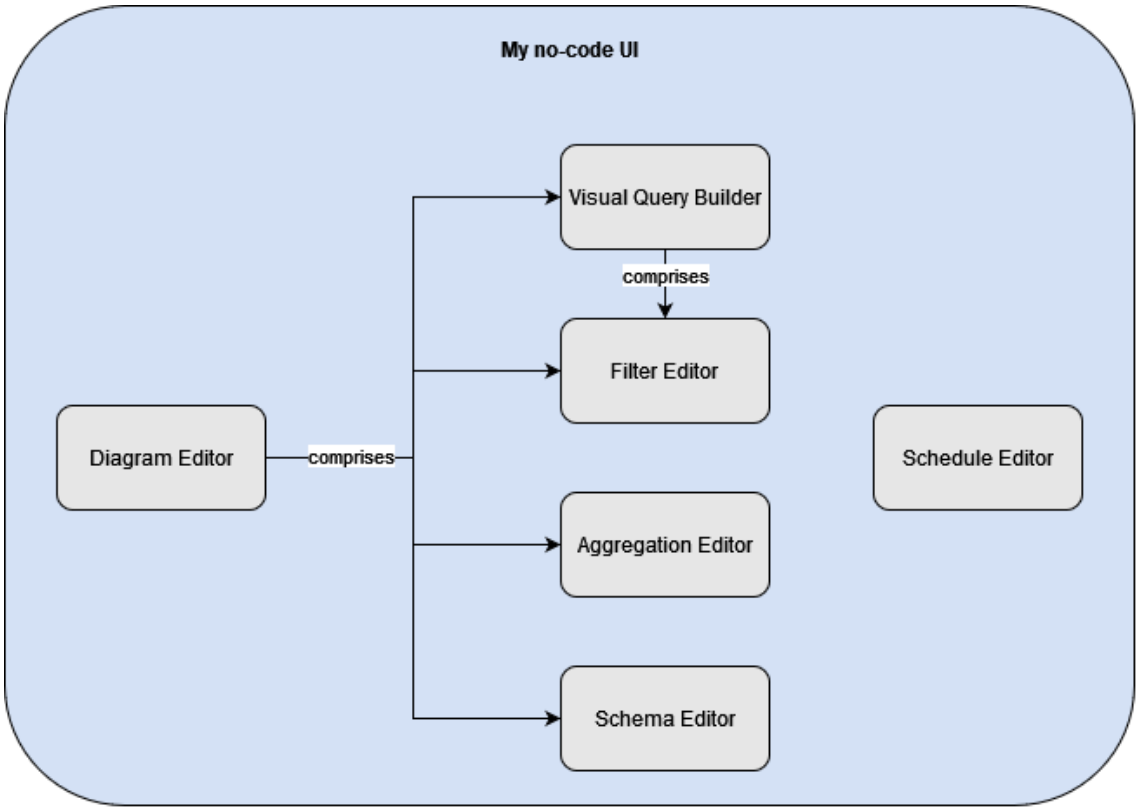


Figure 3.3: Components comprised in the no-code UI.

information is extracted by the main parser in the Compiler service. The GraphQL and SQL parsers are called because an ”api-request” node was identified. These extract the parameters inside the node and translate the query into the respective query language. For each node, a specific compilation method is called. Here, there are certain generic functions and specific lines of JavaScript code that are written into a file. Then, the Scheduling service is called and a schedule library and Node.js are used to execute the file at the specified times.

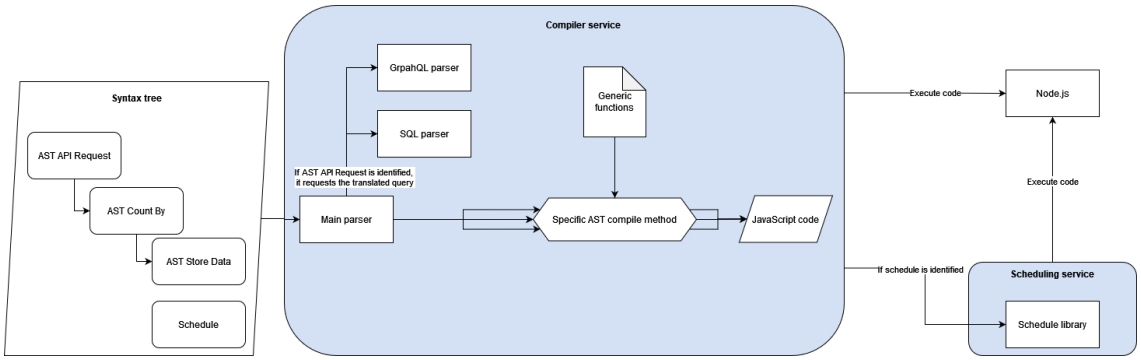


Figure 3.4: Process of compiling and executing a syntax tree.

Finally, the Figure 3.5 shows a simplified version of the structure of the syntax tree used in the Figure 3.4. It is a simplified version because it does not include information such as connection points, identifiers, among others. We can distinguish the three nodes and the schedule. There is also always a ”script-head” node containing all the nodes. It is worth noting that the ”requestOptions” on the ”api-request” node are empty because this tree at this stage had not gone through the Compiler service yet, where these are filled manually (this should be avoided in the future).

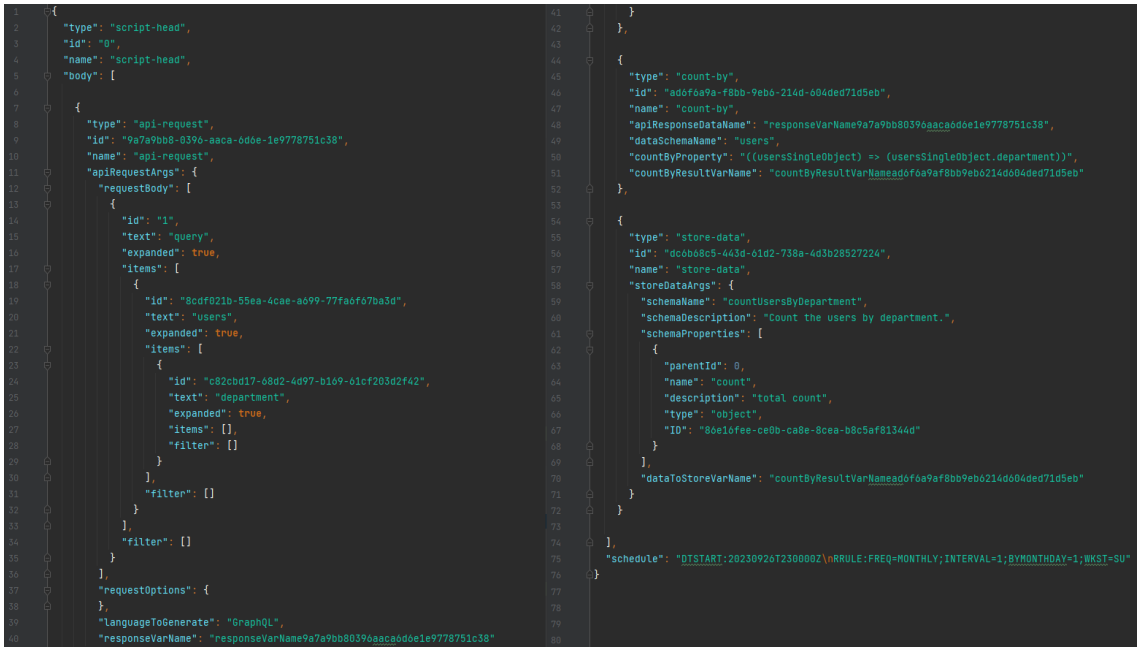
3.6 Solutions for the thesis' challenges

3.6.1 No-code UI

The orchestration of the several components described above make up the no-code UI. Although we say that the interface is *no-code*, it can be considered *low-code* in some cases. Despite the user not needing to write code, some programming concepts are required. For instance, when defining a schema in the Schema Editor, the user needs to know what a property is and what its type is.

3.6.2 Data extraction from heterogeneous data sources

We present various common solutions for extracting data from heterogeneous data sources in section 2.4. Nevertheless, the chosen one for this thesis is the ETL solution. With the help of the components and services presented above, it is possible to extract data (and



```
{
  "type": "script-head",
  "id": "0",
  "name": "script-head",
  "body": [

    {
      "type": "api-request",
      "id": "9a7a9bb8-0396-aaca-6d0e-1e9778751c38",
      "name": "api-request",
      "apiRequestArgs": {
        "requestBody": [
          {
            "id": "1",
            "text": "query",
            "expanded": true,
            "items": [
              {
                "id": "8cdf821b-55ea-4cae-a699-77fa6f67ba3d",
                "text": "users",
                "expanded": true,
                "items": [
                  {
                    "id": "c82cbd17-68d2-4d97-b169-61cf203d2f42",
                    "text": "department",
                    "expanded": true,
                    "items": [],
                    "filter": []
                  }
                ],
                "filter": []
              }
            ],
            "filter": []
          }
        ],
        "requestOptions": {
        },
        "languageToGenerate": "GraphQL",
        "responseVarName": "responseVarName9a7a9bb80396aaca6d0e1e9778751c38"
      }
    },

    {
      "type": "count-by",
      "id": "ad6f6a9a-f8bb-9eb6-214d-604ded71d5eb",
      "name": "count-by",
      "apiResponseDataName": "responseVarName9a7a9bb80396aaca6d0e1e9778751c38",
      "dataSchemaName": "users",
      "countByProperty": "((usersSingleObject) => (usersSingleObject.department))",
      "countByResultVarName": "countByResultVarNamead6f6a9af8bb9eb6214d604ded71d5eb"
    },

    {
      "type": "store-data",
      "id": "dc6b68c5-443d-61d2-738a-4d3b28527224",
      "name": "store-data",
      "storeDataArgs": {
        "schemaName": "countUsersByDepartment",
        "schemaDescription": "Count the users by department.",
        "schemaProperties": [
          {
            "parentId": 0,
            "name": "count",
            "description": "total count",
            "type": "object",
            "ID": "86e16fee-ce0b-ca8e-8cea-b8c5af81344d"
          }
        ],
        "dataToStoreVarName": "countByResultVarNamead6f6a9af8bb9eb6214d604ded71d5eb"
      }
    }

  ],
  "schedule": "DTSTART:20230926T230000Z\nRRULE:FREQ=MONTHLY;INTERVAL=1;BYMONTHDAY=1;WKST=SU"
}
```


Figure 3.5: Simplified structure of a syntax tree.

applying transformations if needed) from any data source and store it in a single system. In our case, the Super-creator is responsible for this task. Thus, a unified view of data coming from heterogeneous data sources is then available for another user - the Creator - to build workspaces. This last part is not yet implemented, but discussed in subsubsection 6.1.0.1.

Regarding the federated system option, besides often being a payed service, it is not a good solution for this problem. Our platform requires the introduction of new data sources very frequently. Some of these data sources may not be supported by the chosen federated system. Thus, we opted to have more control over the data integration process, implementing an ETL approach.

The custom solution with mediators and wrappers was discarded because we assume, for simplification purposes, that the Super-creator knows what the data query language of the data source is. Plus, this solution is able to extract data from multiple data sources with just one request, which is something we cannot currently do. Besides, large dataset extraction is a problem when adopting this solution (or a federated system) because the data is extracted and computed on demand, which is not the case with the ETL solution. This is discussed in more detail in the next subsection.

3.6.3 Large dataset extraction

To solve the problem of efficiently extracting large quantities of data, we use one of the techniques presented in the section 2.5: the aggregations and computations are made *a priori*. This way, the Creator can worry only about consuming this data, without waiting for it to be computed. Again, this task is performed by the Super-creator.

4

Implementation

This chapter outlines the development journey of the solution. It presents the key milestones in the implementation of the solution in chronological order, allowing the reader to follow the programmer's steps and comprehend the decisions made throughout the process.

4.1 JavaScript file to fetch, aggregate and store data

The first major challenge was to find a way of fetching, aggregating, and storing data using JavaScript.

Initially, we opted to utilize the company's GraphQL endpoint to communicate with its database. For making requests, we employed the HTTPS module [23] included in Node.js. After comprehending the schema, we decided to make a GraphQL query to retrieve "absences". The resulting output would consist of fundamental information about the absences: ID, associated employee, approvers, date range, and others.

Secondly, we aimed to perform an aggregation on the output data. To achieve this, we developed a function that took the JSON query output and counted the number of absences per month of a given year. After running the query and the aggregation function, we obtained an array with twelve entries. Each entry contained a field indicating the number of absences for the corresponding month of that year.

We required a method to store the result in a schema, which led us to construct a generic function for this purpose. This function took a schema name as its input and performed a check to determine its existence. To perform this check, we made a call to one of the company's endpoints. If the schema already existed, any existing instances of that schema were deleted. This step ensured the schema could be updated with the incoming new data. In the event that the schema did not exist, an alternative endpoint was invoked to create a new schema.

The syntax for defining the schema was previously illustrated in the Figure 3.1. Despite the function's generic design, it still needed some adaptation based on the nature of the data being stored. In this particular case, the data had a unique type, requiring a specific

implementation for inserting instances.

Finally, we ran the created JavaScript file using Node.js. In order to check if the data was stored correctly, we made a request to the endpoint that returned all the instances of a schema.

4.2 Updates to the Compiler service

The following step was generating and executing a JavaScript file, like the one described previously, automatically. To achieve this, we needed a service capable of reading a syntax tree in JSON format and generating a JavaScript file. Instead of coming from the front-end, the JSON code was manually created temporarily. The JSON file simply contained essential information, and the service would have to be able to produce generic JavaScript code from it.

Initially, we already had an available service serving this purpose. However, the parser only supported basic nodes, such as arithmetic operations, logical operations, and others. Thus, we had to extend the parser to support the new nodes introduced in the syntax tree - the fetch, aggregate and store nodes. This was a challenging task because the parser was written in JavaScript, which is not a strongly typed language. Besides, the parser code of this service was compacted into a single file. Hence, the existing Compiler service needed a few updates.

The updates began with manually converting the JavaScript parser code into TypeScript, providing better typing support.

Additionally, we introduced the concept of AST, as explained in subsection 3.2.1. This modification allowed the parser to read the syntax tree and identify the AST nodes, which, in turn, would add specific lines of code to the resulting JavaScript file.

To organize the service, we adopted a layered architecture. The ”presentation” folder contained the API and the controller. The first one housed endpoints accessible from the front-end. The second one managed logical validations and service calls. Within the ”service” folder resided the actual service, responsible for reading each node of the syntax tree and invoking the compile method for the corresponding AST node. There was another folder added - the ”utils” folder - which contained the all the AST classes, the parser, and other useful functions, such as the one that discovered the line on the JavaScript file where a new import should be added.

Lastly, we added the possibility for the Compiler service to execute the generated JavaScript file using Node.js.

4.3 Compile a JSON file to fetch, aggregate and store data

4.3.1 Lodash

Lodash [31] is a modern JavaScript utility library that provides a wide variety of functions that work with arrays, numbers, objects, strings, among others. It is very useful in common programming tasks, such as performing iterations, data manipulations, and more. By leveraging Lodash, developers can enhance their productivity and create more efficient and readable code.

4.3.2 Implementation

One remaining task related to the Compiler service involved expanding the parser's capabilities to support additional nodes.

The first new essential node was the "api-request" node. This node was responsible for making HTTPS requests and contained options for the request. These options included the hostname, the path, the method type (POST, GET, among others), and the headers. Plus, there was a string field to identify the name of the new variable where the request response would be stored. This node also received a query.

To demonstrate that this solution could aggregate data, it was necessary to support at least one type of aggregation and provide an option for easily adding new ones. Therefore, Lodash was the ideal choice for this purpose. We opted to include the "count by" aggregation, which counts the number of occurrences of a specified property in an array of objects. For example, we could count the number of occurrences of approved absences within the absences array. In the parser's code, we introduced a new "ASTCountBy" node. The corresponding compilation method would import and invoke the "countBy" method from the Lodash library using the provided parameters: the array and the property to count by.

Finally, we extended the parser by adding an "ASTStoreData" node. At this stage, we simply passed the schema declaration as shown in the Figure 3.1. This node produced the necessary code to store the data in the schema, which involved incorporating the required imports ("HTTPS" and "URL" [32]) and the generic function for data storage, which were mentioned earlier in section 4.1.

With the addition of these three nodes, the Compiler service became capable of compiling and executing a JSON file to fetch, aggregate, and store data.

4.4 Build and compile a tree in the UI to fetch, aggregate and store data

4.4.1 DevExtreme

DevExtreme [33] offers a set of pre-built and customizable JavaScript components for web development, targeting developers using frameworks such as Angular [34], React, Vue [35], and jQuery [36].

Some included components are a data grid, interactive charts, data editors, navigation and multi-purpose UI components. The availability of all these tools, combined with both robust support and extensive documentation, makes DevExtreme an excellent fit for the Skills Workflow's platform.

4.4.2 Implementation

Previously, we manually created a syntax tree in JSON format and passed it to the Compiler service. Now, we required the UI to automatically generate this tree.

When I started working on this thesis, the company was already employing DevExtreme in its projects. The "Diagram" component from DevExtreme provided an ideal starting point for developing the no-code interface due to its responsiveness, intuitive interface, and integration with React. This component provided several different shapes that could be easily dragged and dropped from a side panel into a designated pane. These shapes could be interconnected, and their properties modified.

The Figure 4.1 shows a snapshot of the initial interface of the "Diagram" component provided by DevExtreme.

Clearly, we needed to introduce three new shapes to represent the concepts of fetching, aggregating, and storing data. Initially, for testing and simplification purposes, these shapes were kept minimalistic and non-editable. In other words, each shape was initialized with manually predefined data. Our intention was to progressively replace this data with dynamic inputs provided by the user.

The Figure 4.2 depicts the tree built with the three new nodes. The first one - the magnifying glass - represents the "ASTApiRequest" node. In the middle, the hash icon corresponds to the "ASTCountBy" node. The last one, illustrated by the database icon, represents the "ASTStoreData" node.

Based on the utilized nodes and the connections between them, the front-end would automatically produce a syntax tree in JSON format. The code responsible for this task was already implemented, but required modifications to support the new nodes. Subsequently, the user could click a button to compile and execute it using the Compiler service.

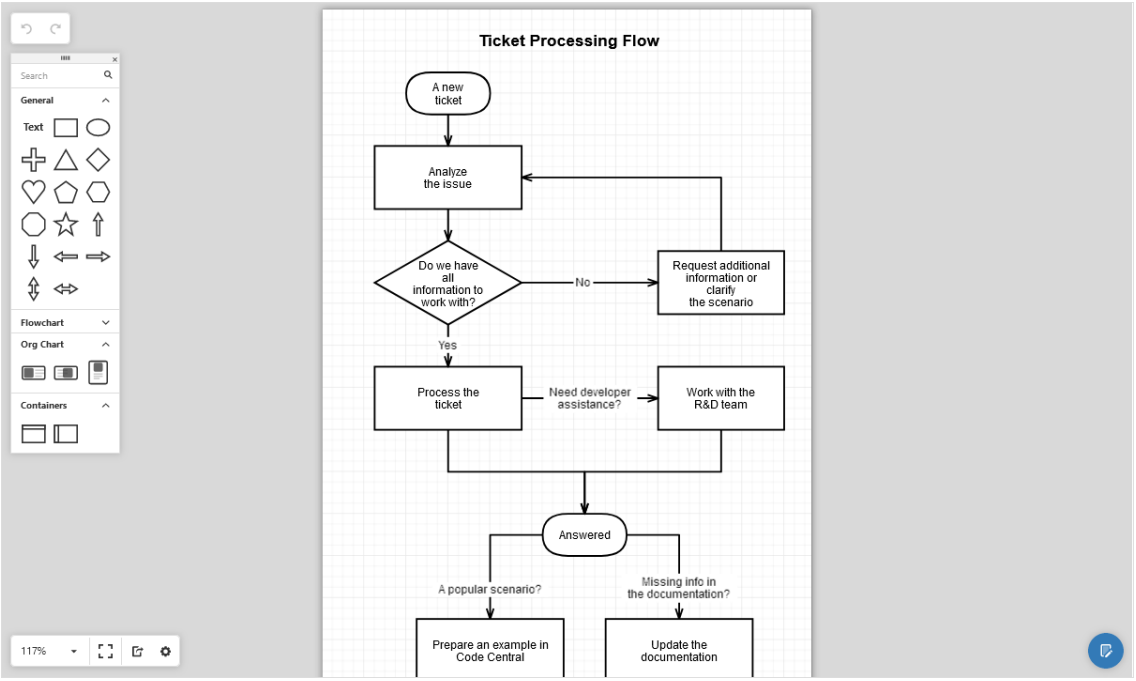


Figure 4.1: Snapshot of the "Diagram" component from DevExtreme (Adapted from [37]).

4.5 Integrating the Diagram Editor into the Skills Workflow's platform

Up to this point, the Diagram Editor was running locally. However, we needed to integrate the Diagram Editor into the Skills Workflow's platform. Therefore, we had to make the Diagram Editor run in a non-local environment.

So, instead of the browser requesting the Diagram Editor from the localhost, it would now request it from a server hosted on a cloud provider. This transition granted improved accessibility for users accessing the Diagram Editor from any device with an internet connection.

In essence, upon initiating the web application, the browser sends requests to retrieve all of the system's packages through a service called Package service. These packages contain workspaces, each of which comprises segments for customizing the UI and housing JavaScript code responsible for handling events.

This process involved creating a new workspace with two key wrappers[1]: one for the Diagram Editor and another for the button that called the Compiler service to compile the syntax tree. This task was accomplished by sending a request to an already existing endpoint. The body of the request had a complex JSON structure, which is explained in the two paragraphs below.

[1]A wrapper is a representative component that renders one or more components. For example, the "Button" wrapper renders a DevExtreme button. It contains logic to handle the button's click event, and some fields to customize the button's text and dimensions.

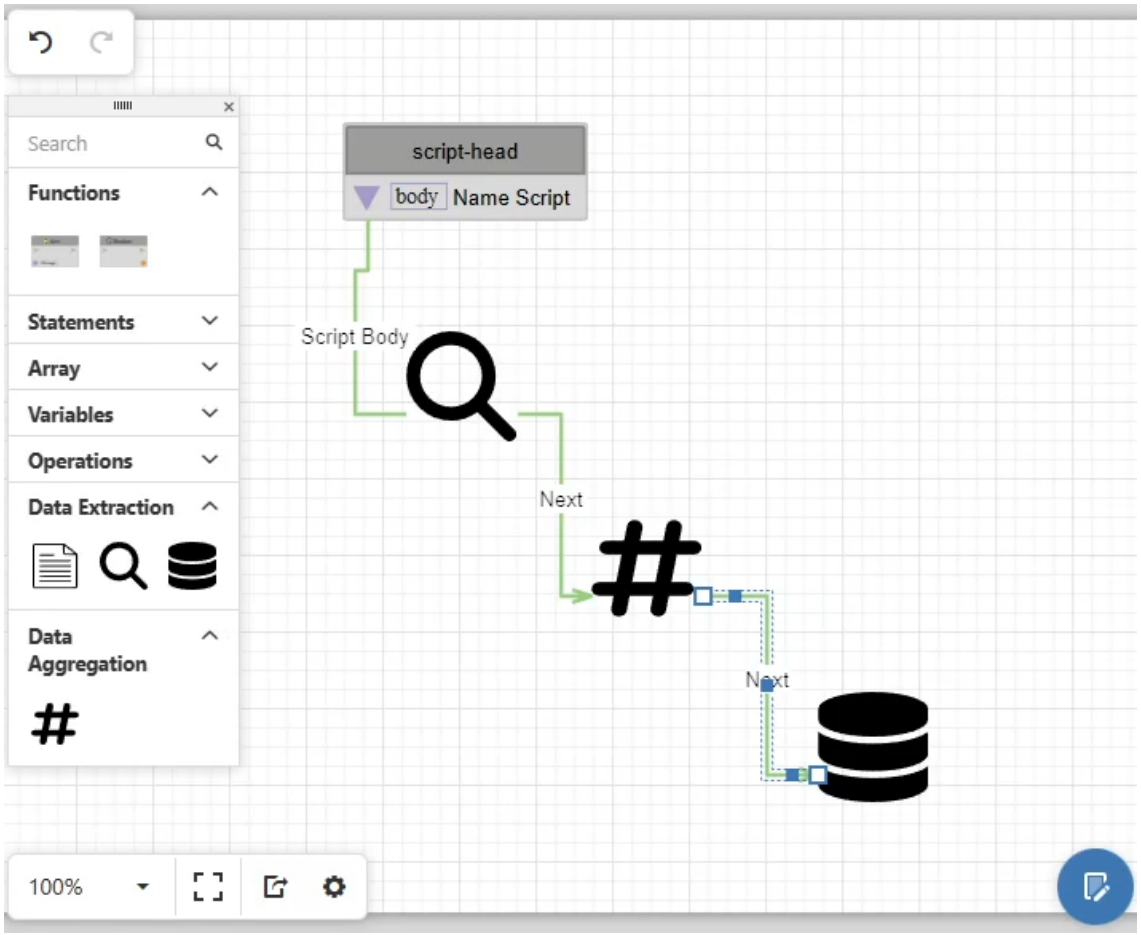


Figure 4.2: Snapshot of the UI with a tree for fetching, aggregating, and storing data.

The "root" section definition resembled CSS code, and was responsible for the design of the UI of the workspace. The "root" had two primary child sections. The first child section, the header, provided styling configurations for the workspace's title. The second section displayed the Diagram Editor and the "save diagram" button. It was configured with its own styling and initialized with the values of two variables: the syntax tree and the schema.

Lastly, the "library" section housed the JavaScript code to be executed when the "onClick" event was triggered. As explained in preceding sections, this code sent a POST request to the Compiler service, with the syntax tree as the request body.

In summary, we had to extend the shell of the Skills Workflow's platform to include the Diagram editor, integrating it through wrappers. To make this editor available to the Super-creator we created a new workspace.

4.6 Visual Query Builder - dynamic inputs for API requests

Although the Diagram Editor was functional and allowed users to fetch data, the request options were fixed. Hence, a new component, the Visual Query Builder, was developed enabling users to dynamically specify the data they wanted to fetch.

Fortunately, due to the high level of customization offered by DevExtreme components, it was possible to attach a button to the "api-request" shape, as depicted in the Figure 4.3. This button was labeled as "edit request".

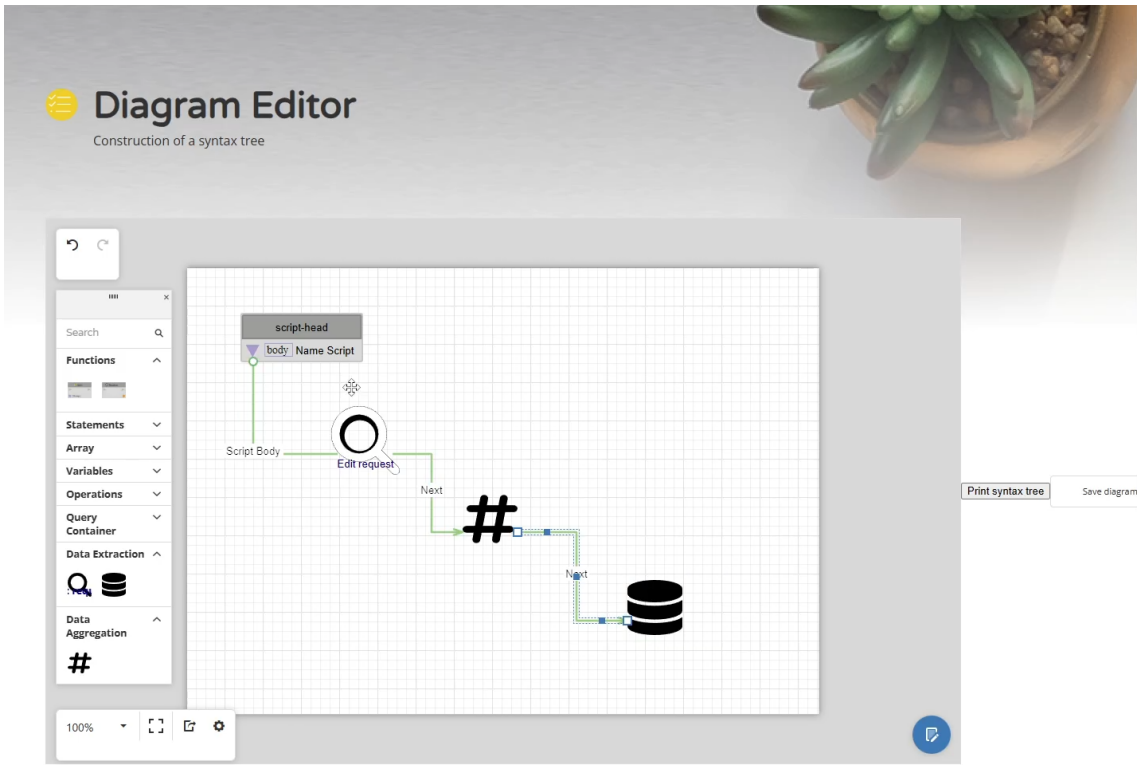


Figure 4.3: Button on "api-request" shape for editing request options.

When the "edit request" button was clicked, a new pop-up showed up: the Visual Query Builder component. This pop-up presented a no-code interface for users to construct a query in a tree-like structure. The versatile design made the component ready to support translation to several different query languages. However, at this stage, only data sources queried via GraphQL were accepted. This limitation is addressed in section 4.11.

The "Tree View" component [38] of DevExtreme was a good choice for this purpose because it could be easily manipulated by the user, and customized by the developer. The Figure 4.4 shows the initial interface of the Visual Query Builder component. In the upcoming sections, the design is improved and new features are added. To add a new node, which represents a query entity or field, the user clicks on the "add child node" button and enters its name. Here, we are assuming that the user is familiar with the schema of the data source. In subsubsection 6.1.0.3, an alternative solution to avoid this

assumption is discussed. The query built in the Figure 4.4 requests ”absences”, more specifically, the field ”value” of the stage of each absence, which may be ”approved”, ”rejected”, among others. This structure is very intuitive because it resembles the syntax of a basic GraphQL query. Therefore, by adding a new child node, the user is essentially either performing a ”join” between two entities, or selecting a field from an entity. There were also other buttons: ”remove node” and ”edit node”, to remove and edit a node from the tree, respectively.

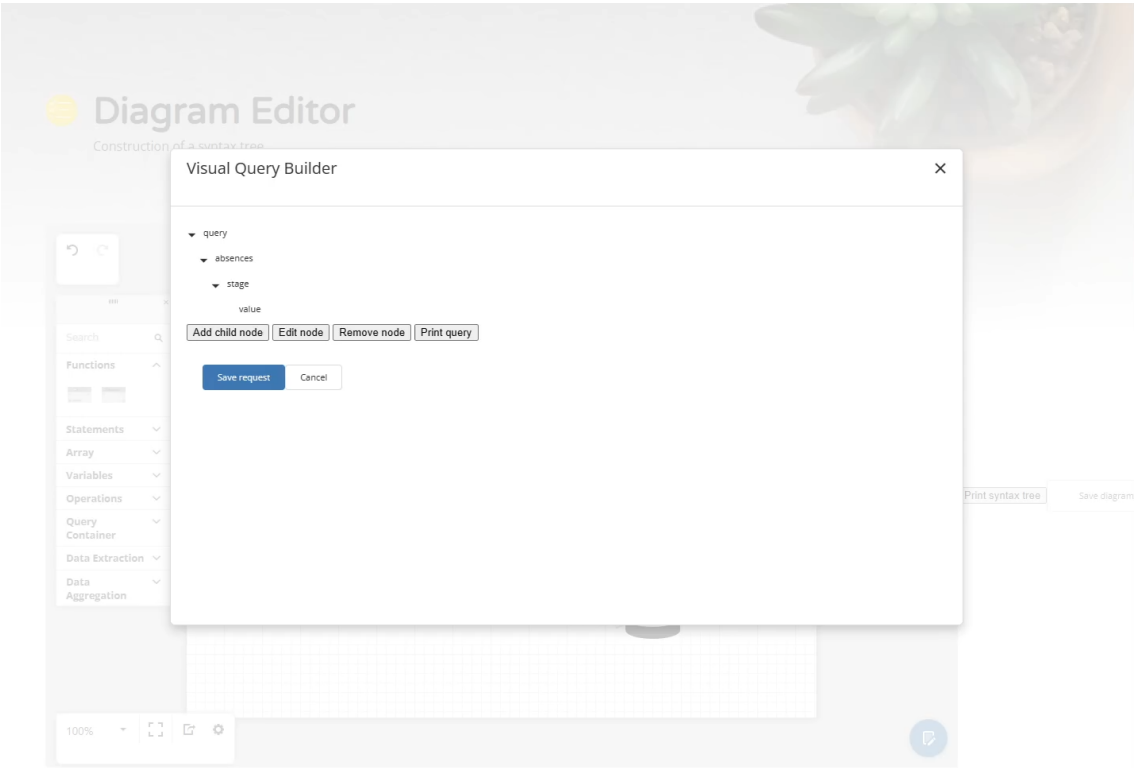


Figure 4.4: Snapshot of the first prototype of the Visual Query Builder.

Regarding the syntax tree in JSON format passed to the back-end, its ”api-request” node contained a variable produced by the UI, instead of fixed data. This variable stored the built query. The request options were still a parameter, but fixed. Again, in the last chapter, in subsubsection 6.1.0.4, a solution for making these options dynamic is discussed.

Behind the scenes, a new parser was introduced to the Compiler service to support the new dynamic information of the ”api-request” node. This parser contained a recursive function to transform the incoming structured data, which stored a query (in a tree too), into a query with GraphQL syntax.

4.7 Visual Query Builder's discarded solution - database entities approach

Sometimes, it is interesting to show the reader the alternative paths that were considered but later discarded. This section describes another solution for the interface of the Visual Query Builder component and it also presents the reasons why this option was not used in the final prototype.

The Figure 4.5 shows a snapshot of the Diagram Editor with a tree built for the same purpose as the one in the Figure 4.4. There, the box on the left would be destined for the "absences". The other one would be for the "stage", and for the "value", as a selected field. Ideally, the names of the entities would appear at the top of the box, with the selected fields displayed at the bottom. To add or edit these properties, each box has a button labeled "edit entity", which would open a pop-up with forms and/or select boxes.

In contrast to the prevailing solution, the "api-request" shape was not represented with an icon and a button that opens a pop-up when clicked. Instead, the query was constructed using interconnected entity boxes inside a collapsible and resizable container. These elements consisted in the Visual Query Builder.

Regarding the connections between entities, these would be done either manually or automatically. Manual connections required the user to have knowledge of the schema and select the field to join. Conversely, automated connections required the system to have knowledge of existing primary and foreign keys.

This approach bore some similarities with the Diagram pane of SSMS in the Figure 2.4. Consequently, some users might be familiar with this type of interface. Nevertheless, when compared to the solution using the "Tree View" component, this interface could become confusing when dealing with many fields and/or entities. All this information would occupy too much screen space. Although the container was collapsible and resizable, and that it might have been possible to add a scroll bar to the container and to the pop-up for defining the entity's properties, the boxes inside the container would still be excessively large. Proceeding with the implementation of this approach would require too much effort and time because there was no pre-built component available. In conclusion, this solution was discarded and replaced by the one described in the previous section.

4.8 Schema Editor - dynamic inputs for schema properties

After fetching data, our next task was to work on storing it. We already had a shape with fixed parameters for data storage. The next step was to create a new component, the Schema Editor, to allow this shape to receive dynamic inputs from the user, specifically the schema's definition.

During the implementation phase, two solutions were developed for this component. Both solutions required the user to first construct a tree containing a "store-data" shape and

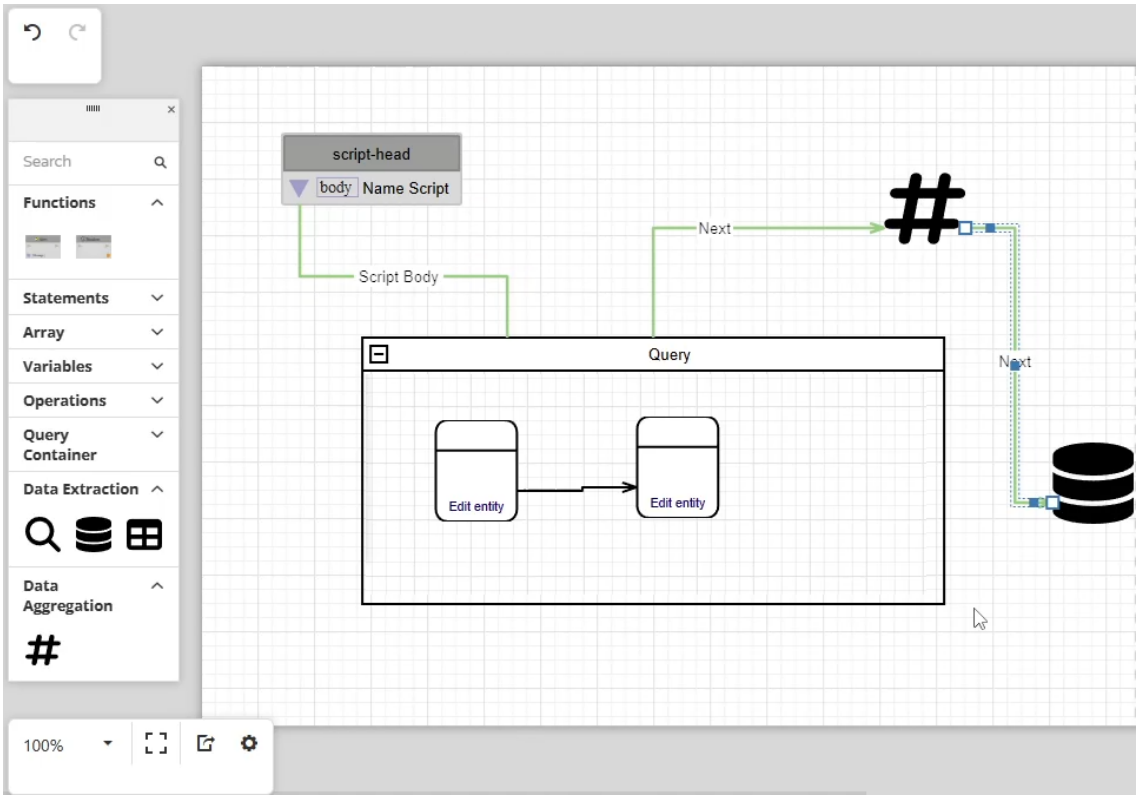


Figure 4.5: Snapshot of the Visual Query Builder's discarded solution - database entities approach.

then click on the corresponding "edit schema" button. Again, for better understanding of what a schema declaration requires, we may look at the example provided in the Figure 3.1.

In the first solution, as depicted in figures 4.6 and 4.7, each schema had a name, a description, and properties. Each property comprised a name, a description, and a type. The editable properties list was developed using the "List" component of DevExtreme. While this interface allowed the user to define multiple properties (by clicking on the "add property" button), it did not support properties of types such as "object" or "array", which are common in schemas. Therefore, this limitation led to the development of the second solution.

In the second approach, as illustrated in the Figure 4.8, with the usage of the "Tree List" component of DevExtreme, the interface already allowed the user to define properties of type "object" or "array". Despite this front-end improvement, the back-end was not yet capable of storing schemas with properties of these types. To address this issue, we would need to extend the generic function for data storage to be able to handle the new property types.

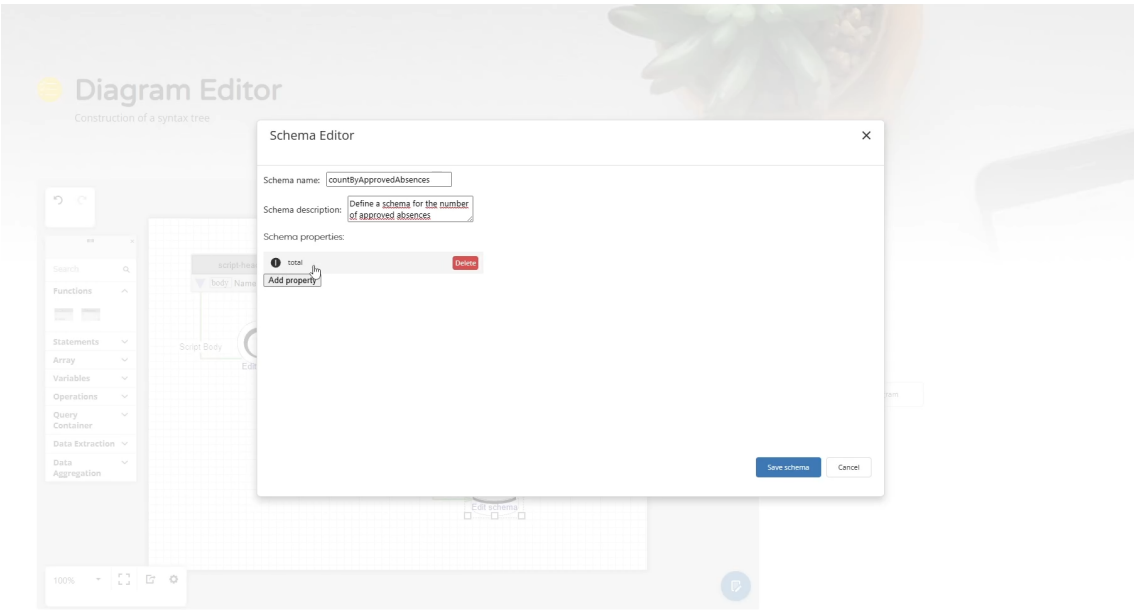


Figure 4.6: First prototype of Schema Editor - snapshot with schema's name, description, and properties list.

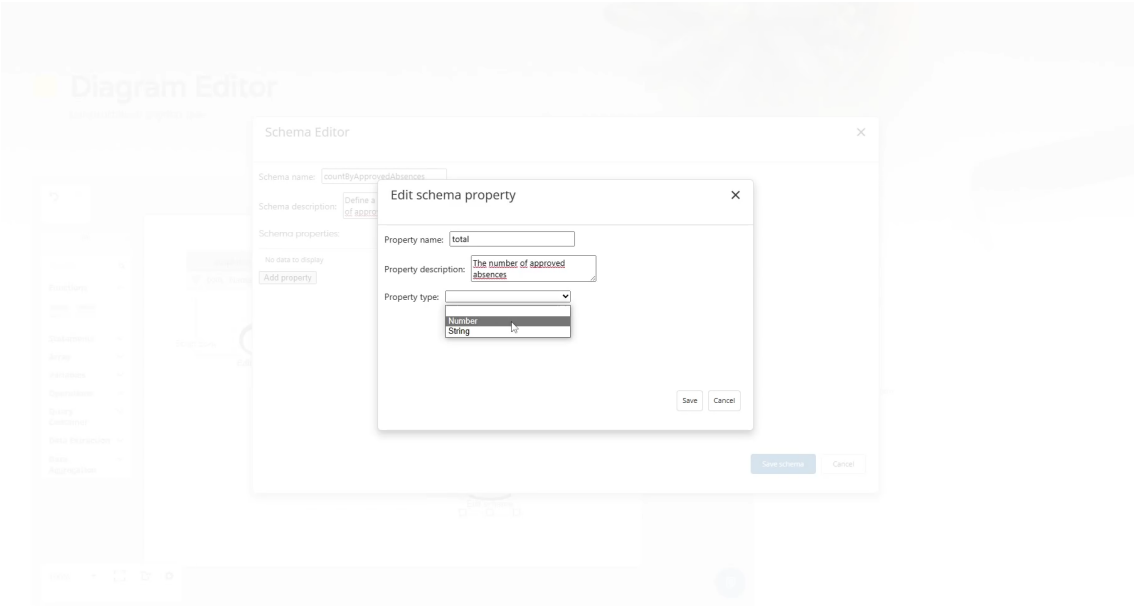


Figure 4.7: First prototype of Property Editor - snapshot with property's name, description, and type.

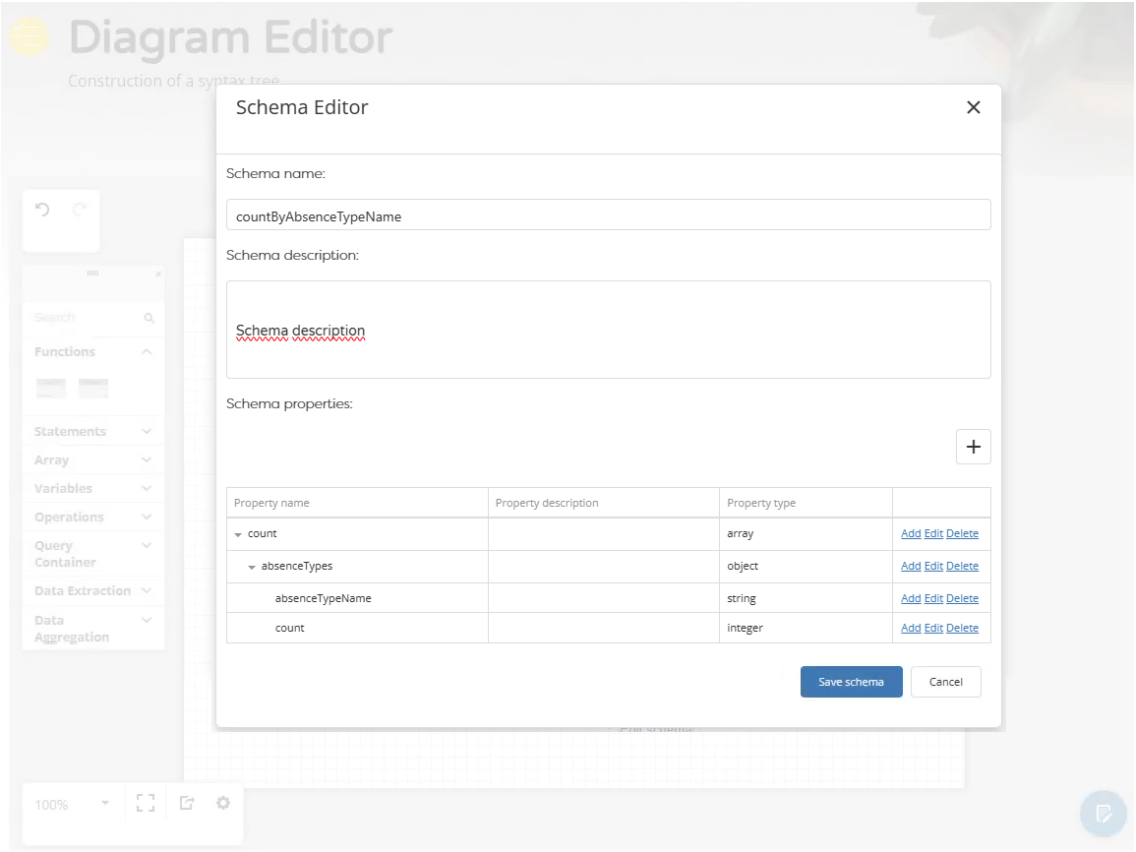


Figure 4.8: Second prototype of Schema Editor - snapshot with schema's name, description, and properties list.

4.9 Aggregation Editor (”Count by”) - dynamic inputs for aggregation parameters

Just like the request and storage nodes, aggregation nodes also required dynamic user inputs. To add the first type of aggregation to the interface, we introduced a new shape in the Diagram Editor. Because this aggregation was the ”count by” aggregation, it was represented by a hash icon, visible in the Figure 4.9. This shape also featured a button labeled ”edit aggregation”, which opened a pop-up, the Aggregation Editor, allowing users to define the aggregation parameters. In this case, the property in the query to count by was selected using a drop-down menu that contained all the properties of the query.

In the example presented in the Figure 4.10, the selected property was ”users.department”. After clicking on the ”save aggregation” button, the front-end was responsible for creating a node of type ”count-by” and placing it in the correct position within the syntax tree. This node was constructed with two parameters. First, the selected property required to be split into two parts: the name of the collection (”users”), and the name of the property to count by within the collection (”department”). The first part received the content that appeared before the first dot found.

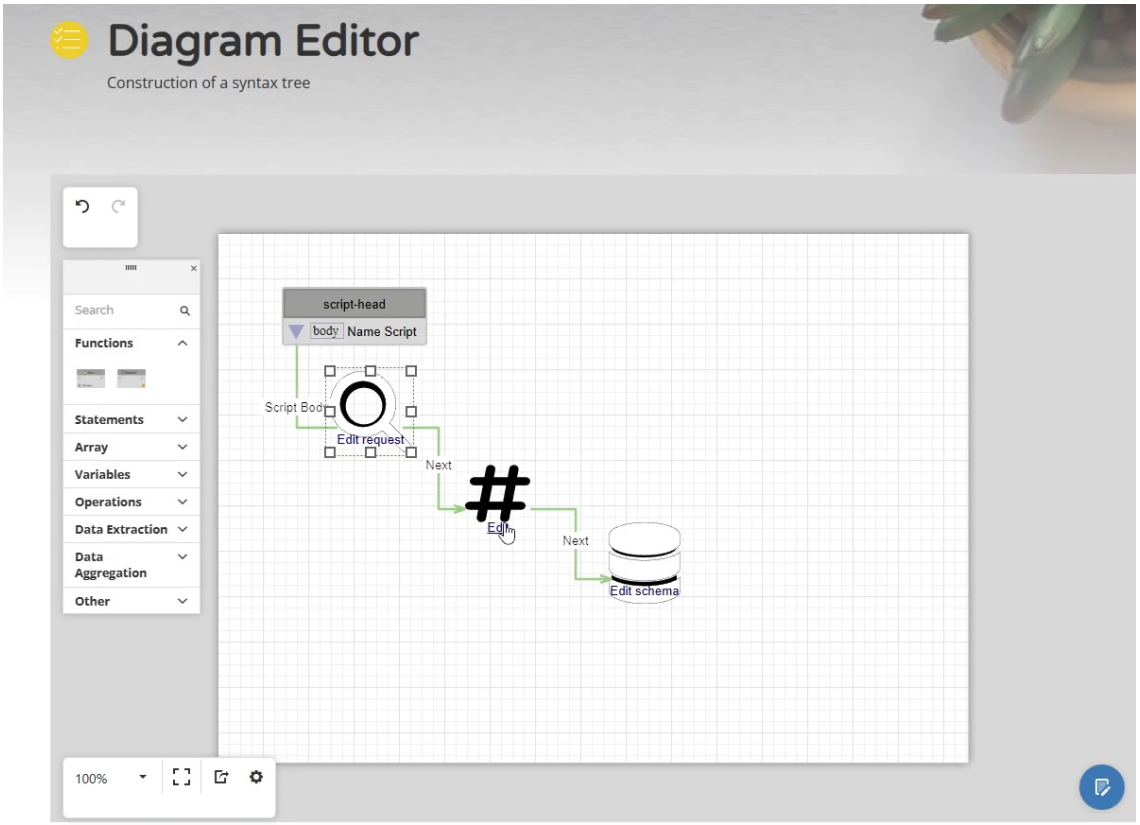


Figure 4.9: Snapshot of the Diagram Editor with the new "count by" shape.

At the back-end level, we added a new node to the parser: the "ASTCountBy" node. This node received the two parameters mentioned earlier. The system then simply invoked the "countBy" method from the Lodash library.

To support other types of aggregations, we would need to introduce new shapes at the front-end level. Besides, each shape would open a slightly different pop-up, depending on the required parameters. At the back-end level, the parser would need to be extended, and the corresponding new aggregation methods from the Lodash library would be called. It's worth noting that the parser also offers support for an "ASTGroupBy" node, despite the front-end not yet having a corresponding shape, nor visual editor for it.

Another important aspect to consider is that this interface initially contained two text forms: one for entering the property name and another for entering the collection name. However, this approach was replaced by the much more practical single drop-down menu. With this transition, the code implementing the Aggregation Editor required some changes. Populating the drop-down menu with the properties was simple, as the "SelectBox" component from DevExtreme already provided this functionality. Nevertheless, we added a function to handle updates to the variables that stored the collection and the count by property.

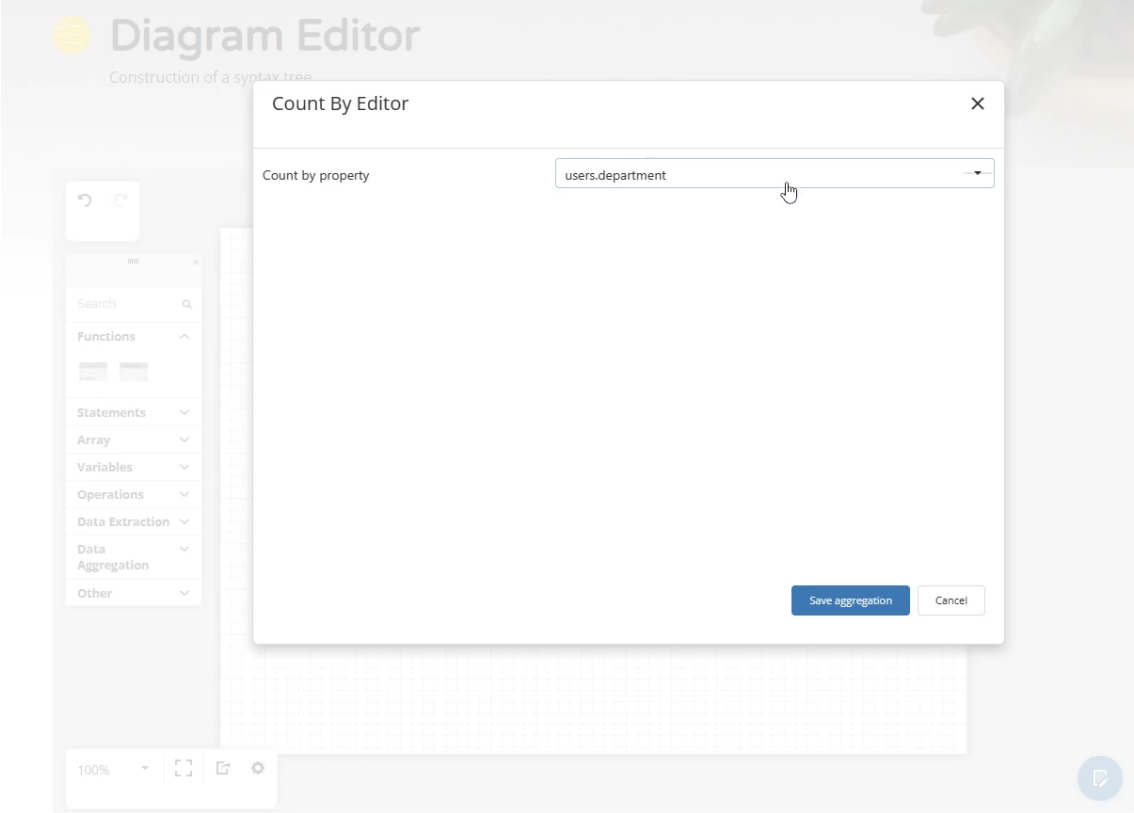


Figure 4.10: Snapshot of the Aggregation Editor.

4.10 The need for filters in the queries

4.10.1 Schema introspection

Schema introspection refers to the process of querying the structure and metadata of a data schema or data model at runtime. This subsection is necessary because the syntax for filters in GraphQL is not universal and depends on the implementation of the server being used. To recognize the syntax (if the server does supports filtering), the system must perform schema introspection.

4.10.2 Implementation

Filters are an essential feature in queries. We introduced two possible methods for adding a filter: either by dragging a specific ”filter” shape into the Diagram Editor, or by constructing it inside the Visual Query Builder component. In either case, the user utilizes a component called the Filter Editor to define the filter without the need for coding.

The first method’s existence is justified for possible scenarios that may be supported in the future. For instance, a user might want to request data from two separate sources and merge the results. In this case, dragging the ”filter” shape would allow filtering the resulting merged data.

Moreover, having a specific "filter" shape was useful due to the prototype's limitations, which did not support schema introspection. Therefore, we assumed that the user was aware of the filtering capabilities of the queried data source. If the data source did not support a specific filter, the user could connect the "api-request" and the "filter" shapes. First, a simple request was made, and then the resulting data was filtered using JavaScript functions. It is worth mentioning that this only happened when the user intended to generate GraphQL. Otherwise, for SQL generation, for example, filters could be attached to the request[2] because the syntax is consistently structured.

The usage of the "filter" shape is depicted in figures 4.11 and 4.12. The first one shows a syntax tree with three shapes, while the second one presents the component that appears after clicking the "edit" button attached to the "filter" icon. To replicate this scenario the user can click the "plus" icon in the Filter Editor to add a new filter condition. Then, the user can select a field (in this case, "users.department") from the drop-down menu containing all the query's child properties. The operator ("equals") and the value ("Development") are also user inputs.

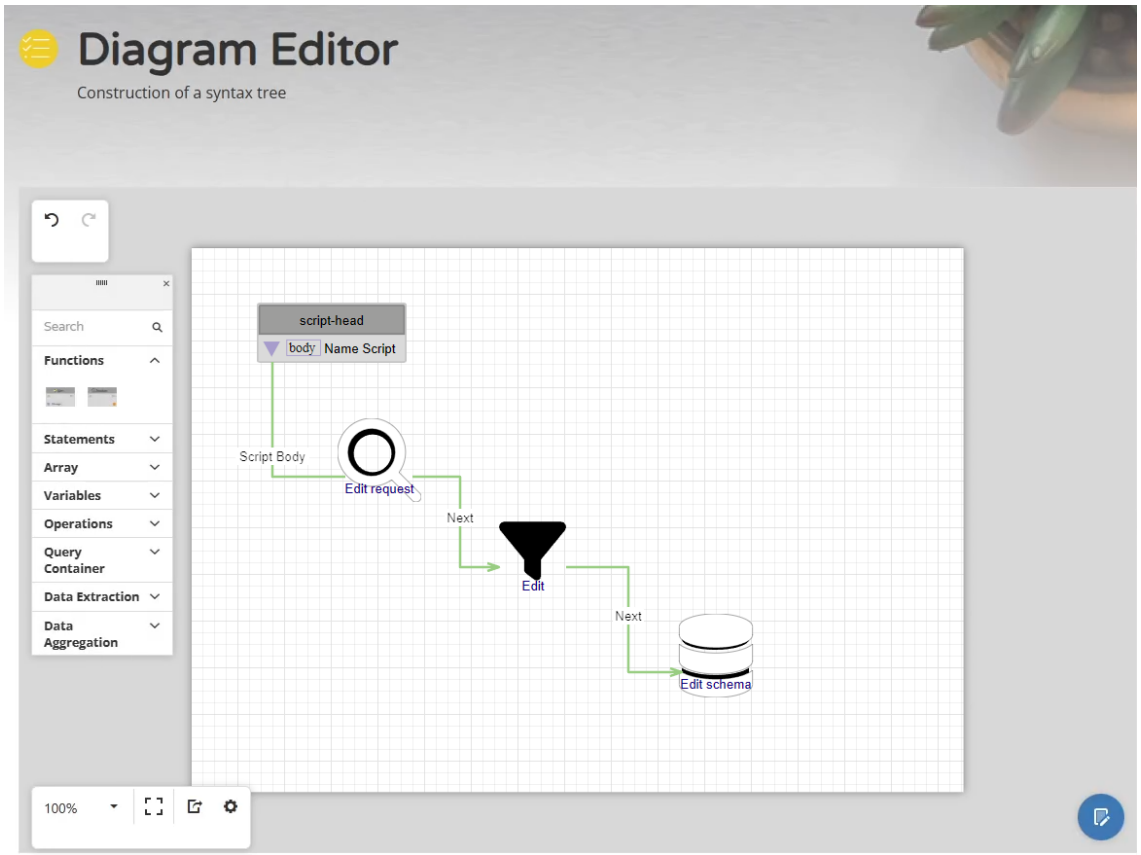


Figure 4.11: Snapshot of the Diagram Editor with the new "'filter" shape.

The second method of adding a filter was very intuitive. When constructing a query on the Visual Query Builder component, the user had the option to select an entity and

[2]Currently, the system supports only a limited range of SQL filters, as addressed in subsubsection 6.1.0.2.

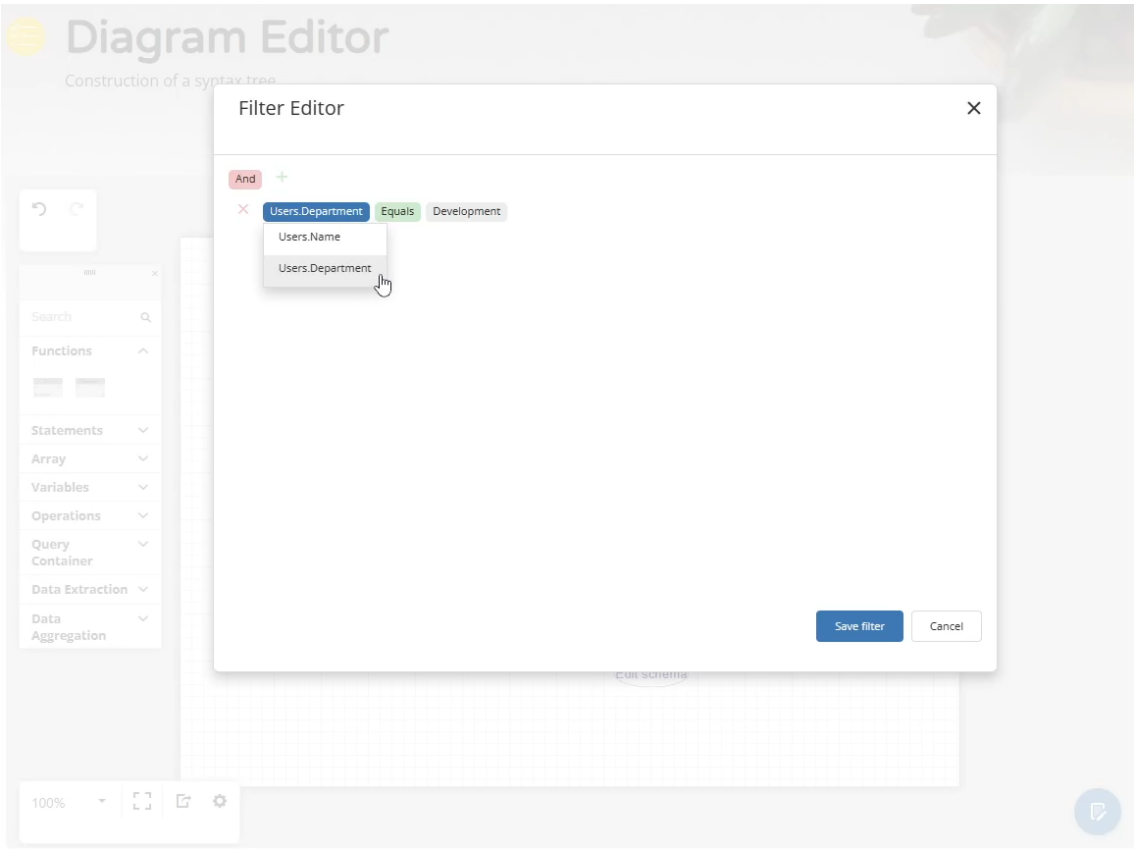


Figure 4.12: Snapshot of the Filter Editor.

click on the ”filter child nodes” button. The Filter Editor would then appear. The only difference between this component and the one in the first method was the content of the drop-down menu for the first argument, which included only the child nodes of the previously selected property.

Figures 4.13 and 4.14 provide an example of applying a filter while using this second method. In this case, only two nodes were needed for the tree, and the end result was the same as that obtained by using the ”filter” shape.

Regarding the back-end, there were differences between the two options for adding a filter. These differences are explained in the following two paragraphs.

In the first one, an ”ASTFilter” node was introduced to the parser. This dedicated node received the filter, the name of the variable where the data request’s response was stored, and the randomly generated name of the variable where the filtered data would be stored. The filter came in a specific format defined by the output of the ”Filter Builder” [39] component of DevExtreme. For instance, the filter constructed in the Figure 4.12 was represented as an array: [”users.department”,”=”,”Development”]. We needed to write a function to convert this format into a format accepted by the ”filter” method of Lodash. In this case, the function would return the following string: ”users.department == ’Development”’. Currently, the system only supports simple filters,

such as the one in the example, without logical operators. Finally, the system invoked the "filter" method from the Lodash library.

In the second option, at the back-end level, before the GraphQL translation, all received filters needed to be converted. It is worth noting that the query generated by the Visual Query Builder component was a tree, and each node of the tree contained a "filter" field. As the system was progressively generating the GraphQL query, if the current node had a non-empty "filter" field, the system would introduce the filter conversion inside parenthesis in front of the current entity. This time, the same filter from the previous example could be converted to the following: "users (where:{department:{eq: 'Development'}})". I say "could be" because this prototype does not yet have schema introspection, so the system assumes the filtering syntax. After this, in the front-end, a new small filter icon would be added to the corresponding filtered parent property, as it is visible in the Figure 4.14.

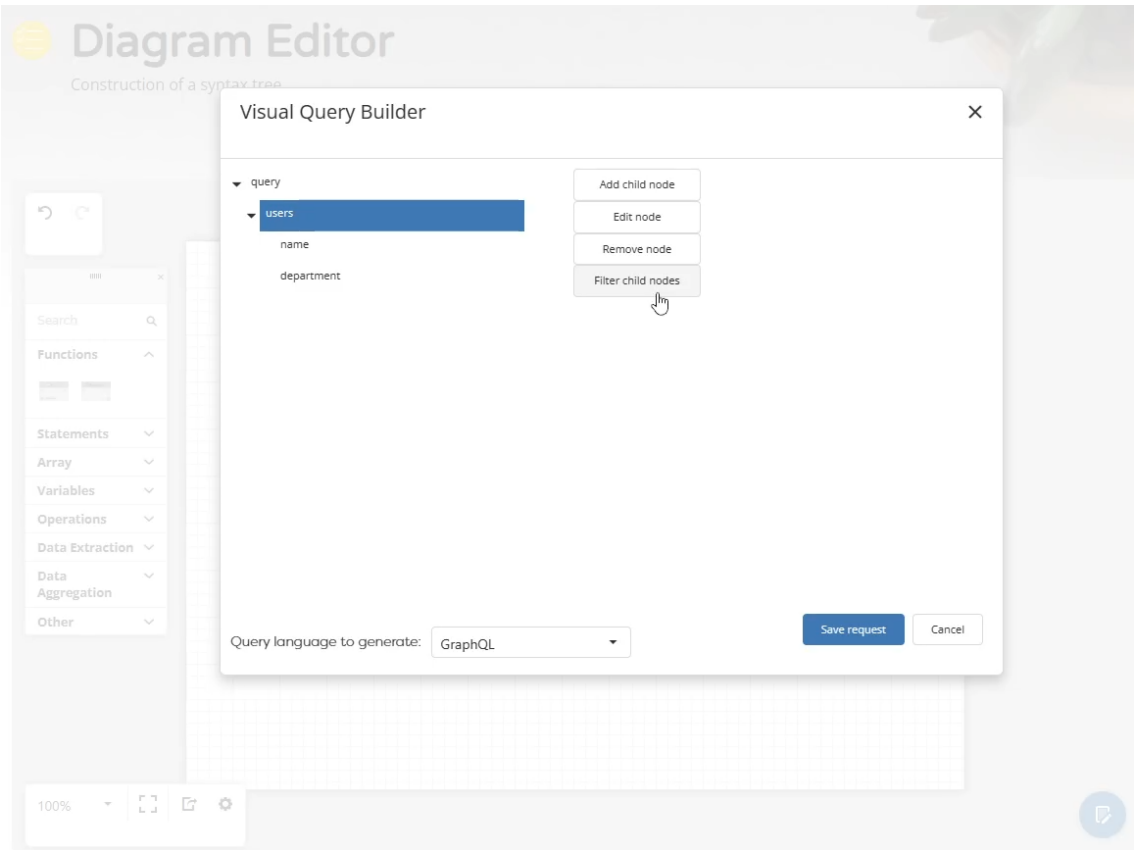


Figure 4.13: Snapshot of the Diagram Editor before adding a filter.

4.11 Option to make requests using other query languages

To accommodate requests from heterogeneous data sources, we needed to support more query languages other than GraphQL. Adding support for SQL requests would introduce

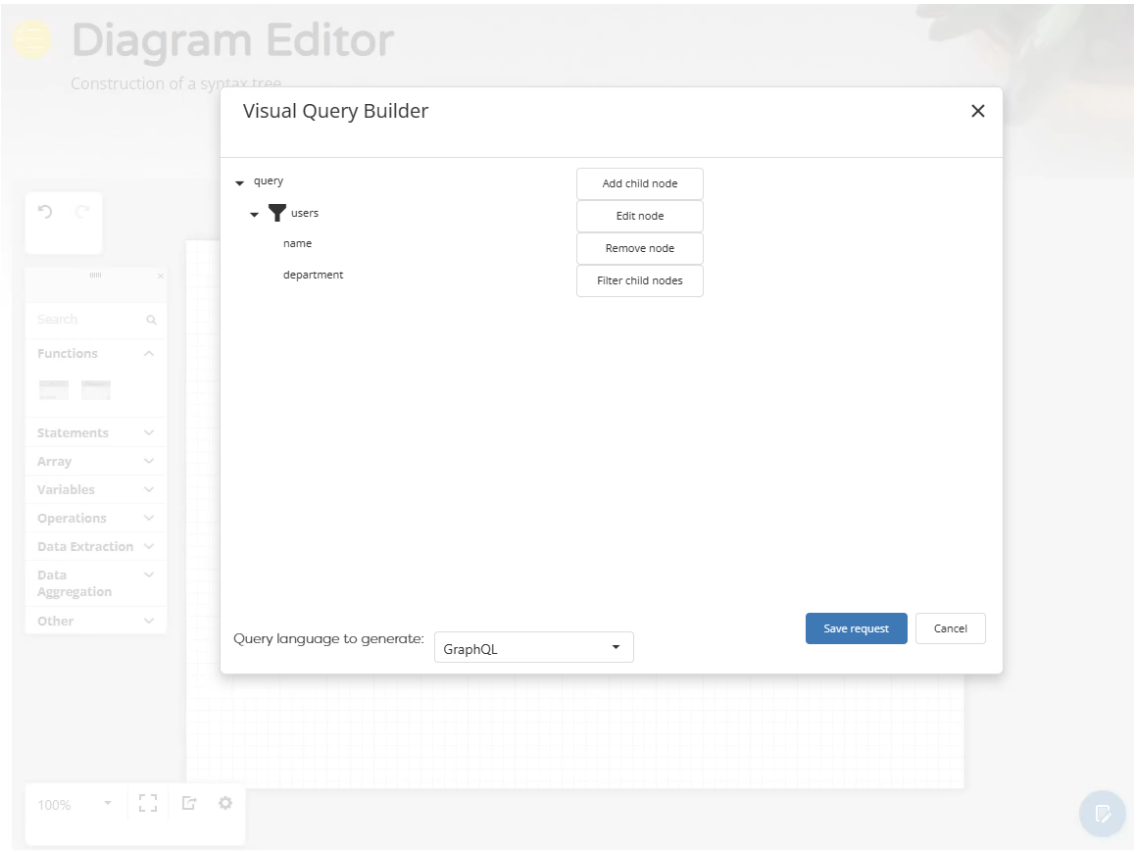


Figure 4.14: Snapshot of the Diagram Editor after adding a filter.

heterogeneity. Furthermore, we anticipate the potential need to support other query languages in the future, following a similar approach.

We introduced a simple drop-down menu (visible in the Figure 4.13) in the Visual Query Builder. This menu contained the available query languages to generate. Visually, the query structure the user needed to construct remained the same.

In the back-end, the Compiler service was also updated - we introduced a parser to generate SQL. Initially, the main parser detected an ”ASTApiRequest” node, and then, based on the parameter specifying the language to generate provided by the user, the system called one of the two available parsers: the GraphQL parser or the SQL parser. These parsers were responsible for reading the query within the syntax tree and translating it into the corresponding language. Currently, some fixed request options are required, including authentication. Furthermore, the code only supports the translation of simple queries, without joins, filters, and other complex features such as ”COUNT” or ”GROUP BY”, for example. These limitations are addressed in subsubsection 6.1.0.2.

4.12 Option to add a recurring schedule to the request

4.12.1 Scheduling libraries and tools

Firstly, a recurrence rule is an expression that follows a standard pattern and describes the schedule for a recurring event. Specifically, it can be used to define the frequency, interval, and duration of an event. The iCalendar (RFC 5545) [40] is commonly employed to represent such recurrence rules.

In order to create a component that enables users to establish a schedule for a task without requiring coding skills, an intuitive interface is essential. These rules must be generated automatically according to the user's selections. Furthermore, the system must possess the capability to interpret these recurrence rules and execute tasks at the right times. Numerous open-source libraries are available for these purposes. The following subsections introduce a selection of them.

4.12.2 React RRule Generator

The React RRule Generator [41] is an open-source React project that provides a no-code visual interface for creating recurrence rules.

The Figure 4.15 shows part of the interface without any modifications. Obviously, to integrate it in the Skills Workflow's platform, adaptations are necessary. For instance, the "RRule" row should be removed, as users do not need to be aware of the generated recurrence rule in iCalendar format.

A limitation of this component is that the minimum interval for a recurrence rule is one hour. Fortunately, within the context of the Skills Workflow's platform, the need for tasks to be executed more frequently than once per hour is rare. Nevertheless, to address this limitation, we have two options: either extend the existing component or create a new component from scratch.

In the first case, only the UI and its output generation (prior to parsing to RFC format) would be modified. This is because the React RRule Generator project utilizes rrule.js [42], a library that already supports intervals with very fine granularity, such as seconds.

In the second scenario, we would need to develop a new component capable of generating a representation of a recurrence rule, which could then be parsed for executing tasks at the corresponding times. Regarding the referred execution process, the node-cron [43] library could prove useful, as it offers a convenient way to schedule and run tasks at specific intervals, ranging from seconds to months. However, generating an expression in the specified format would be necessary. For instance, if we wanted to represent "running a task every two minutes" in the node-cron format the expression would be "*/2 * * * *". On the other hand, in the RFC 5545 format it would be "RRULE:INTERVAL=2;FREQ=MINUTELY". When it comes to representing more complex recurrence patterns, it is possible that the node-cron format is not expressive enough to do so, or that the representation is not intuitive and elegant. Thus, it would take

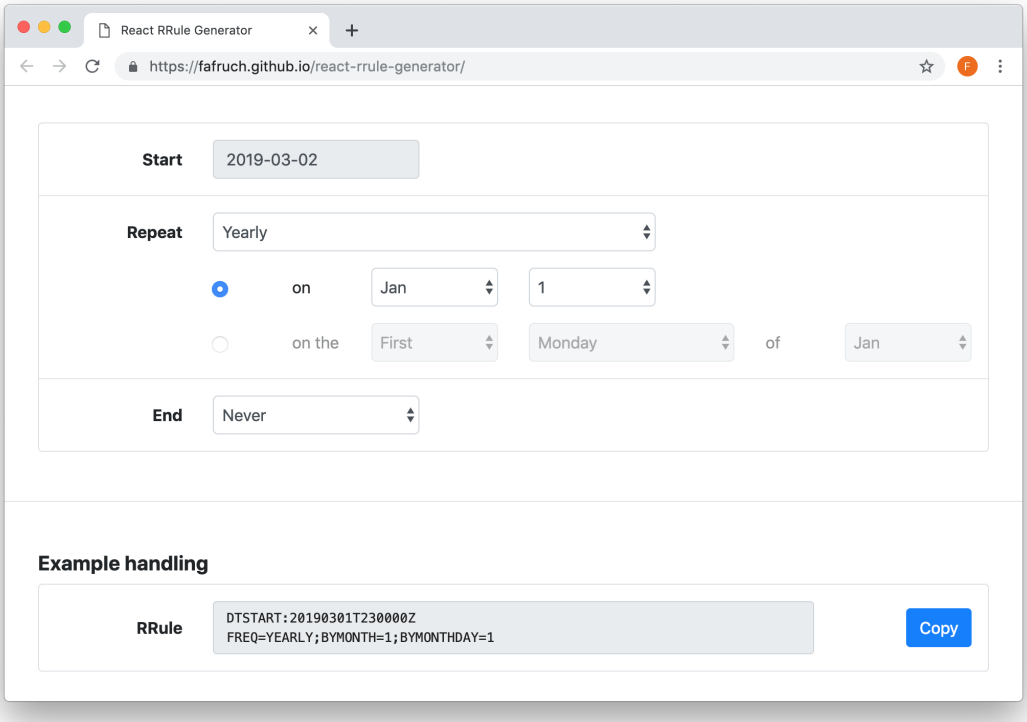


Figure 4.15: Snapshot of the React RRule Generator's original interface.

significant time and effort to create a component that generates all possible recurrence expressions in the node-cron format.

4.12.3 Node Schedule RRule

The Node Schedule RRule [44] library creates and executes scheduling jobs. As parameters, it requires a recurrence rule in iCalendar RFC format and a function to be executed. Although the project's GitHub repository mentions the need for some performance testing and optimization, it functions well for the simpler recurrence rules typically used in the Skills Workflow's platform.

4.12.4 Implementation

To introduce the possibility of attaching a recurring schedule to the user's constructed request, we updated the UI and developed a new service.

Regarding the interface, we attached a switch to the right side of the Diagram Editor, next to the "save diagram" button, as we can see in the Figure 4.16. This way, after constructing a syntax tree, the user could click on the switch to enable the scheduling option. Subsequently, a pop-up, known as the Schedule Editor, appeared (Figure 4.17), containing the React RRule Generator component with some adaptations, as mentioned

before. The user could also define a schedule, close the pop-up, and later reopen it to edit the schedule by clicking the ”edit schedule” button. The syntax tree’s structure was updated to include a ”schedule” node containing the specified recurrence rule in iCalendar RFC format.

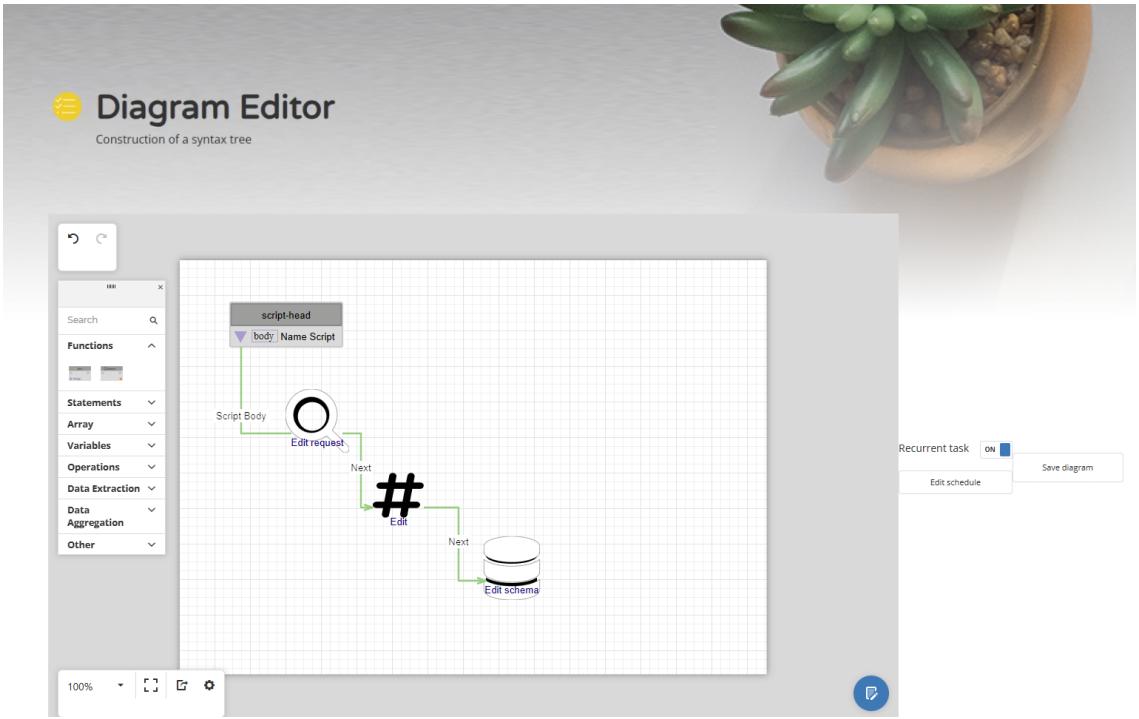


Figure 4.16: Snapshot of the Diagram Editor with the new scheduling switch.

On the back-end, if the user did not specify a schedule, the system would simply execute the request once, as described earlier. Otherwise, the system would invoke the newly created Scheduling service. The Compiler service made these calls and did not wait for a response to avoid actively waiting and becoming unavailable for other requests. In turn, the Scheduling service was responsible for asynchronously executing the request at the specified moments, utilizing the Node Schedule RRule library and Node.js.

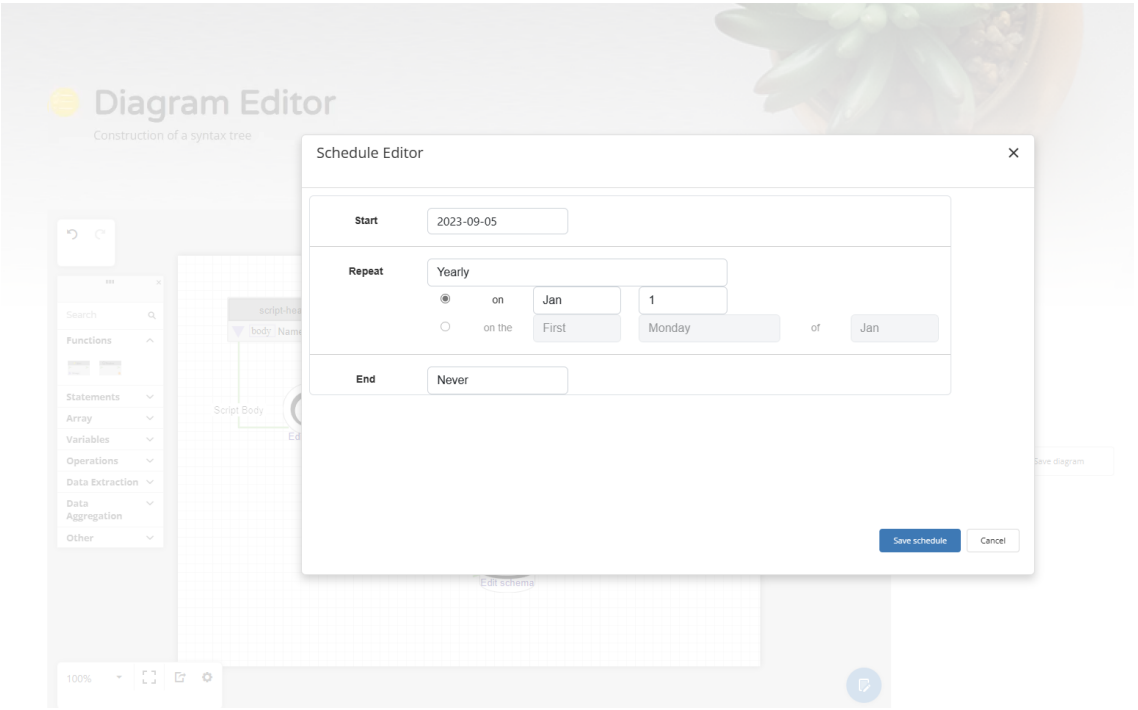


Figure 4.17: Snapshot of the Schedule Editor.

5

Validation

This chapter presents a qualitative and quantitative validation of the prototype. The last section aims to discuss if the produced solution is a good one, taking into account the validation, the proposed idea, and the solutions presented in the State of the art chapter.

5.1 Qualitative validation

In this section, we present three relevant use cases for the prototype. These cover the most common situations users are likely to encounter, along with a comprehensive range of components and features. It is very important to note that executing these examples requires no input in the form of code.

The first use case aims to retrieve and store all users belonging to the "Development" department. This data is found in the Skills Workflow's database, queried via GraphQL. Given the visual nature of this thesis, certain examples, including this one, have already been provided in chapter 4. The Figure 4.11 shows the three nodes necessary for constructing a syntax tree to request, filter, and store data. By clicking the "edit request" button to open the Visual Query Builder, the user can compose the query, as demonstrated in the Figure 4.13. The Figure 4.12 showcases the filter construction within the Filter Editor after clicking the "edit" button on the "filter" icon. Alternatively, the user may create a syntax tree with just two nodes, without the "filter" shape. In this case, the filter construction is done the same way, but occurs within the Visual Query Builder by selecting the "users" node in the query and clicking the "filter child nodes" button. As explained in subsection 4.10.2, a new small icon appears to indicate where the filter is applied (Figure 4.14). To edit the storage details, the user should replicate the setup depicted in the Figure 5.1.

The second use case's goal is to count users by department. The data fetched comes from the same data source as the previous example. Besides, this use case has already been explained previously in this document. The Figure 4.9 shows the syntax tree the user must create for this purpose, while the Figure 4.10 displays the Aggregation Editor's interface for counting users by department. The query constructed in the Visual Query Builder is the same as in the previous example. The schema definition is different, as

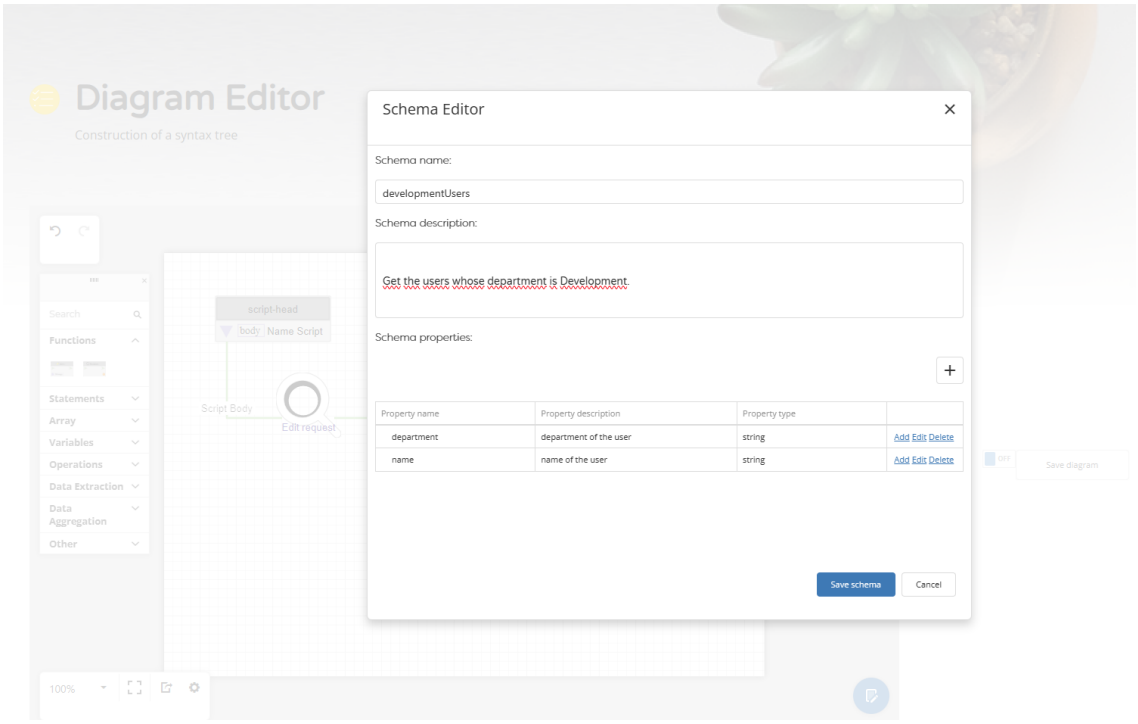


Figure 5.1: Schema Editor snapshot for storing users whose department is "Development".

shown in the Figure 5.2.

Finally, the third use case involves retrieving every month the names and IDs of all companies. The request is be made do a data source via SQL. The Figure 5.3 shows the two nodes the user must drag into the diagram. The query construction is visible in the Figure 5.4, while the schema definition is shown in the Figure 5.5. Lastly, the schedule is defined in the Figure 5.6.

5.2 Quantitative validation

This subsection compares the time required for defining a certain use case using the no-code interface with the time spent writing code to achieve the same outcome. Obviously, this comparison overlooks several factors. For instance, individuals may differ in their proficiency using the prototype, and the time spent writing code varies based on a person's programming skills. Despite these variables, our goal is to provide a rough estimate of the time invested in each approach. This comparison is performed by testing the second use case introduced in the preceding section.

This paragraph explains the time calculations for the no-code approach. Here, we assume that the user conducting the test is familiar with the interface. The actions of selecting, dragging, and connecting the shapes on the Diagram Editor takes approximately 15 seconds. Constructing the query in the Visual Query Builder consumes roughly 20 seconds. Selecting the "count by" property and defining the schema take around 5 and 25

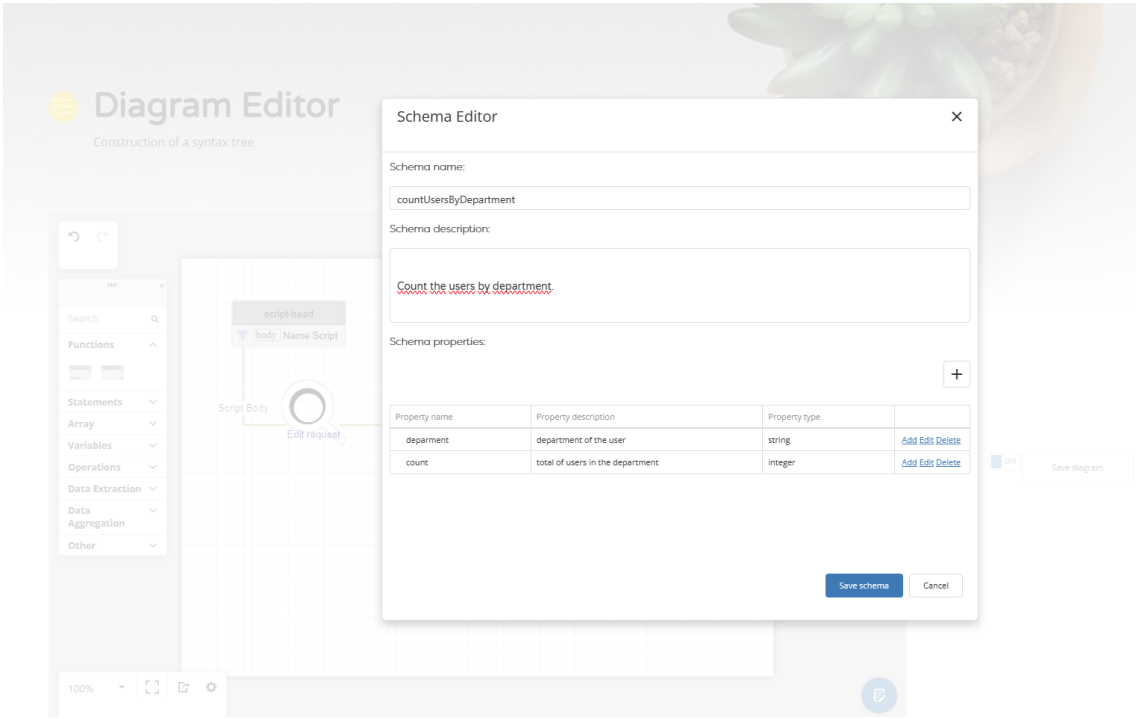


Figure 5.2: Schema Editor snapshot for counting users by department.

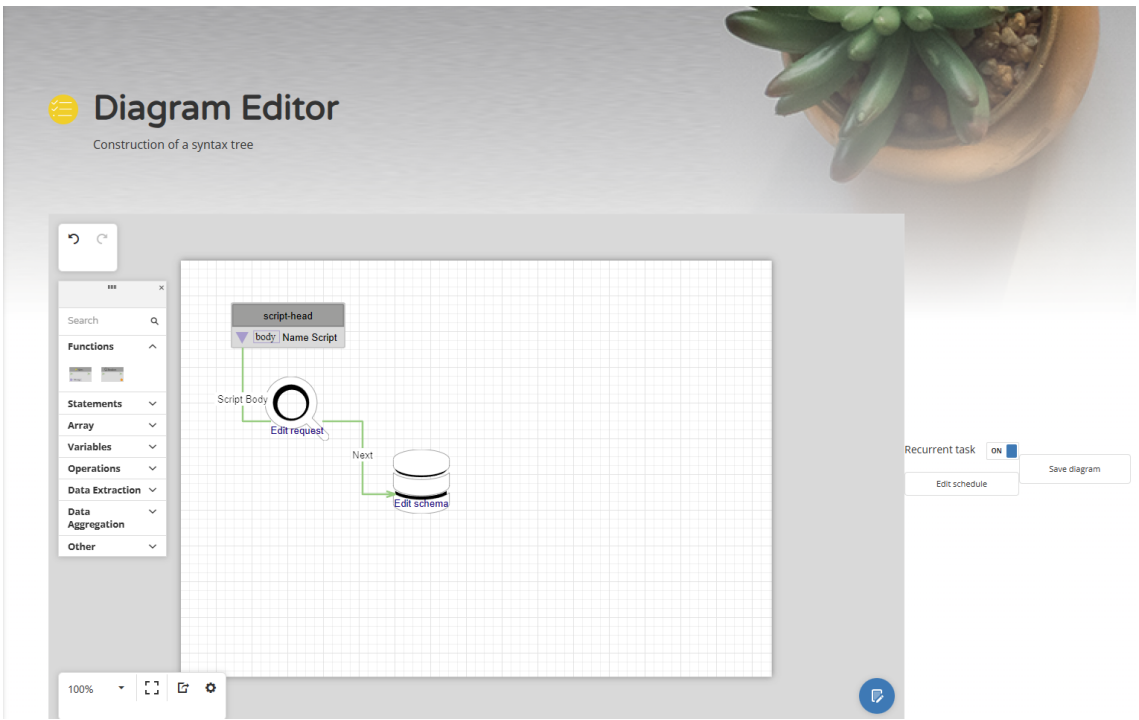


Figure 5.3: Diagram Editor snapshot: syntax tree for retrieving and storing the names and IDs of all companies.

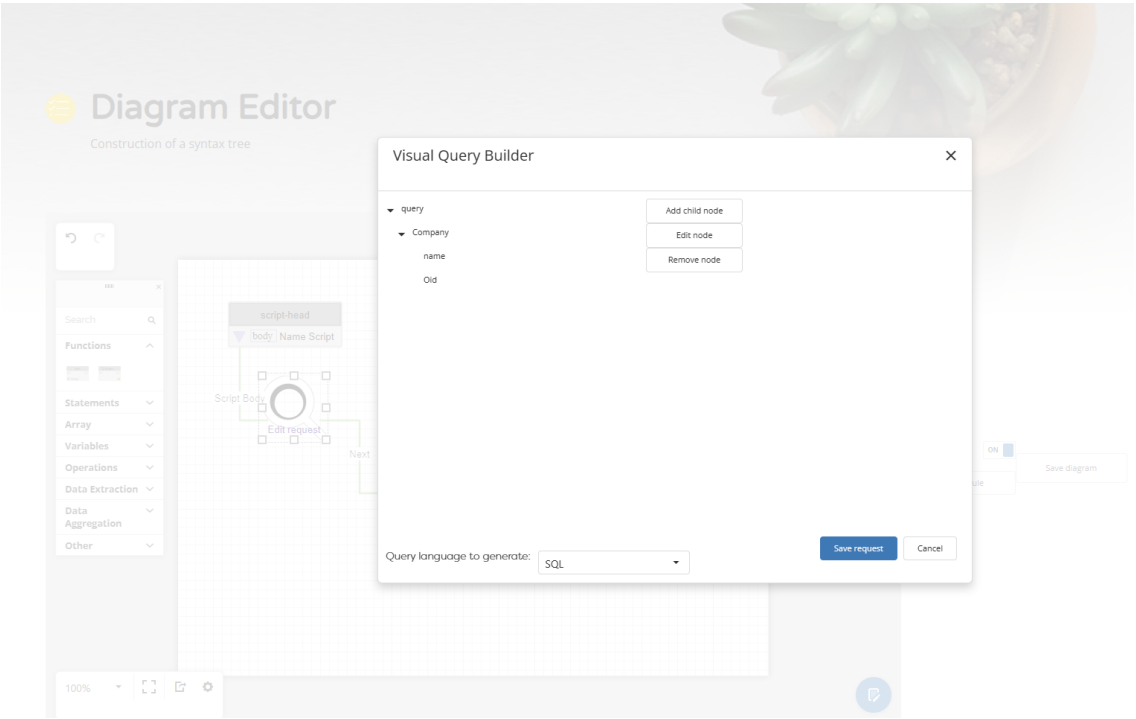


Figure 5.4: Visual Query Builder snapshot: query for retrieving the names and IDs of all companies.

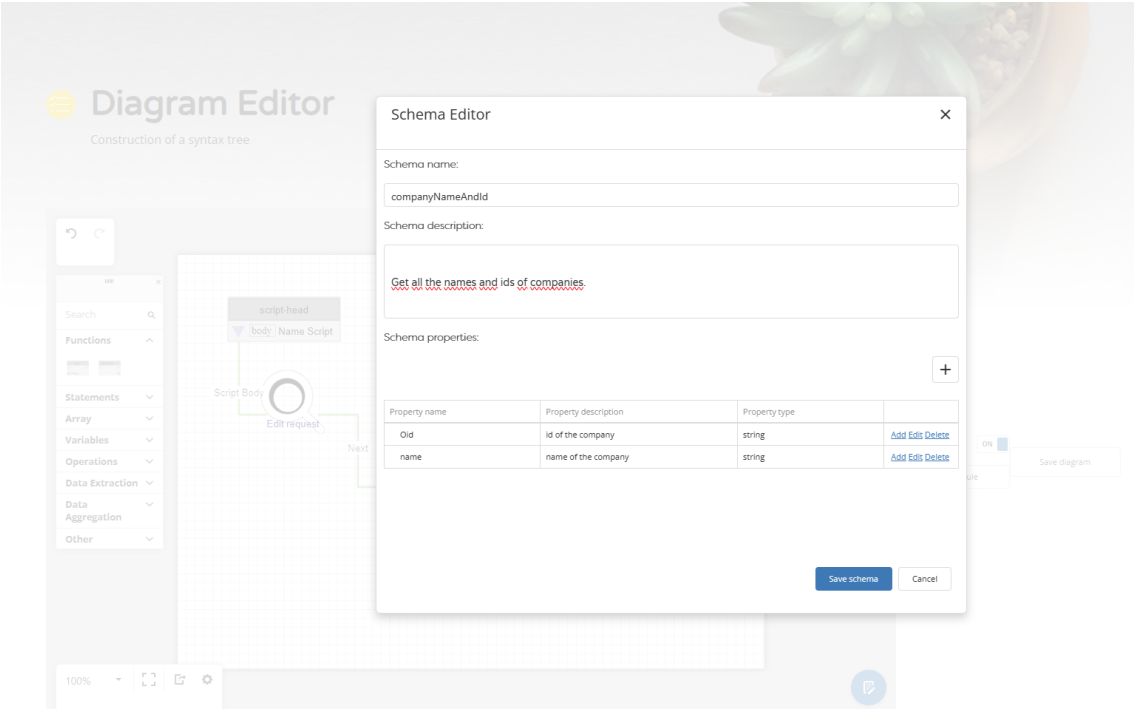


Figure 5.5: Schema Editor snapshot: details for storing the names and IDs of all companies.

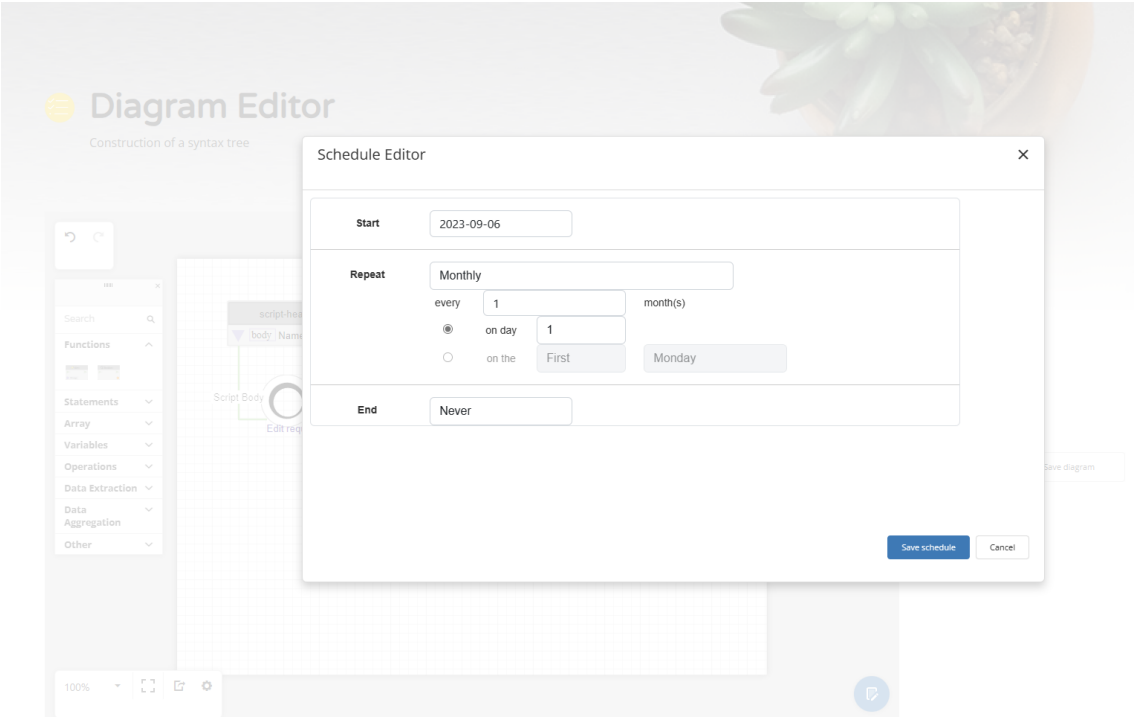


Figure 5.6: Schedule Editor snapshot: inputs for a monthly schedule.

seconds, respectively. Thus, the total time spent on this approach amounts to 65 seconds.

Now, let's consider the second approach, assuming that the user conducting the test is adept at typing and knows exactly what to input in advance. The Figure 5.7 displays a portion of the JavaScript code generated by the system for this use case, specifically corresponding to user inputs. The remaining lines of code consist of generic functions that remain consistent across all use cases. Hence, the time required for writing the code is approximately 5 or 15 minutes[1], depending on whether we include the generic functions or not.

5.3 Global analysis

To conclude if the achieved solution is successful, we must consider the validation presented in the previous sections, the proposed idea, and the solutions presented in the State of the art chapter.

The qualitative validation accesses the quality of the prototype based on the functionalities it can demonstrate. Although the three use cases involve relatively simple queries, they test common scenarios in data extraction and require no input from the user in the form of code. In these cases, the prototype has proven to be successful.

[1]Assuming a typing speed of 50 words per minute, or 250 characters per minute (if the average word length is 5 characters), and knowing the user input code comprises 1391 characters without spaces, the time required to write the code is around 5 minutes (1391/250). For the generic functions, which contain 3785 characters without spaces, the time required is approximately 15 minutes (3785/250).

```
let responseVarName812b2b37d4843cfe0367b60e5c7ed6b0Var;
const responseVarName812b2b37d4843cfe0367b60e5c7ed6b0 = async () => {
    responseVarName812b2b37d4843cfe0367b60e5c7ed6b0Var = await makeApiRequest({
        "hostname": "envoy-gw.orangesmoke-c07594bb.westeurope.azurecontainerapps.io",
        "path": "/730e1e92-8204-4303-af3b-b4a4cca9929e/document-api/graphql",
        "method": "POST",
        "headers": {"Content-Type": "application/json"}
    }, {"query": "query {\n\tusers {\n\t\tdepartment\n\t}\n}\n"}, 'responseVarName812b2b37d4843cfe0367b60e5c7ed6b0');
    return responseVarName812b2b37d4843cfe0367b60e5c7ed6b0Var;
};

let countByResultVarNamedd65af623c05ea0db6a2df9cd82d2b59Var = null;
async function countByResultVarNamedd65af623c05ea0db6a2df9cd82d2b59() {
    const apiResponseObj = await responseVarName812b2b37d4843cfe0367b60e5c7ed6b0();
    countByResultVarNamedd65af623c05ea0db6a2df9cd82d2b59Var = lodHashCountBy(apiResponseObj.data.users, ((usersSingleObject) => (usersSingleObject.department)));
};

countByResultVarNamedd65af623c05ea0db6a2df9cd82d2b59().then(() => {
    genericStore(countByResultVarNamedd65af623c05ea0db6a2df9cd82d2b59Var, 'countUsersByDepartment', {
        "name": "countUsersByDepartment",
        "pluralName": "countUsersByDepartment",
        "jsonSchema": {
            "$schema": "https://json-schema.org/draft/2020-12/schema",
            "type": "object",
            "description": "Count the users by department.",
            "required": ["count"],
            "properties": {"count": {"type": "object", "description": "total of users by department"}}
        }
    });
});
```

Figure 5.7: Specific part of the code required to count users by department.

The quantitative validation process allows us to compare the time taken when using the prototype versus writing code. We conclude that the first approach is at least 5 times faster than the second one. Although the previous type of validation is more relevant (because speed of usage is not the focus on this thesis) this result still confirms that this prototype can be convenient not only for individuals lacking programming skills but also for software developers.

When comparing the prototype with the proposed idea, we have successfully achieved the goal of developing an interface for the Super-creator to supply data for the Creator to construct workspaces.

Finally, the related projects in section 2.6 have different goals and must be compared to the prototype individually. The Tsimmis project lacks a visual interface and instead focuses on unifying heterogeneous data sources. In our thesis, this unification was accomplished in the sense that the data supplied to the Creator is stored by the Super-creator in a single data source, despite it originally coming from heterogeneous data sources. The second project, Lion, not only deals with data integration but also includes a visual query builder designed for individuals without programming skills. In contrast to our Visual Query Builder, the one developed in the referred project (shown in the Figure 2.10) contains too much information, with many unselected fields visible, resulting in a confusing interface with a small font and numerous connecting lines. Therefore, we believe our solution presents improvements in this regard.

6

Future Work and Conclusions

6.1 Future Work

As we have seen, this prototype is successful considered the initial goals. However, there is still room for improvement. In this section, we will discuss some of the possible future work.

6.1.0.1 The interface for the Creator

Throughout this document, the notions of Super-creator, Creator, and End-user have been mentioned several times. However, only an interface destined for the Super-creator is implemented in this prototype. It is missing a similar UI, but for the Creator. Both interfaces could similarities. In fact, the Diagram Editor and the components it comprises could be introduced in the Creator's UI. Besides, the Creator would have access to the Super-creator's extracted and transformed data. With this data and with the help of some no-code additional features, the Creator would be able to create charts, tables, and other types of data visualizations.

6.1.0.2 More complex queries and filters

Right now, despite the user being able to construct very complex filters in the Filter Editor, the back-end only supports the most basic ones. In the future, we plan to improve the capabilities of the function that generates the code that calls the Lodash library, as it supports only a few filter operators, such as ”equals”, and ”greater than”. Moreover, it cannot yet handle logical operators.

Currently, the generated GraphQL filters are equally limited.

The range of SQL queries that the system can generate is also quite narrow. For example, it cannot yet generate queries with ”join”.

6.1.0.3 Schema introspection

The term *schema introspection* has been already defined in subsection 4.10.1. In this section, we will discuss the advantages of implementing it.

Firstly, schema introspection would enhance the user experience in the Visual Query Builder. Instead of manually typing the field name for extraction, the user could benefit from field autocompletion or a drop-down menu featuring available fields.

As mentioned in subsection 4.10.2, schema introspection proves particularly useful when the user is extracting data from a GraphQL server, as the filtering syntax depends on its implementation. In this case, the system should be capable of automatically discovering the syntax of filters, thus abstracting this task from the user. Subsequently, it would generate a filter in the accepted syntax. If the server did not accept filters, the system would initially make the request without filters and then perform an aggregation using the filter function from the library Lodash.

Furthermore, this functionality could resolve another issue in the Filter Editor. Filters consist of properties, operators, and values. The system should automatically determine the type of the selected property and only present the corresponding valid operators. For example, if the property is a string, the ”equals” operator should be available, while the ”is between” operator should not (because it is used for numbers).

6.1.0.4 Dynamic request options and support for more types of data sources

Currently, according to the chosen data source, the system assumes automatically many options for the request, such as the hostname, the path, the method type (POST, GET, etc), and the headers. In the future, the user should be able to change these options dynamically, with a no-code approach.

Similarly, the system also manually passes authorization secrets to the headers of the request. A new solution should be implemented to avoid this.

Furthermore, more types of authentication should be supported (with a no-code approach in the UI), such as OAuth 2.0 [45], for example. This way, the prototype would be able to request data from a wider range of data sources outside the Skills Workflow's platform.

6.1.0.5 Syntax tree with more branches

Currently, we only support a syntax tree with one continuous branch. In the future, we should allow the user to merge two branches into one. This would be useful, for example, if the user wanted to make a request to two different data sources and store the results in the same schema.

6.1.0.6 Issues with the storage function

The current implementation of the data storage function lacks generality. Ideally, this function should exclusively receive data and store it in a generic manner. Right now, it accepts data and stores it based on the incoming format, which is not a scalable approach. For instance, different Lodash functions return results with different structures. When adding support for a new function, preferably the storage function should remain the same.

Additionally, the function currently removes the old instances of the schema before storing the new ones. This may not align with the user's preferences. Originally, we implemented this for testing purposes to maintain a cleaner database, and it has remained as such. While it is a valid approach, the deletion step should ideally be optional. Essentially, the current storage strategy does not allow users to incrementally store data.

Another constraint of this function is that it consistently stores data in a fixed data source. In the future, this should be a parameter that the user could specify.

6.1.0.7 Create wrappers for all components

Creating wrappers for all components in the system would bring several advantages, namely modularization, reusability, and code organization.

Currently, as mentioned in **??**, there is already a wrapper for the Diagram Editor. However, as the implementation process evolved, we ended up including the other components of the prototype inside this wrapper. This is not ideal because it makes the code less modular. Each component should have its own wrapper.

6.1.0.8 Data extraction with artificial intelligence

As we previously discussed in this document, this prototype could benefit from the use of artificial intelligence. We believe it is possible to request, transform, and store data using text or even speech commands.

To be more specific, the goal would be to transform text commands into the already existing AST nodes. Thus, big part of the work developed in this thesis would be reused.

6.2 Conclusions

In conclusion, this master's thesis successfully addresses the problem of data extraction from heterogeneous data sources using a no-code approach. This prototype is a great new innovative tool for the SKills's Workflow platform, as it is a step forward in the direction of allowing users to create their own workspaces without the need for programming skills.

During the development of this prototype, I perfected my skills in JavaScript, TypeScript, React, Postman, and Git. I also learned GraphQL, which is a technology gaining widespread recognition. Moreover, I used Node.js for the first time, along with many

libraries. The incorporation of the Agile Scrum methodology by Skills Workflow not only enriched my experience but also introduced me to a prevalent practice in the software industry.

Considering potential future enhancements, if I had more time to continue the development of this project, I would first focus on implementing schema introspection, as it would be a great improvement in the user experience. Then, I would work on the interface for Creators to complete the process of extracting, transforming, storing, and visualizing data in a workspace. After this, I would possibly introduce intelligent capabilities to the prototype. This step presents an exciting challenge and could potentially be a topic for a future master's thesis.

Regarding things I would do differently, given the practical nature of this master thesis for being developed in an enterprise environment, I would have balanced theoretical and practical aspects during the initial semester. This approach would have provided a better understanding of existing company code and tools, such as the Diagram Editor from DevExtreme, which played a crucial role in the prototype's development. Knowing about it would have gave me a clearer view of what the prototype would look like.

In summary, one of my biggest achievements was developing a functional no-code UI with several components, including the Diagram Editor, which houses the Visual Query Builder, the Filter Editor, the Aggregation Editor, the Schema Editor, and the Schedule Editor. Another great accomplishment was the implementation of a back-end composed of a Compiler service and a Scheduling service. All these components and services work together to extract, transform, and store data from heterogeneous data sources, a contemporary challenge in the industry, as well as to solve the problem of large dataset extraction.

I take pride in the development of this comprehensive document that not only reports challenges, related concepts and work, implemented features, and a validation process but also provides a valuable reference for future work in this field.

Bibliography

[1] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on pp. ii, iii).

[2] *JSON Editor Online: JSON editor, JSON formatter, query JSON*. URL: https://jsoneditoronline.org/#left=local.sazutu (cit. on p. iii).

[3] G. W. Group. *GraphQL Specification Working Draft*. 2023. URL: https://spec.graphql.org/draft/ (cit. on pp. 4, 5).

[4] E. S. Ghebremicael. "Transformation of REST API to GraphQL for OpenTOSCA". 2017 (cit. on pp. 5, 12).

[5] D. James. "LION: Listen Online. Using GraphQL as a mediator for data integration and ingestion" (cit. on pp. 6, 20–22).

[6] *Introduction to Apollo Client - Apollo GraphQL Docs*. URL: https://www.apollographql.com/docs/react/ (cit. on p. 6).

[7] I. Lauriola, A. Lavelli, and F. Aiolli. "An introduction to Deep Learning in Natural Language Processing: Models, techniques, and tools". In: *Neurocomputing* 470 (2022-01), pp. 443–456. ISSN: 18728286. DOI: 10.1016/j.neucom.2021.05.103 (cit. on pp. 7, 8).

[8] D. Khurana et al. "Natural language processing: state of the art, current trends and challenges". In: *Multimedia Tools and Applications* (2022-01). ISSN: 15737721. DOI: 10.1007/s11042-022-13428-4 (cit. on pp. 7, 8).

[9] M. Mars. *From Word Embeddings to Pre-Trained Language Models: A State-of-the-Art Walkthrough*. 2022-09. DOI: 10.3390/app12178805 (cit. on pp. 7, 8).

[10] A. Vaswani et al. "Attention Is All You Need". In: (2017-06). URL: http://arxiv.org/abs/1706.03762 (cit. on p. 8).

[11] J. Finnie-Ansley et al. "The robots are coming: Exploring the implications of OpenAI codex on introductory programming". In: Association for Computing Machinery, 2022-02, pp. 10–19. ISBN: 9781450396431. DOI: 10.1145/3511861.3511863 (cit. on p. 9).

[12] *OpenAI Models - Documentation*. URL: https://beta.openai.com/docs/models (visited on 2022-01-26) (cit. on pp. 9, 10).

[13] I. Trummer. "From BERT to GPT-3 Codex: Harnessing the Potential of Very Large Language Models for Data Management". In: vol. 15. VLDB Endowment, 2022, pp. 3770–3773. DOI: 10.14778/3554821.3554896 (cit. on p. 10).

[14] M. C. Chirodea et al. "Comparison of Tensorflow and PyTorch in Convolutional Neural Network - Based Applications". In: Institute of Electrical and Electronics Engineers Inc., 2021-07. ISBN: 9781665425346. DOI: 10.1109/ECAI52376.2021.9515098 (cit. on p. 10).

[15] M. Corporation. *SQL Server Management Studio (SSMS)*. 2023. URL: https://learn.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver16 (cit. on pp. 10, 11).

[16] G. Community. *GraphiQL*. [Online; accessed 06-Feb-2023]. 2017. URL: https://github.com/graphql/graphiql (cit. on p. 12).

[17] G. Fusco and L. Aversano. "An approach for semantic integration of heterogeneous data sources". In: *PeerJ Computer Science* 2020 (3 2020), pp. 1–30. ISSN: 23765992. DOI: 10.7717/peerj-cs.254 (cit. on pp. 13, 15, 16).

[18] J. Chakraborty, A. Padki, and S. K. Bansal. "Semantic ETL-State-of-the-Art and Open Research Challenges". In: Institute of Electrical and Electronics Engineers Inc., 2017-03, pp. 413–418. ISBN: 9781509048960. DOI: 10.1109/ICSC.2017.94 (cit. on pp. 13, 14).

[19] S. Busse et al. *Federated Information Systems: Concepts, Terminology and Architectures*. Tech. rep. Technische Universität Berlin, Fachbereich 13 Informatik, 1999. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.23.5830 (cit. on pp. 14, 15).

[20] A. Bergvall. "Performance in Apollo Federation-A Controlled Experiment Evaluating the Effects of Execution Strategies and Number of Subgraphs". URL: https://lup.lub.lu.se/student-papers/record/9094853/file/9094856.pdf (cit. on p. 15).

[21] C. Labadie et al. "FAIR Enough? Enhancing the Usage of Enterprise Data with Data Catalogs". In: vol. 1. Institute of Electrical and Electronics Engineers Inc., 2020. ISBN: 9781728199269. DOI: 10.1109/CBI49978.2020.00029 (cit. on p. 17).

[22] *Apache Atlas – Data Governance and Metadata framework for Hadoop*. URL: https://atlas.apache.org/#/ (cit. on p. 17).

[23] *What is Caching and How it Works | AWS*. URL: https://aws.amazon.com/caching/?nc1=h_ls (cit. on pp. 17, 32).

[24] P. Kavitha. *A Survey on Lossless and Lossy Data Compression Methods* (cit. on p. 18).

[25] *The gzip home page*. URL: https://www.gzip.org/ (cit. on p. 18).

[26] S. Chawathe et al. "The TSIMMIS Project: Integration of Heterogenous Information Sources". In: 1994. URL: http://ilpubs.stanford.edu:8090/66/ (cit. on pp. 18, 19).

[27] *React*. URL: https://react.dev/ (cit. on p. 25).

[28] J. Zhang et al. "A Novel Neural Source Code Representation Based on Abstract Syntax Tree". In: vol. 2019-May. IEEE Computer Society, 2019-05, pp. 783–794. ISBN: 9781728108698. DOI: 10.1109/ICSE.2019.00086 (cit. on p. 26).

[29] *Node.js*. URL: https://nodejs.org/en (cit. on p. 26).

[30] *Azure Cosmos DB | Microsoft Learn*. URL: https://learn.microsoft.com/en-us/azure/cosmos-db/ (cit. on p. 28).

[31] *Lodash*. 2023. URL: https://lodash.com/ (cit. on p. 34).

[32] *URL | Node.js v20.7.0 Documentation*. URL: https://nodejs.org/api/url.html (cit. on p. 34).

[33] *DevExtreme Features - JavaScript UI Components for Angular, React, Vue and jQuery by DevExpress*. 2023. URL: https://js.devexpress.com/Overview/ (cit. on p. 35).

[34] *Angular*. URL: https://angular.io/ (cit. on p. 35).

[35] *Vue.js - The Progressive JavaScript Framework | Vue.js*. URL: https://vuejs.org/ (cit. on p. 35).

[36] *jQuery*. URL: https://jquery.com/ (cit. on p. 35).

[37] *Overview - DevExtreme Diagram: React Components by DevExpress*. URL: https://js.devexpress.com/Demos/WidgetsGallery/Demo/Diagram/Overview/React/Light/ (cit. on p. 36).

[38] *Hierarchical Data Structure - DevExtreme Tree View: React Components by DevExpress*. URL: https://js.devexpress.com/Demos/WidgetsGallery/Demo/TreeView/HierarchicalDataStructure/React/Light/ (cit. on p. 38).

[39] *Customization - DevExtreme Filter Builder: React Components by DevExpress*. URL: https://js.devexpress.com/Demos/WidgetsGallery/Demo/FilterBuilder/Customization/React/Light/ (cit. on p. 47).

[40] B. Desruisseaux. *Internet Calendaring and Scheduling Core Object Specification (iCalendar)*. RFC 5545. 2009-09. DOI: 10.17487/RFC5545. URL: https://www.rfc-editor.org/info/rfc5545 (cit. on p. 50).

[41] *Fafruch/react-rrule-generator: Recurrence rules generator form built with React*. URL: https://github.com/Fafruch/react-rrule-generator (cit. on p. 50).

[42] *jakubroztocil/rrule: JavaScript library for working with recurrence rules for calendar dates as defined in the iCalendar RFC and more.* URL: https://github.com/jakubroztocil/rrule (cit. on p. 50).

[43] *node-cron/node-cron: A simple cron-like job scheduler for Node.js*. URL: https://github.com/node-cron/node-cron (cit. on p. 50).

[44] *msageryd/node-schedule-rrule: An RRule based job scheduler for Node.* URL: https://github.com/msageryd/node-schedule-rrule (cit. on p. 51).

[45] *OAuth 2.0 — OAuth*. URL: https://oauth.net/2/ (cit. on p. 61).