Informatica & Economie

Euridice (Extensible Data Cleansing Framework)

Intuitive data quality classification and cleansing

Max Boone (s2081318)
github.com/maxboone/euridice

First Supervisor: Dr. Guus Ramackers
Second Supervisor: Dr. Jan van Rijn

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

August 9, 2022

Acknowledgements

It would not have been possible to build this framework and write this thesis without the support of many people. First and foremost, thanks to my supervisor Guus Ramackers, who on numerous occasions helped me get back on track with the work on this thesis after setbacks, and helped drive the project to something I expect to keep maintaining and improving as an open source contribution for the years to come.

Thanks to Irene Martorelli, whose work and ideas stood at the base of this project. The data from the MycoDiversity DB project was central to steering this project towards an end-to-end solution.

Finally, thanks to my significant other, Nathalie van Veen for her love, patience and support. I couldn't have done this without the critical reviews of the final drafts of the thesis, or without her listening to my pitches (that barely made sense) each time I got stuck at a crossroads in the software.

Abstract

Data is central to modern business strategies, opportunities for data-driven business decisions and external regulations require businesses to ensure good quality of their core data. The responsibility for and awareness of data quality should be shared throughout a business and shouldn't require, or be reserved for expert teams on this subject.

In this thesis, an extensible and source-agnostic framework is proposed to do data cleansing on structured data. It is purposed for (non-technical) users of the data, after an initial configuration by a domain expert. For sourcing data, the framework provides distributed storage of CSV-files or accepts API-provided (OpenML) structured data. Imported data can be classified and cleansed through workflows ordered in distinct definitions and approaches to data quality. By default, the definitions of DAMA UK and The ABC of Data are configured.

Workflows are structed as directed acyclic graphs consisting of templated nodes. A graph translates to a stack of IPython cells for execution. Each node is assigned a template (Jinja2) that is either hard-coded or dynamically loaded into the database of the framework. Processed data and the generated code is visible in the editor at each step. Workflows can be downloaded as Python notebooks for debugging or further expert analysis for application in the business.

The framework is deployable as a monolith or as federated microservices through a service bus. For analysis, the framework was deployed for two case studies on federal salary data and MycoDiversity DB sample data were executed in multiple workflows. Of the salary data 25% to 32% of entries were classified as dirty, of which all were cleansed. Of the MycoDB data, 30% to 47% of entries were classified as dirty, of which 18% to 88% were cleansed.

Contents

1 Introduction

1.1 Context

An increase in affordable software-as-a-service offerings and systems readily available to collect data on consumer activity trend positive with data collection in businesses of any size [1, pp. 17-18]. Systems that allow businesses to gather, store and process large amounts of data are affordable and indispensable in businesses [2, pp. 4-5].

With more available and more affordable systems, the activity of collecting data and using it to drive business decisions is core to economic activity of many agents in the current markets. However, while it has become easier to collect data, maintaining data quality has proven to be a difficult and complex task to businesses [1, pp. 26-27] [3]. Until recent years, data driven businesses were usually larger enterprises that have staff well-versed in the domain of information technology [1, p. 26] [4, p. 748]. These businesses, other than SMEs, often have staff aware of exposure on privacy and security concerns surrounding data driven activities.

Good practice of data driven activities requires data to be as close to the truth as possible. A task that becomes increasingly difficult when the amount and variety of data grows [5, p. 3]. Uncertain data, and thus low quality data, introduces a barrier for a business to properly use data. This is a burden in compliance with regulatory bodies, and impacts the reliability of activities that drive business decisions based on data [6, p. 80].

Besides opportunity costs, unclean data might cause regulatory issues. Businesses that are uncertain of the data that they store and use might unintentionally store personally identifiable information (PII) without proper discretion or traceability. Under various (European) directives, businesses are required to give customers the right to access, rectify, forget, port and object to the use of their data [7, 8, 9]. Businesses might incur fines if they can not show to be compliant to these requirements or accidentally leak PII. Regarding the financial sector, there is an increasing need to "know your customer" due to legislation from authorities such as FINRA [10, p. 9] and the ECB [11, p. 2]. Certainty of data is essential to a business's ability to adhere to Anti-Money Laundering requirements from such authorities [12, pp. 5-6].

1.2 Problem Statement

It is difficult to keep the quality of data high due to increasing speed, size and complexity of collected data. Businesses can try to keep the influx of data as clean as possible, but it is unavoidable that unclean data will make its way into all the collected data [13, p. 116].

Data cleansing is an activity that identifies and addresses low quality in raw data [14]. This activity should not be, but is often reserved for expert staff [13, p. 122]. Limiting the awareness of data quality and the efforts of data cleansing to central units introduces a concentration in power and separation of knowledge. Insights and patterns from cleansing the data are difficult to communicate to other activities in a business and are unlikely to be learned from. Yet, data scientists must rely on (undocumented) information and insights from others to clean data.

In this thesis a framework is proposed for a system that can be incorporated in a company to make data cleansing easier to execute and communicate.

1.3 Research Objectives

The main objective for this thesis is to design an end-user friendly system that can be used to intuitively profile quality characteristics of a dataset and cleanse the dataset within the same environment.

This can be further split up, resulting in the following research objectives:

1. Design an extensible, turn-key data cleansing solution based on established data quality and data governance research.
2. Design a solution that is accessible to data experts as well as business specialists and data scientists; In order to achieve this, the design and implementation is based on low-code development principles.
3. Design a solution that can be deployed with little costs, allows easy development and scales to the required context; In order to achieve this, the deployment is based on established deployment strategies.
4. Design a solution that is easily extensible by other developers and allows plug-in development for specialized deployments in businesses.

1.4 Research Methodology

The problem statement and research objectives outline a practical problem. For practical problems, research can be done by designing and implementing a solution, commonly known as research by design or design-based research [15, p. 4].

First, the problem is investigated by an extensive literature study on both deployment infrastructures and data quality. This provides the basis for the solution that is then designed and implemented. Finally, the solution is validated through two datasets to be cleansed as case studies:

- Employee Salaries, Montgomery County, MD
- MycoDiversity DB, Martorelli

The final products are an implemented solution or artifact (the framework) and the theoretical basis for the artifact (this thesis). The source code of the artifact is published as open source.

1.5 Overview

Chapter 2 documents research and definitions of existing concepts on data cleansing, software architecture and the implementation of scalable and extensible software systems. Chapter 3 describes the logical and technical architecture and implementation of the solution proposed in this thesis. It maps an architectural overview of the solution and its interfaces with the built solution and provides a blueprint for future work. Chapter 4 displays elements of the designed user interface and its implementation. Chapter 5 validates the solution by two case studies that cleanse a dataset on multiple dimensions. Chapter 6 discusses the results from the case studies and the designed solution. Chapter 7 concludes by reflecting on the research objectives and describes shortcomings of the implementation, to be used as a baseline for future work.

2 Background and Related Work

This chapter covers established literature and approaches to the problems described in the problem statement. In the first section, the definition and context of data is defined. In the second section, definitions of data quality are documented. In the third section, patterns for software architecture and deployment are documented. Furthermore, standards for inter-process and inter-application communication are documented.

2.1 Data

Data can be described as a collection of signals, structured or unstructured. In this thesis data is defined as structured data according to the relational model [16, p. 379]. Values (data points) are collected in relations (tables), meaning data is stored in n-tuples (rows) of m attributes (columns). Examples of data with this definition are data collected in spreadsheets, SQL-databases [17] or DataFrames [18]. Each row of data is considered a record, each column of a record is considered an attribute value. A schematic overview of this data structure is shown in Figure 1.

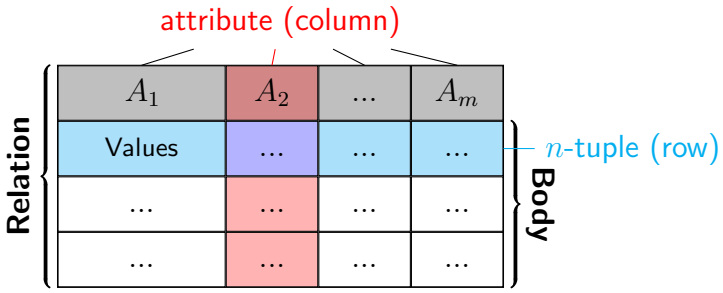

Figure 1: A schematic view of the relational model using one relation (table)

2.2 Data Quality

Data itself is merely a collection of signals that can seem random and sparse at first sight. A hierarchical model that can be used as a guide to leverage collected data is the DIKW-pyramid model [19, pp. 166-168]. This separates the usage of data into two reactionary activities (information and knowledge) and one anticipatory activity (wisdom).

Starting out with raw data, the activity of information described and labels the data using statistical methods and algorithms. Then, the activity of knowledge interprets the information and it's structure, either automatically or manually. This gives an explanation of why the raw data is structured the way it is. Finally, the activity of wisdom is to collect and interpret the knowledge. This allows for anticipatory decisions by agents based on the initial collection of raw data. Future data can then traverse the same path of activities and the decisions can be evaluated for their effect.

Considering this hierarchy, it is required that the quality of the data is high. Information, knowledge and wisdom are subsets of the data and substandard data will inadvertently lead to substandard results. However, the quality of data in itself is a rather broad and abstract term. For example, imagine a table of cashier transactions with the following issues at time of collection:

- Bought products are registered, quantities are not
- Due to a software bug, only self-scan transactions were logged for a week
- A brand of toilet paper has switched SKU after a change in distributor
- Some employees manually enter composite codes for products to save time
- A customer shares their loyalty card with their colleagues

The first two issues cause some form of incompleteness, the third and fourth issues cause a form of inconsistency, and the last issue causes a form of inaccuracy. These issues are distinct and can not be solved using the same interventions. Even the issues that share the same form need a different kind of intervention.

In the following paragraphs, distinct definitions of data quality are documented. Literature research shows that data quality is generally approached from four distinct types (or, one could say, schools) of definitions. First, schema normalization, a definition of quality directly following the applicability of the relational model of data. Second, definitions viewing quality as a combination of distinct measurable dimensions. Third, definitions that view quality from a perspective of levels that can be progressed. Fourth, definitions that view quality from the perspective of ability to execute activities on the data.

2.2.1 Schema Normalization

The relational model approaches data quality from the approach of predicate calculus. One of the objectives of the relational model was to provide an interface to structured data built with predicate calculus (first-order logic) [16, p. 381]. For data in the relational model to be queried and edited while adhering to the rules of predicate calculus the data or data model must comply to a set of requirements. In the relational model, these sets of requirements are defined as normal forms.

Each normal form is numbered[1] and successive to the normal forms with a lower number. They are generally abbreviated as the order of the form:

Name	Locations
LIACS	Snellius, Huygens
PolSci	Pieter de la Court, Wijnhaven

Normalization to 1NF

Name	Location
LIACS	Snellius
LIACS	Huygens
PolSci	Pieter de la Court
PolSci	Wijnhaven

Normalization to 2NF

Location	FID
Snellius	1
Huygens	1
Pieter de la Court	2
Wijnhaven	2

FID	Name
1	LIACS
2	PolSci

Figure 2: Example of the normalization of a decomposable attribute to 1NF to 2NF

[1] The Boyce-Codd Normal Form (BCNF) is not numbered, but is referred to as both BCNF and 3.5NF.

no normal form (0NF), first normal form (1NF), second normal forms (2NF), et cetera. The first normal form (1NF) requires that all domains in a relation are simple. Meaning that all values in a table are atomic, they can not be decomposed further. In practice, this means that tables can't contain other tables or lists of values as values. Normalization is the practice of making data compliant with normal forms. An example of normalization of data from no normal form (0NF) to the first normal form (1NF) and successively the second normal form (2NF) is given in Figure 2.

Further normal forms test the data for functional and joining dependencies. These normal forms eliminate side effects that occur from modifying data in tables and adding attributes to the tables. Essentially, these normal forms try to eliminate as much (hidden) dependencies across values in the same table, so a change to the data in one value can not create an unexpected situation elsewhere [20]. In practice, having such dependencies results in redundantly storing data in a database, which can lead to several problems, among which but not limited to [21, p. 607]:

- **Redundant Storage** - Repeatedly storing the same data.
- **Update Anomaly** - If one copy of repeated data is updated, an inconsistency is created unless all copies are similarly updated.
- **Insertion Anomaly** - It may not be possible to store certain information unless some other, unrelated, information is stored as well.
- **Deletion Anomaly** - It may not be possible to delete certain information without losing some other, unrelated, information as well.

The latter two anomalies usually only occur if empty values (or null-values) are not allowed in the database.

2.2.2 Dimensions of Data Quality

Another approach to define data quality is to try to measure data along a set of dimensions. Similar to what length and width are to area. A collection of such definitions is curated by the Data Management Association [DAMA], a non-profit organization that publishes research on "data governance" and offers certification for professionals who specialize in governing data. They publish a set of standard terminology and guidelines (a body of knowledge, BOK) named DMBOK (Data Management BOK) [22]. In the DMBOK (2nd ed.) a section is dedicated to "Data Quality". In a subsection of the DMBOK regarding "Dimensions of Data Quality"[2] DAMA writes about four frameworks for data quality [22, pp. 454-459]:

Strong-Wang [23] First, the framework that Strong & Wang propose views data through the perspective of a data consumer and accordingly defines shortcomings that are applicable to this consumer. Quality can then be measured through dimensions that create distance between what the customer needs and what it currently is.

They explicitly divide dimensions of data quality into four distinct categories: intrinsic, contextual, representational and accessible. The categories with underlying dimensions are shown in Table 1.

[2] *"The term dimension is used to make the connection to dimensions in the measurement of physical objects (e.g., length, width, height)."* – [22, p. 454]

Table 1: Beyond Accuracy: What Data Quality Means to Data Consumers [23]

Intrinsic DQ	Contextual DQ	Representational DQ	Accessibility DQ
Accuracy	Value-added	Interpretability	Accessibility
Objectivity	Relevancy	Ease of understanding	Access security
Believability	Timeliness	Representational consistency	
Reputation	Completeness	Concise representation	
	Appropriate amount of data		

Redman [24] Second, the framework that Redman proposes views quality on how the data should be constructed according to the model, and how well this construction performs. They don't use the relational definition of data but instead define it as a "representable triple": $(entity,\ attribute,\ value)$, meaning each data *value* belongs to an *attribute* (say, it's domain), which in turn belongs to an *entity*.

This abstraction tries to explain that data is constructed through a model, an entity that has attributes with assigned values. Vice versa, the model is constructed by values, as a set of values belong to an attribute which in turn belongs to an entity. Finally, Redman adds that a set of rules surrounding the recording of data are also a source of data quality (representation). The dimensions are listed in Table 2.

Table 2: Data Quality for the Information Age [24]

Data Model	
Content	Relevance of data, the ability to obtain values, clarity of definitions.
Level of Detail	Attribute granularity, precision of attribute domains.
Composition	Naturalness, identifiability, homogeneity, minimum necessary redundancy.
Consistency	Semantic consistency of the components of the model, structure consistency of attributes across entity types.
Reaction to Change	Robustness flexibility.
Data Values	
Accuracy, Completeness, Currency & Consistency	
Representation	
Appropriateness, Interpretability, Portability, Format precision, Format flexibility, Ability to represent null values, Efficient use of storage, Physical instances of data being in accord with their formats	

English [25] Third, English views dimensions of data quality from a consequential point of view. They split data quality into inherent and pragmatic characteristics. Inherent characteristics are independent to usage of the data while pragmatic characteristics can change when usage of the data changes. The dimensions are listed in Table 3.

DAMA UK [26] Finally, the DMBOK concludes with a white paper written by DAMA itself. In *The six primary dimensions for data quality assessment* DAMA publishes the results of a working group tasked to find best practices on data quality. This results in the six dimensions listed in Table 4.

Table 3: Improving Data Warehouse and Business Information Quality [25]

Inherent quality characteristics	Pragmatic quality characteristics
Definitional conformance	Accessibility
Completeness of values	Timeliness
Validity or business rule conformance	Contextual clarity
Accuracy to a surrogate source	Usability
Accuracy to reality	Derivation integrity
Precision	Rightness or fact completeness
Non-duplication	
Equivalence of redundant or distributed data	
Concurrency of redundant or distributed data	

Table 4: The six primary dimensions for data quality assessment [26]

Completeness	The proportion of stored data against '= the potential of 100% complete'
Uniqueness	Nothing will be recorded more than once based upon how that thing is identified.
Timeliness	The degree to which data represent reality from the required point in time.
Validity	Data are valid if it conforms to the syntax (format, type, range) of its definition.
Accuracy	The degree to which data correctly describes the "real world" object or event being described.
Consistency	The absence of difference, when comparing two or more representations of a thing against a definition.

Concluding the frameworks mentioned in the DMBOK, the authors note that none of these frameworks is perfect in all situations. Given any dataset, one framework may be fit better than others.

2.2.3 Data Readiness Levels

All the frameworks highlighted by DAMA focus on the definition of quality in different dimensions, but provide no relation or importance of dimensions respectively to the others. Similarly, normal forms are either true or not, while some functional dependencies might be more crucial to avoid than others. The following two frameworks propose an approach where data quality is defined through different levels of readiness. Different from aforementioned methods, these frameworks are opinionated on the importance of various data quality metrics.

Lawrence [27] First, in *Data Readiness Levels* [27] Lawrence proposes a set of bands that rate the fitness of the data. They split the data into three bands: A, B & C, with decreasing fitness. The bands are in turn split up into sublevels, so that A1 is the best state overall and C4 is the worst state overall.

- `Band C` – **Accessibility** – Is it being stored in the format that we need it in? Is it stored on a difficult to access distributed system? Are there privacy or legal constraints on the data?
- `Band B` – **Faithfulness and representation** – Is there noise or are there data errors in the data? Is the data that we expect there? How are missing values dealt with?

- `Band A` – **Data in Context** – Do we need more data to answer the questions that we have? Does the current data need annotation by humans? Can this data be used to reach the goal?

Lawrence then continues with examples of cases where the data readiness levels would've helped with better expectation management and steering of projects. They conclude that these projects would've benefited from the awareness of data readiness levels.

While the data readiness levels present a useful tool for communication and planning, it is difficult to implement the data readiness levels as presented into a consistent software solution. The descriptions of the levels are rather abstract and differ greatly per dataset.

Castelijns, Maas & Vanschoren [28]

Continuing from Lawrence, in `The ABC of Data` more detailed and precise definitions for data readiness levels are given. Castelijns et al. define weighted deficiencies in each band and add two extra bands that improve on the A-band.

- `Band C` – Conceive
- `Band B` – Believe
- `Band A` – Analyze
- `Band AA` – Allow analysis
- `Band AAA` – A clean dataset

Each band is defined by a set of underlying deficiencies with weights. If a dataset has none of the deficiencies, it can progress to the next band. The deficiencies, weights and bands are shown in Table 5.

Table 5: The ABC of Data [28]

Band	Weight	Deficiency
C	40	Parseability
	25	Data storage
	15	Decoding
	10	Data Formats
	10	Disjoint Datasets
B	20	Column Types
	30	Missing Values
	20	Inconsistent Data Entries
	10	Duplicated Records
	20	Meaningful Values
A	20	Interpretable Values
	20	Feature Scaling
	20	Outlier Detection
	30	Feature Selection
	10	Coverage Gap Detection
AA	40	Legal Violations
	40	Security Risks
	20	Bias Detection
AAA		None

2.2.4 Executionability [29]

Another perspective to approach data quality is to assess how well a business can execute their core business with the data at hand. In the ISO 9000:2015 and 9001:2015 quality management principles (QMPs) are proposed and standardized for enterprises. The standard states that [30, p. 1]:

> "The relative importance of each principle will vary from organization to organization and can be expected to change over time."

The notion that quality management principles are of relative importance to organizations is relevant for the management of data quality. To assess the importance of data quality and methods to improve data quality, requirements must first be defined. When these requirements are defined as executionable code the code can be run and according to how well the code runs, the required quality can be assessed.

In Bicevskis et al. [29] the authors propose a model of data quality where it is defined and measured using executionable language. In their paper they use SQL as a DSL [31, p. 13], but the idea can be

implemented in other languages or systems as well.

They split executionable requirements into three categories:

- **Syntactic Control** - Checks that can be executed locally within the same record (e.g. are all required attribute values present).
- **Contextual Control on interrelated data** - Checks that validate if the data is compatible with related data entries (e.g. do project items start after the start of the superseding project)
- **Contextual Control on the entire dataset** - Checks that validate if all values or the aggregate of a dataset is valid (e.g. are all values in a column unique)

2.3 Design Considerations

In the previous section, a multitude of methods to approach data quality are mentioned. The list of documented methods is not exhaustive. There are more approaches to this problem and in the future other definitions for problems will be considered and need to be tried.

There are various strategies for software and deployment architecture, and some strategies allow for easier development, scalability or extensibility. In this section different strategies and their main applications are documented.

2.3.1 Software Architecture

Software developers spend most of their time on the job on figuring out how a system works before they can write code for it [32]. If an application has a well-documented and fairly standard architecture, it is assumed that the threshold for re-use and contribution is kept low.

For this thesis, the programming language was predetermined to be Python 3. Taking into account that the programming language is a given, there are three main risks when reviewing and choosing architectures. For starters, architectures are generally reviewed based on how well they support reuse of components [33] and the application of broadly-used software patterns [34]. However, it might be the case that the components and patterns important for the reviewer are not relevant for the implementation of the solution in this thesis. Second, the availability of established literature might be biased towards the authors' work, opinion or customs. For illustration, businesses create value out of developing and propagating a certain architecture. A business that sells middleware will likely steer towards the adaptation of a service-bus oriented architecture so that their software will work with other systems that are essential to their customers. Finally, architectural patterns might describe the abstract solution in a sensible way, but computer systems don't always map well to theoretical frameworks. Something might make sense in an abstract presentation, but would require a lot of extra (boilerplate) code in the language used by the developer. The culture of developers that prefer a certain programming language might also prefer certain architectures, making it difficult to find and incentivize developers.

Architectural patterns that see widespread use in software development and design are Monolithic architecture, service-oriented architecture (SOA), microservices (M/SOA), and n-tier architecture, these are reviewed in the following paragraphs.

Monolithic Architecture First, a monolithic architecture describes the architecture of software that is encapsulated as self-contained and independent of other software and systems.

All the parts of a monolithic application are in code reachable across the entire application and share the same codebase and platform. Monolithic architectures work well in the context of a small codebase and a small team of developers that work on the application [35]. It is easy to set up a development environment and to deploy the application to a production environment, as there is only one application that needs to be deployed and monitored.

The biggest caveats of monolithic applications however are that they don't scale well and incentivize coupling. Scaling is difficult due to the atomicity of the application. If one part of the program requires more resources, the entire deployment needs to be scaled up. If the application needs to be highly available, it needs a complete copy running elsewhere, instead of just the important parts. Coupling is incentivized as developers work in a singular codebase, which encourages (invisible) reuse of code that is used elsewhere in the application for an entirely different purpose. As the application grows, parts that seem unrelated can become functionally dependent on each other. This makes Quality Assurance and Development difficult as code changes might reflect differently than can be expected.

In recent years, the monolithic approach has seen an increase in use again after being abandoned, in both small and large enterprises [36, 37]. In recent adoptions of the monolith, its internals are approached as distributed and domain-driven components. While the repository is shared, there can be an internal separation of concerns. This way, unintended coupling can be prevented, and parts of the application can be scaled up and down [38]. A schematic example of a monolithic architecture and distributed monolith is shown in Figure 3.

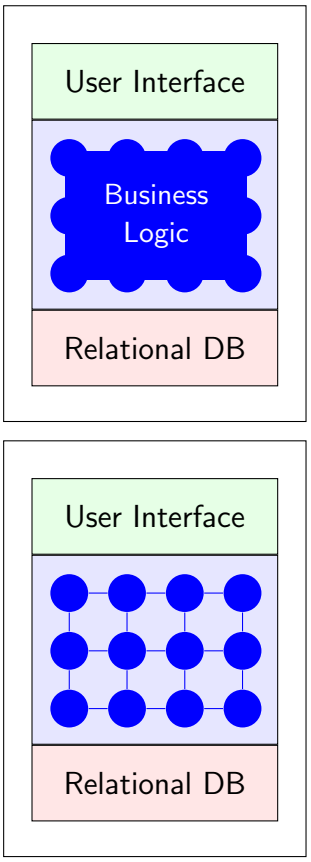

Figure 3: Example of a monolithic and distributed monolithic application.

Service-Oriented Architecture (SOA) Service oriented architecture is a software architecture that envisions applications as a collection of separate services. There are many definitions and reference models for the service-oriented architecture, but the de facto standards are written by the Object Management Group, Open Group and OASIS [39].

According to the Open Group: "A service is a logical representation of a repeatable activity that has a specified outcome. It is self-contained and is a 'black box' to its consumers" [40, p. 20]. Service-Oriented Architecture allows businesses to take a more holistic approach to development, focusing on a macro-level (services) instead of on a micro-level (classes). To offer an application to customers services are often exposed through a service bus or Enterprise Service Bus (ESB).

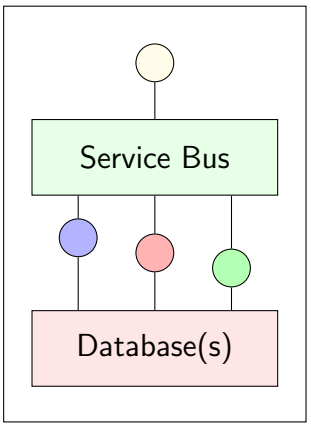

Figure 4: Example of a service oriented architecture

One of the main challenges is that two services might need to share their state, which introduces coupling between the two services that should be independent and unaware of each other. This diminishes the benefits such as scalability, granularity and oversight. A schematic example of a Service-Oriented Architecture is shown in Figure 4.

Microservices (M/SOA) and Serverless Microservices [35] & serverless architectures [41] are similar to service-oriented architecture. However, in contrast to SOA in microservices the services are approached more like building blocks in pipelines that don't necessarily encapsulate business logic themselves. Instead of using a service bus that exposes a single service to a customer, there is no overarching orchestrator. Rather, each service can be independently queried and mutated.

Serverless architecture is a subset of M/SOA, herein the deployment of the microservices is considered not part of the architecture. Instead, the architecture uses off-the-shelf offerings (such as AWS Lambda) that execute a "function".

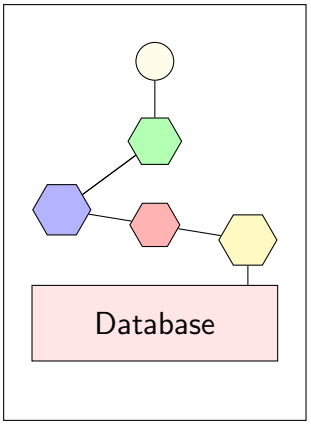

Figure 5: Example of an application using microservices

One of the main issues of microservices architectures is that the complexity of running and building the application is moves into the operational tasks of an organization [42]. All microservices have specific inputs, outputs and dependencies, keeping those up-to-date and being aware of the changes elsewhere in the organization and their effect on the service your team is maintaining is a complex task. A schematic example of an M/SOA architecture is shown in Figure 5.

Multi-tier (n-tier) architecture Finally, an often used architecture in web development is the n-tier architecture. This views software architecture as consisting of multiple layers that build on top of the underlying layers [43, p. 31].

In practice this often manifests as a three-tiered model: (1) presentation tier, (2) logic tier and (3) data tier. In this model, the presentation tier formats the data it receives from the logic tier into a view for the end-user. The logic tier decides what should be requested from the data tier and what should be presented. The data tier handles connections with the database and retrieves the information that is requested from it. A schematic example of an n-tier architecture as 3-tier architecture is shown in Figure 6.

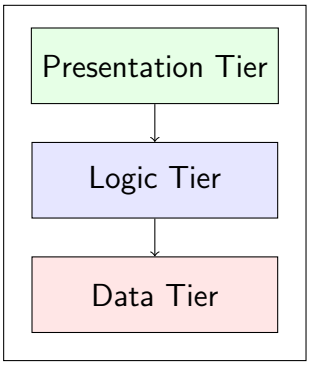

Figure 6: Example of a 3-tier architecture

2.3.2 Programmable Interfaces

For the solution, an application programmable interface (API) is built for communication between processes and/or separated services. There are various paradigms and standards on the writing of programmable interfaces. In this section, a selection of these are described and considerations relevant for choosing a standard is given. Representational State Transfer (REST), gRPC and GraphQL were applied while designing the solution and are discussed in order.

Representational State Transfer (REST)

Representational State Transfer (REST) is the most widely used API-standard used for web applications [44]. It is based on a proposed set of constraints from the REST-framework, which doesn't precisely prescribe how an API can be built but rather provides a design pattern [34] that can be implemented in different contexts [45, pp. 76-86].

When the standard HTTP call methods are used (GET, POST, UPDATE, PUT, DELETE) the constraints from REST stand. When the client is provided a base URI, a list of endpoints and the required media type the requirements for REST are met. APIs that provide this are generally called RESTful APIs.

```
urlpatterns = [
    path('faculty/<str:name>/buildings', views.buildings),
]
```

↓ provides an interface ↓

```
curl --json http://${baseURL}/faculty/${faculty}/buildings
```

↓ returns the object ↓

```
{
    "id": 1,
    "name": "HUYGENS",
    "code": "str('HUYGENS')"
}
```

Figure 7: Example of a RESTful query against Django RESTful framework

There are various libraries available for popular programming languages that implement both servers and clients that are compatible with the RESTful framework. An example of such a framework and its result is shown in Figure 7.

Remote Procedure Calls (gRPC)

Different than REST, gRPC does not try to provide an architectural framework but rather provides a set of tools for applications to communicate using a gRPC-client to a gRPC-server [46]. It is based on the architecture of remote procedure calls [47], meant to support inter-process communication. It is not intended for the communication between a web client and servers. Rather, a locally available process (stub) converts a request to the correct and specific predefined format and calls to the server.

```
service FacultyGuide {
  rpc GetBuildings(Faculty)
  returns (Feature) {}
}

message Building {
  required string name = 1;
}

message Faculty {
  required string name = 1;
}
```

Figure 8: Example of a gRPC definition

Considering the predefined format and the requirement of a local (specific) stub remote procedure calls are often used in predefined and compiled systems. Due to their predefined nature communication can be easily verified, and as they can be pre-compiled the execution is quicker than dynamic systems such as RESTful and GraphQL. Furthermore, gRPC avoids the overhead of regular HTTP-requests by only transferring the necessary data. It uses protocol buffers, a serialization protocol that converts the messages to binary wire format transmitted

over HTTP/2. A client and server need predefined descriptions of the remote procedure calls to serialize and deserialize the data.

Data Querying & Manipulation Language (GraphQL)

Finally, GraphQL proposes a structured language for the querying and manipulation of data. It focuses on defining an API that can be queried over HTTP and Websockets for compatibility with web-clients, while enabling the client to send and receive just the necessary information [48].

In implementations of GraphQL the server provides a schema with scalars, queries and mutations that the client can use to build a request. The client then provides a tree with the information to be resolved, which is returned by the server with the same structure. A GraphQL API has only one endpoint, and the task given to the service is dependent on what is given in the body of the request to that endpoint.

```
query {
  buildings(faculty: "FWN") {
    id, name
  }
}
          ↓ returns ↓
{
  "data": {
    "buildings": [
      {
          "id": 1,
          "name": "HUYGENS",
      },
    ]
  }
}
```

Figure 9: Example of a GraphQL query

GraphQL was originally developed for internal use at Facebook. After open sourcing its development was moved to the GraphQL Foundation which is part of the Linux Foundation. There are various implementations available for popular programming languages, for Python the most widely used implementations are graphene (code-first, generating a schema) [49] and ariadne (schema-first, providing hooks to a schema) [50].

3 System Design

The methodology of this thesis is based on the principle of research by design. In the previous section established work on data quality, software architecture and programmable interfaces relevant to the design of the solution are documented. In this section the work is reviewed and considered for to decide and implement a final design.

First, the design is explained on a logical level. How are frameworks for data quality used, what architecture is implemented for the system and how do services in the system communicate with each other. Second, the resulting system is documented service-by-service and the development and deployment strategies for the system are documented.

3.1 Logical Design

In the introduction four research objectives were introduced. These research objectives pave the way in the choices made for the final system design. First, the objectives are repeated and translated to motivations of choices based on the established work documented in the Section 2. Then, the resulting system architecture is presented.

Extensible & turn-key data cleansing First of four, the system does not require extensive configuration, but specified configuration can be done if needed. This is relevant for three components of the design. The principle of how data quality is measured should be given in an opinionated manner, but other definitions of data quality must be supported. Also, common data should be loadable into the system, but other data sources can be added. Finally, the application must be deployable without extensive software engineering or development experience.

For the definition of data quality, the system allows a hierarchy of data quality definitions as Dimension Sets which are a list of Dimensions. Out of the box, the definitions from DAMA UK 2.2.2 and The ABC of Data 2.2.3 are provided:

- DAMA UK (2013)
  - **Dimensions of Data Quality** `<DimensionSet: Dimension[]>`
    Completeness, Uniqueness, Timeliness, Validity, Accuracy, Consistency,
- Castelijns et al. (2020)
  - **Band C** `<DimensionSet: Dimension[]>`
    Interpretable Values, Feature Scaling, Outlier Detection, Feature Selection, Coverage Gap Detection,
  - **Band B** `<DimensionSet: Dimension[]>`
    Column Types, Missing Values, Inconsistent Data Entries, Duplicated Records, Meaningful Values,
  - **Band A** `<DimensionSet: Dimension[]>`
    Parseability, Data storage, Decoding, Data Formats, Disjoint Datasets,

These dimensions are chosen as the set of DAMA dimensions were selected by a working group of experts and should cover most cases of problems with data quality. After the DAMA dimensions are progressed and are not helpful anymore, the user can the ABC of Data approach for more fine grained and relevant control over quality. By providing the definitions of data quality as a `DimensionSet` consisting of `Dimensions`, other sets can be introduced and existing dimensions can be reused over multiple sets.

For the portability and support of multiple data formats, the data must be first serialised to a format that is usable for the rest of the system. To achieve this, data is loaded into a separate service that provides an abstraction of the data to the rest of the system. This service can then be extended or replaced to serialise data of different formats.

For easy deployment, a daemon or supervisor is provided to orchestrate all the required services.

A schematic overview of these three motivations as a system design is given in Figure 10:

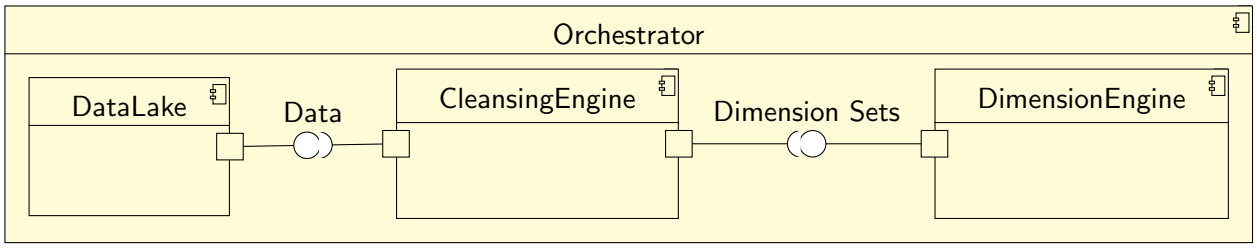

Figure 10: Architecture implications of the first objective

Low-code development principles Second of four, low-code is allowed by representing the classification and cleansing activities in drag-and-drop workflows. These workflows consist of cells that import, process or export data. Each cell is based off a template, which is dynamically loaded at runtime. A schematic overview of this is given in Figure 11

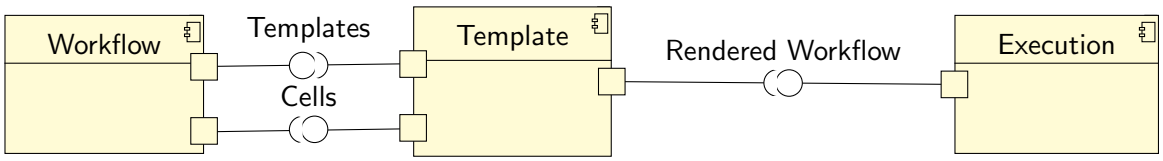

Figure 11: Architecture implications of the second objective

Easy development & scaling Third of four, for good scaling, the system needs to be split up in multiple distinct services as some concerns require a different resource profile than others. However, splitting up a system into multiple services complicates development on the system as a whole.

First, a system of multiple microservices was devised and set up. Herein, the APIs of the microservices were defined schema-first to ensure that interfaces were always well documented, and all code is traceable to an endpoint. This is displayed in Figure 12.

However, orchestration of such microservices in different development environments turned out to be a difficult and error-prone task. Furthermore, for a production deployment, a customer would need a

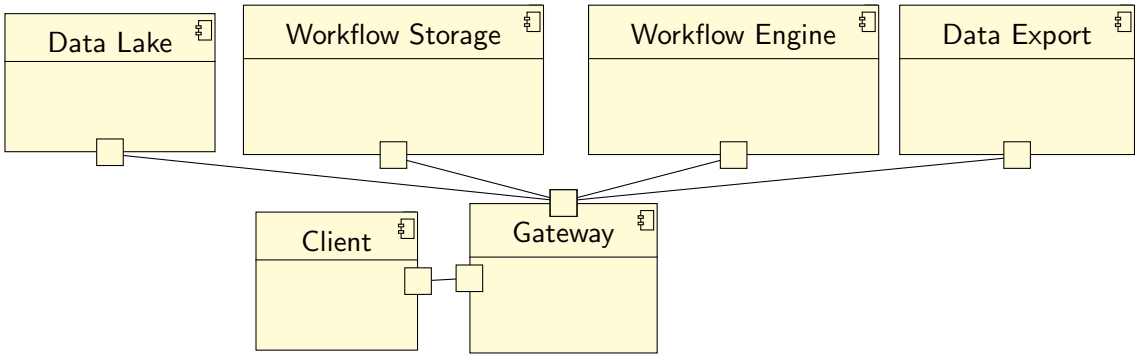

Figure 12: Architecture implications of the third objective: Microservices

container orchestration suite such as Kubernetes. The system was re-written to run as a code-first distributed monolith, which is packaged as shown in Figure 13.

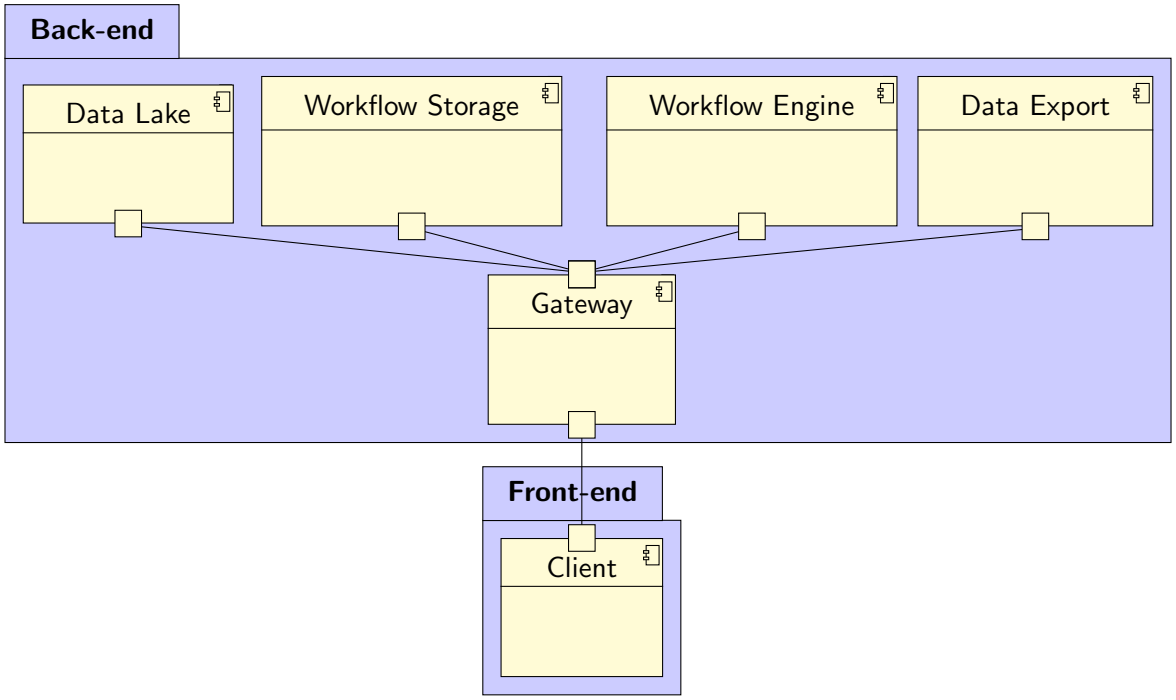

Figure 13: Architecture implications of the third objective: Distributed Monolith

Allows plug-in development Last of four, the templates that are used for cell rendering are loaded dynamically. This enables developers to create specific cells and plug-ins that are relevant for certain users. Furthermore, due to the separation of concerns of services, services can be either partially replaced or extended.

Combining all mentioned architectural decisions, the resulting logical architecture is displayed in Figure 14 on the next page.

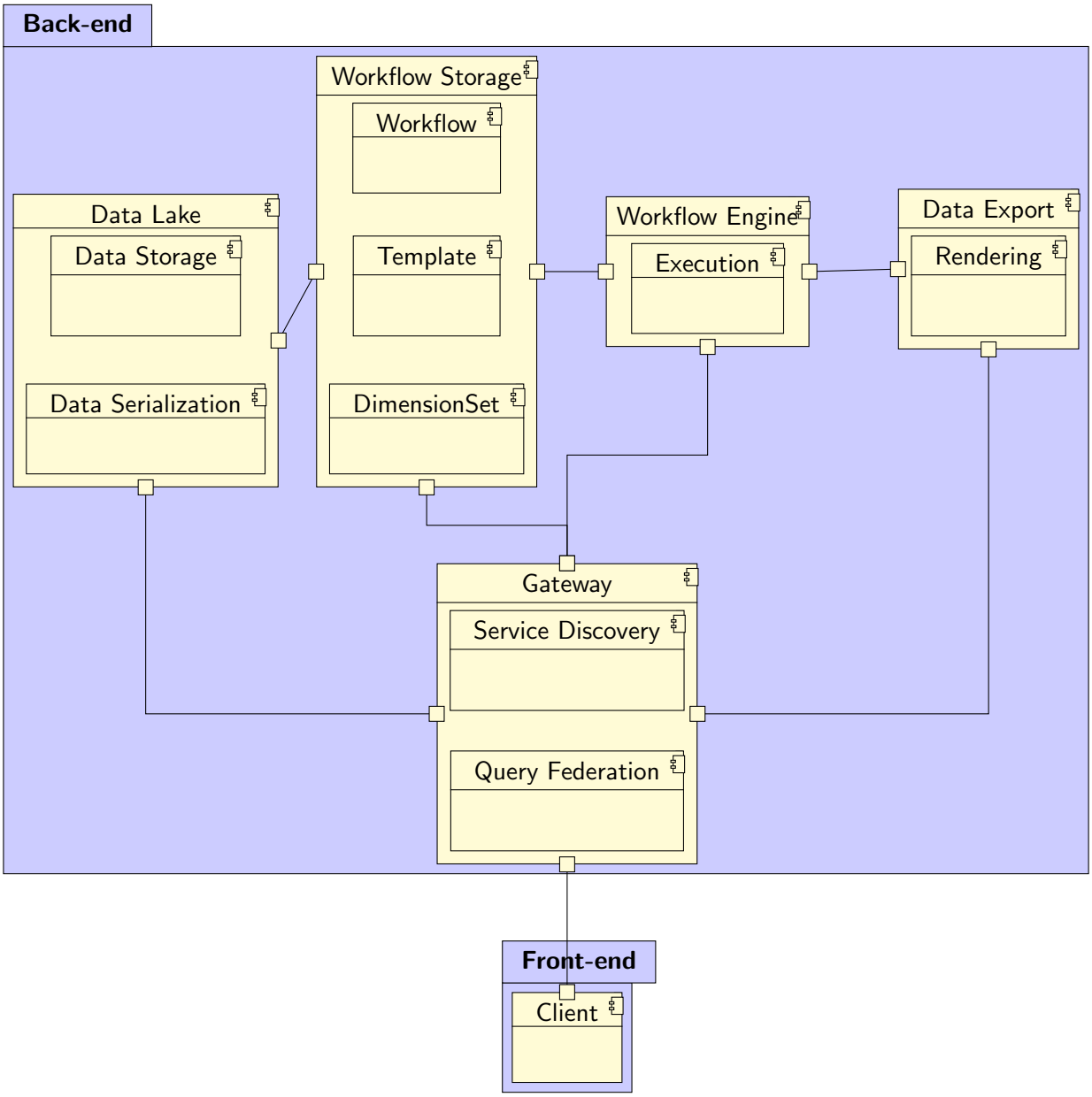

Figure 14: Logical architecture of the system

On a semantic-level the system is modeled after the service oriented architecture (SOA). The Client connects to the Gateway (service bus) which federates requests to underlying services. It is not required for all services to be available at runtime for the system to boot up and run.

However, due to the complexity of development and local deployment of SOA-applications the framework is written as a distributed monolith and packaged as a single application. During runtime, flags can be set so that only part of the provided services by the package start.

3.2 Technical Design

The architecture from Figure 14 translates to the following implemented services:

- **Lens** (Client) is a React [51] application that connects to *Scout* (Gateway) through a GraphQL API.
- **Scout** (Gateway) is the public entry point for the framework. It stores and maintains a register of all the services available to the application. For the end-user this is the only service that needs to be accessible over the network. It federates API-calls and queries to the other services in the system. If the framework is in development mode, it also exposes an IDE for all (internal) GraphQL endpoints.
- **Lake** (Data Lake) is a service that abstracts the data storage to the framework. It connects to an internal or external Dask-server [52] to store the datasets that need cleansing.
- **Flow** (Workflow Storage) is a service that stores the data quality definitions and the respective workflows. It stores workflows (diagrams) used for profiling and cleansing as directed acyclic graphs, and it contains a templating engine that generates python-code from the graphs.
- **Engine** (Workflow Engine) is a service that communicates with the execution engine for the python-code, Jupyter. It generates Jupyter-notebooks [53] from python-code and runs these using the Jupyter API. It stores the results and the runtime status of each diagram.
- **Audit** (Data Export) is a service that converts the results of the *Engine* runs into human-readable results and exports the notebooks generated by the workflow engine.

The framework is written as a Django [54] project that consists of multiple Django applications. The project has a main *ludbf* application that links together the configuration and dynamically loads the configured services at runtime. This gives end-users the ability to either run the entire framework in one deployment or to do multiple deployments with one service configured each. Each Django-application has an internal GraphQL API that can be used for local queries and exposes a GraphQL API that can be used by other services.

3.3 Implementation

Each Django-application has a set of objects registered as models. These models are mapped to a PostgreSQL database by the Django-framework and can be accessed internally by the service. These models are converted to GraphQL-definitions using the `graphene` and `graphene-django` libraries. Custom GraphQL resolvers and decorators are added for federation-purposes and to avoid clients to do complex mapping. Details of how `graphene` and the Django ORM work are out of scope for this thesis.

Lens (React) Lens is written as a TypeScript React application that is built and served by Django using the static files provider. When it is accessed through Django it automatically configures the URL of the *scout* service for communication. The client only communicates with the *scout* service through GraphQL and does not need to access other services in the framework.

It does not store local state, all actions by the user are deferred to *scout* and the metadata in the client is partially refreshed after a change is confirmed by *scout*. Each query is executed using the *React Query*

library, which exposes the queries and functions for queries internally throughout the application. An example of such a query and partial refresh is shown in Figure 15.

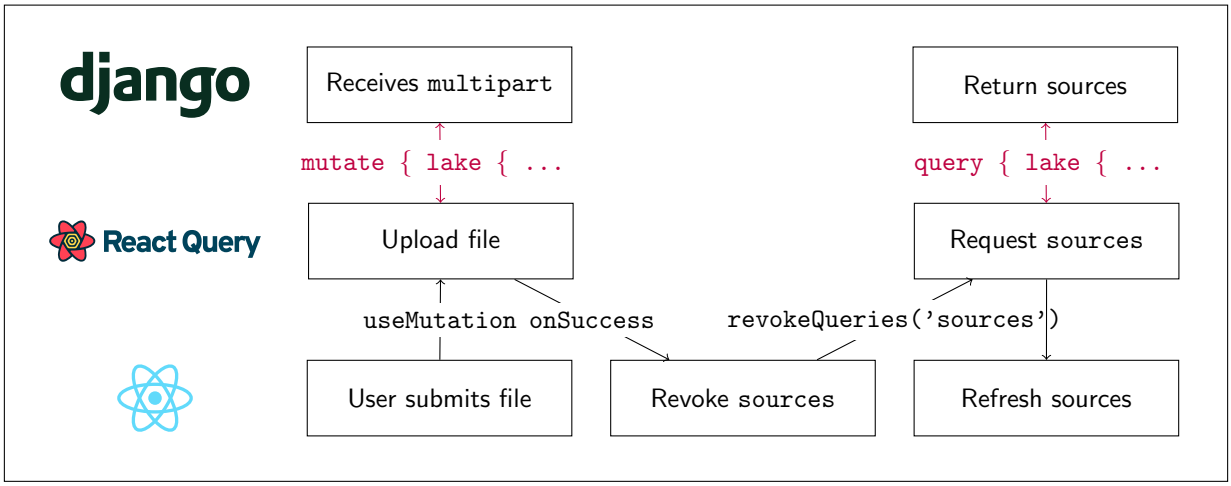

Figure 15: Asynchronous usage of React Query for state management

The type definitions used in TypeScript are auto-generated using the schema given by *scout* and the queries that are defined in the `api/` folder. At build-time the queries and their usage in the React components are checked for null-safety and if used variables match the expected type.

In `lens`, the UI elements that were written are split off in a separate (in-repo) component library. If new fields need to be added, or pages need to be extended in the style of the current UI, the components can be re-used.

Scout (Django) Scout is the service bus for the framework and is implemented as a Django-application. It uses the Django ORM, Graphene and GraphQL libraries to store available services and to disassemble and federate queries. On start-up, it does a lookup in the system settings to see which applications the system should be running locally and adds those to the main database if not already there.

It then assembles type definitions for the services it can federate to by importing the schema definitions of those services. Each service has an immutable key, queries against services (federated or in-thread) can be done by querying the application and passing the service's key. An example of such a query is shown in Figure 16.

```
query FederatedLake {
  lake(host: "19") {
    sources {
      name
    }
  }
}

query InternalLake {
  internal {
    lake {
      sources {
        name
      }
    }
  }
}
```

Figure 16: Example of a federated and internal GraphQL query using *scout*

Lake (Django) Lake is the service that handles the storage and querying of datasets in the framework. It is implemented as a Django-application and uses the Django ORM and Graphene for querying and mutating metadata of datasets in the database. Furthermore, uploads and downloads of datasets are initiated through the

GraphQL API and are placed in a Redis queue implementing a pub-sub pattern. Meaning, each upload and download of a dataset is added to a queue. The progress and result of the operation is stored in the queue as metadata.

When an operation is added to the queue as pending, a separate worker thread picks up the operation and tries to add the resulting file to a Dask instance. If this fails (e.g. due to corruption of the file or improper formatting), it doesn't lock up the main server thread but instead produces an exception that is logged in the queue. The main (server) thread listens for results in the queue and stores them in the database permanently.

Completed operations are stored as dataframes in the Dask instance and the details needed to access the dataframe from the Dask instance are stored in the database for querying by other services. A diagram describing this pattern is given in Figure 17.

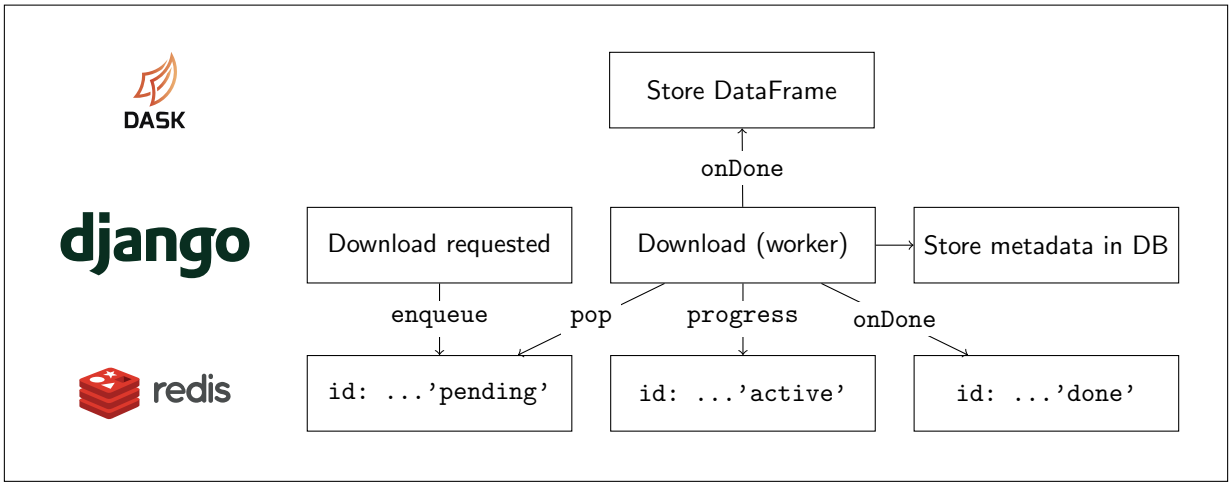

Figure 17: Asynchronous downloads of CSV-files using a pub-sub pattern

Flow (Django) Flow is a workflow engine and template renderer for cleansing diagrams and workflows. It is implemented as a Django-application similar to Scout and Lake.

When a workflow is created and executed for profiling, the user can proceed to the creation of a cleansing workflow that cleans and fixes the data that was previously marked as problematic. A cleansing workflow starts out with the last recorded dataframe from the profiling workflow. Then, the result of the cleansing is measured by running the profiling workflow to determine if the quality of the data has increased.

Each step in a workflow has a user-selectable template. Templates are configured with a set of variables and their respective types. If a variable is marked as *resolvable* the client will query Flow to get a list of possible options to select. Using predefined resolvers, operations on the dataset's schema can be tracked to avoid mistakes and exceptions on execution.

Workflows are stored as directed acyclic graphs (DAGs) in the database. It maps the workflow to steps (nodes) and edges in the Django ORM. In this data model it is possible to create a cycle, this is validated at runtime by tracking if a step is executed twice. Templates can be built with the Jinja2 language, and can either be stored as files in the project or as records in the database. The database structure of Flow is shown in Figure 18.

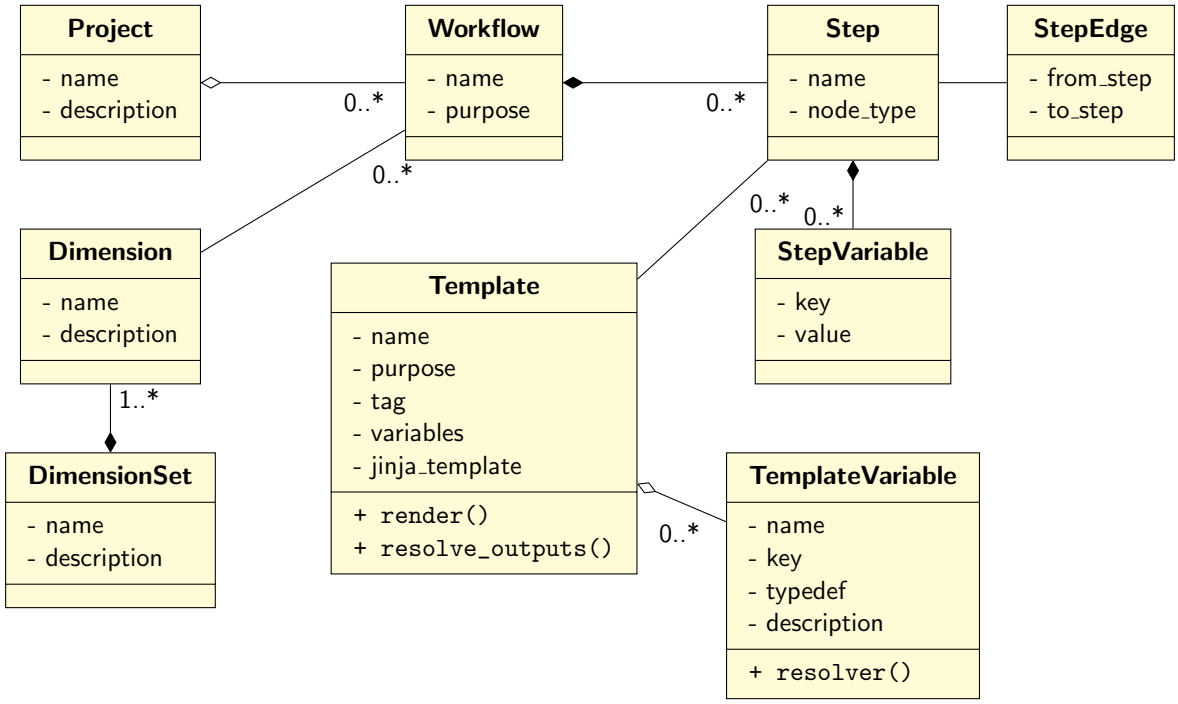

Figure 18: Diagram of the data model used in flow.

Engine (Django) Engine abstracts the executor for the workflow code into a GraphQL API. It is implemented as a Django-application and requires a running Jupyter instance to start properly. Execution is done through a breadth-first search of the workflow DAG. An example of this manipulation is shown in Figure 19.

Engine uses a Redis queue to keep track of the running notebooks and queued tasks. It allows the starting of new runs and cancellation of existing runs through a mutation.

Once a job is added and running, the status of the job can be queried. For each step that was passed to the Engine it keeps track of the current status of the jobs. Furthermore, once a job run is finished, the notebook's contents can be queried through GraphQL to view the outputs of each cell.

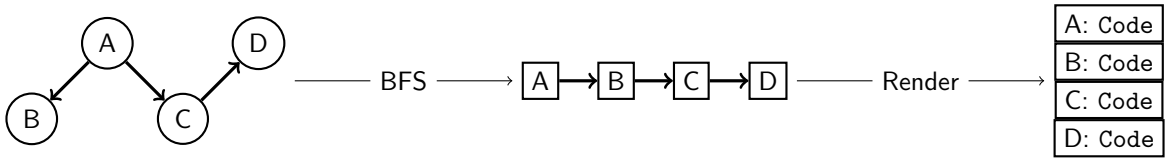

Figure 19: Diagram of rendering a workflow to a Jupyter Notebook.

Audit (Django) Audit can be used to download notebooks from the server. It gathers all the available notebooks from Engines and offers them as downloadable notebooks.

3.4 Development Experience & Deployment Architecture

The framework is structured as a Django project consisting of multiple Django applications. These applications can be run independently or all together, which applications are initialized at runtime by the Django server is configured in the configuration file.

If a developer has a system with an OCI-compatible container runtime (e.g. `podman`, `containerd`, or `docker`) and a CLI that supports the `compose` standard (e.g. `docker` or `nerdctl`) installed on their system it can be started right away with the `docker-compose` file. Upon startup the entrypoint will run all migrations that are necessary to get the application to runtime.

Developers that need to add new functionality to the system can either expand the framework by creating a new Django application in the project or by working in the apps that are already there. The recommended action is to create a new app, and when the changes are stable, the new app can be integrated in the to-be-extended app.

```
.
|-- ludbf
|   |-- audit
|   |-- engine
|   |-- flow
|   |-- lake
|   |-- ludbf
|   |   |-- configuration.example.py
|   |   |-- configuration.py
|   |   +-- settings.py
|   |-- project-static
|   |   |-- ludbf-graphiql
|   |   |-- ludbf-lens
|   |   |-- package.json
|   |   +-- yarn.lock
|   |-- scout
|   +-- static
|-- docker
|   |-- configuration.docker.py
|   |-- entrypoint.sh
|   |-- nginx-unit.json
|   +-- requirements-container.txt
|-- docker-compose.yml
|-- Dockerfile
|-- upgrade.sh
```

Figure 20: Section of the repository tree of the LUDBF project

The framework is set up in such a way that it can easily be deployed as a monolithic framework on a single server, as shown in Figure 21. However, the internal federation and service-bus architecture of `Scout` allows apps to be split up and deployed highly available or over multiple instances for load balancing. For production set-ups that need to be highly available, multiple instances of Scout can be set-up behind a load balancer, as long as they share the same Redis and PostgreSQL-cluster. An example of such a set-up is shown in Figure 22.

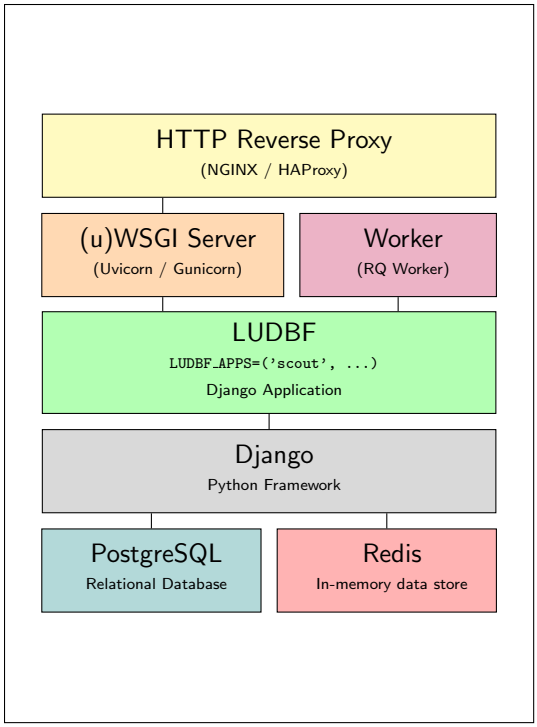

Figure 21: Default monolithic set-up of the framework (default `compose`-setup)

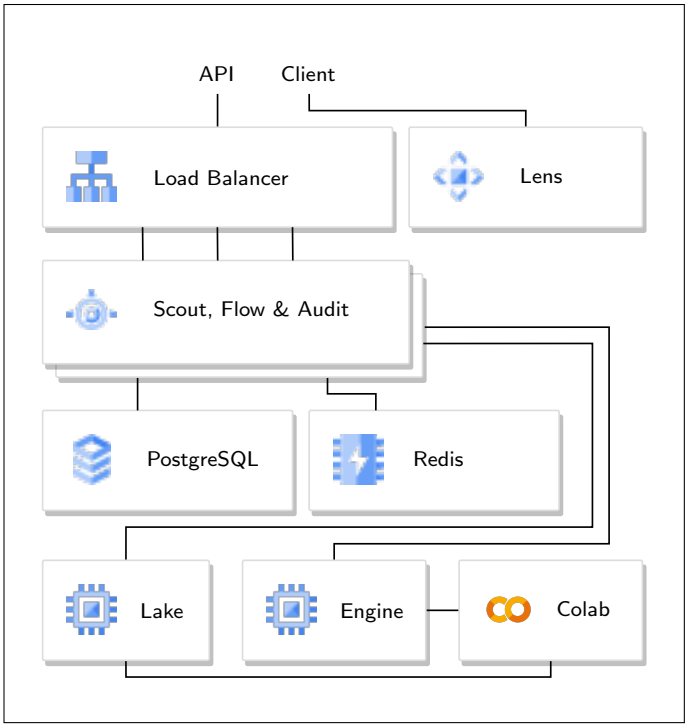

Figure 22: Example of a large production deployment on a cloud platform

4 Interface Design

One of the main objectives of this thesis is to build a framework that can be used for end users to do data cleansing intuitively. Central to an intuitive system is an intuitive user interface. In this chapter the flow that the user takes through the application is described. Highlighting elements and choices in the interface design for good user experience.

This section is limited to `Lens`, as this is the only application that implements a user interface.

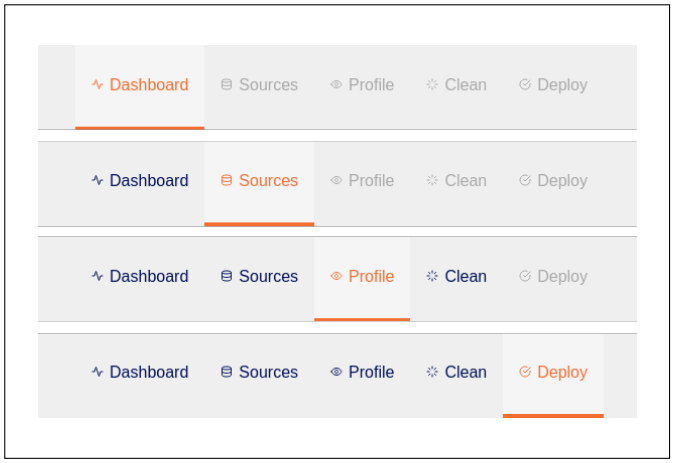

Figure 23: Header navigation dependent on the progress the user has made

4.1 Progressive UI

Progress-based availability

The `Lens` application will only present paths for the user to follow that are available to walk through, as shown in Figure 23. For example, if there are no `Lake` instances or sources available the application will disable the buttons to create and audit workflows.

URL-routing with breadcrumbs

Even though the application is a Single Page Application (SPA) it uses a Router for the page fetching and lazy loading. Throughout the application, each screen has a unique URL with the necessary information to render the screen. At the top of each page a ribbon with breadcrumbs is shown, so users are at all times aware of where in the application they are.

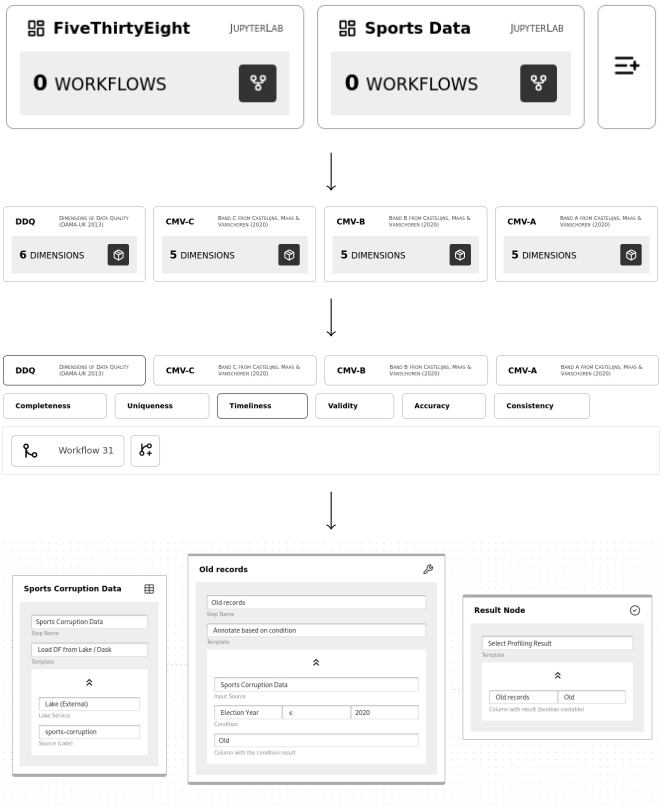

Figure 24: Hierarchical navigation through cleaning dimensions

Hierarchical Navigation

Workflows are categorized under a dimension and project in the application. Users can create different projects for data cleansing and under each project a list of dimension sets and dimensions are given. In Figure 24 the UI is shown for selecting a dimension set and dimension to create or edit a workflow.

4.2 Modularization of UI elements

The source of `Lens` is split out into four main parts: api, application, hooks and `ngUML.components`. The latter being a library with all the reusable components throughout the application. It contains various standard UI-elements ranging from navigation menus to font styles. This helps with keeping boilerplate code around styling in the rest of the application low and with having a consistent UI throughout the application.

Improvements in user experience design can be made in the `ngUML.components` module which will propagate throughout the entire application. Furthermore, the `ngUML.components` module can be used by other Leiden University (or ngUML projects) to keep a consistent user interface over multiple applications and to re-use efforts put into design.

4.3 Workflow Editor

The application uses a third-party library `React Flow` to implement a drag-and-drop grid with each workflow step presented as a node on the grid and each connection between two steps presented as an edge. The diagrams made for profiling and cleansing can be built from scratch in the browser. Their state is written to and read from the server instantaneously, allowing no room for dirty states on the client side.

Steps come in three main categories:

- **Source** - Loading data sources into the workflow. This can either be Lake DataFrames, Factory (Generated) Data or API-backed Data.[3]
- **Flow** - Transforming or evaluating columns, any steps to get the data source to a result. Examples of built-in transformation templates are classification of data points, melting and grouping of two datasets, validation of data points and repair of data points.
- **Result** - Boolean output of a workflow, multiple outputs can be added for later aggregation.

Each step in the diagram can be assigned a template that has the diagram's purpose and step's type tagged to it. Each template has a set of variables which are statically typed or might be typed as an enumeration of which the options can be resolved dynamically through the template engine.

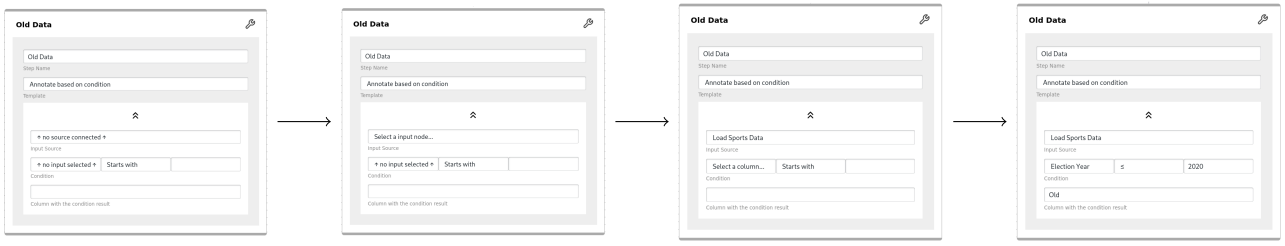

Figure 25: Dynamically filling options of template variables based on a step's state.

In case the variable can be resolved dynamically, the template can provide a message to show if other variables need to be set first, an example of which is shown in Figure 25.

[3]Supports GraphQL (Requires GraphQL schema) and RESTful APIs (Requires `OpenAPI 3.0` schema).

Workflows are run from the editor, using the controls at the top of the screen, shown in Figure 26. When the workflow is done, a toast is shown on the screen. First, the user renders the graph to a notebook. Then, the user can execute the cells from the notebook.

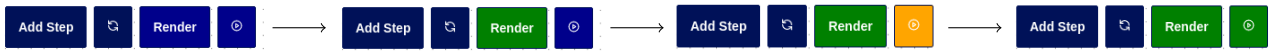

Figure 26: Control Buttons to run workflows in the client

The notebook's output and the generated code can be shown from the editor by opening the step.

If the notebook cell outputs HTML or plain text, this is rendered inside an iframe, so interactive elements can be passed on to the client without risking XSS. If the notebook cell outputs a dataframe, a random selection of 1000 rows will be transferred to the client and are displayed in a browsable and sortable table. An example of this is shown in Figure 27.

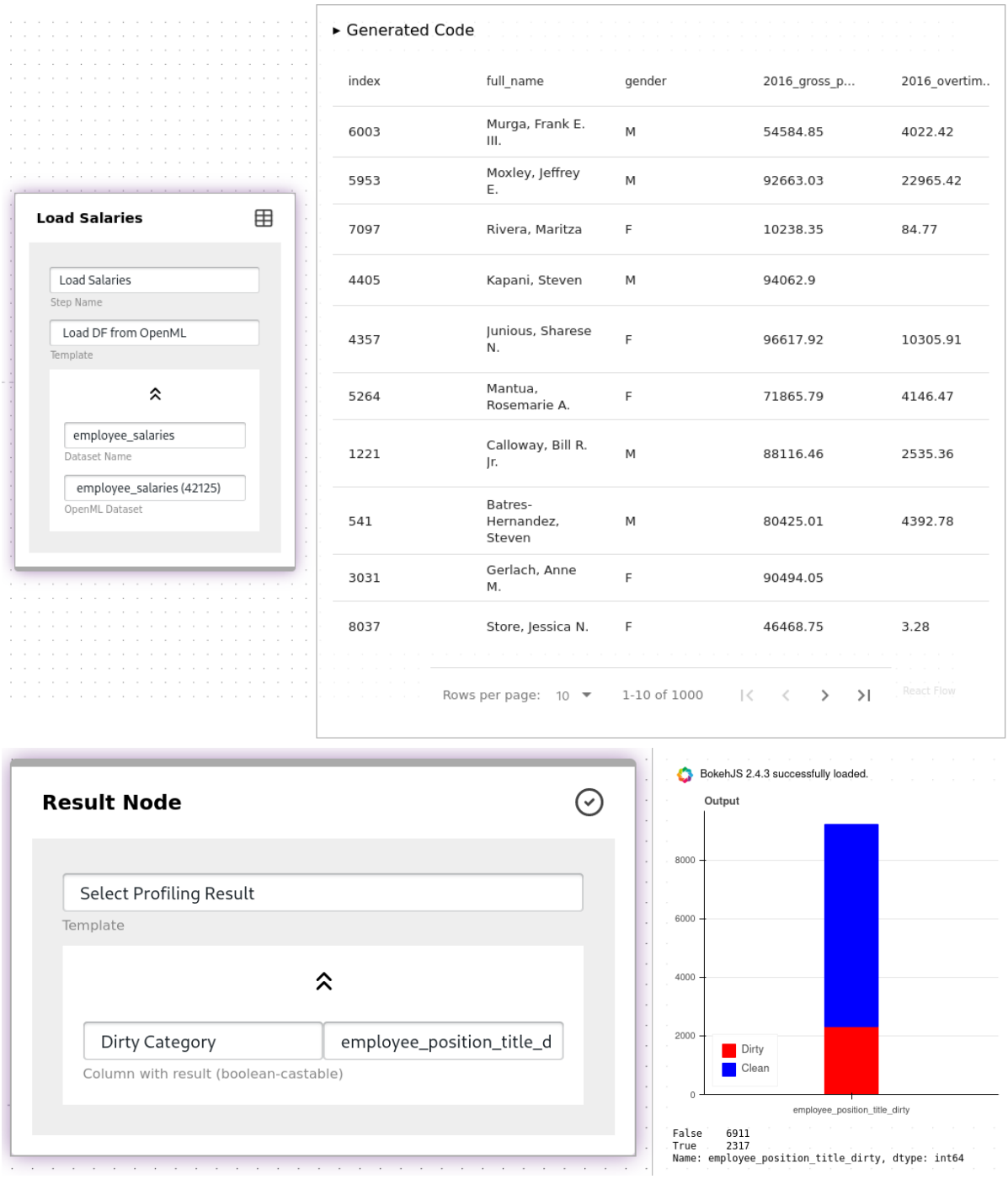

Figure 27: Cell output shown in the client interface

5 Case Studies

To test the execution and effectiveness of the framework, two datasets are chosen for cleaning:

- **Employee Salaries (2016), Montgomery County, MD**
  In *Similarity encoding for learning with dirty categorical variables* [55] this dataset was used to describe the performance of `dirty_cat` for cleaning dirty categories.
- **MycoDiversity DB (Martorelli, et al. 2020)**
  In *Fungal metabarcoding data integration framework for the MycoDiversity DataBase (MDDB)* [56], Martorelli faced data quality issues when trying to incorporate samples into a database because of different notations and unexpected values.

Both datasets were imported into the framework and were cleaned by creating workflows for multiple dimensions of data quality.

5.1 Employee Salaries (2016)

5.1.1 Profiling Workflows

For both dimensions, the dataset is imported into the engine. The data that is used is available on OpenML, so it can be imported directly from OpenML.

Completeness

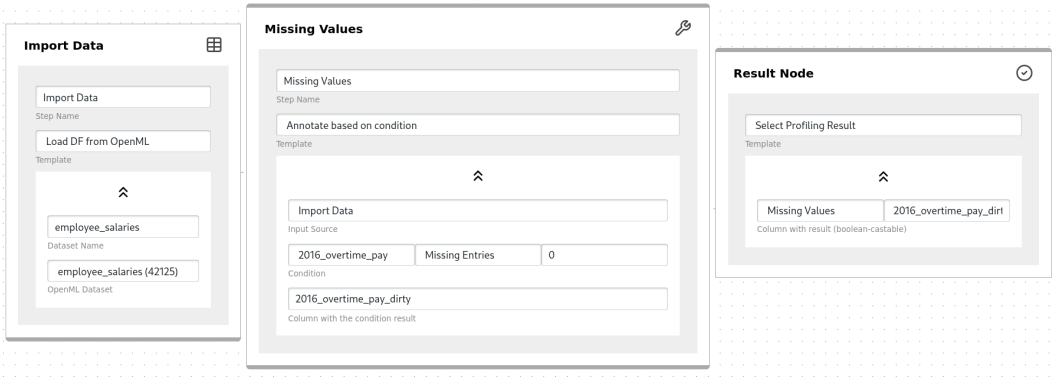

Figure 28: Employee Salaries: Profiling Workflow for Completeness

For completeness, the columns where there are missing values are considered. In this case the overtime column contains missing values, that should be zero.

Each missing value is annotated as dirty, and the annotated column is loaded into the results node.

Consistency

For consistency, the function titles are inspected. Working with the data, various persons doing the same work had a (slightly) different function title in the document.

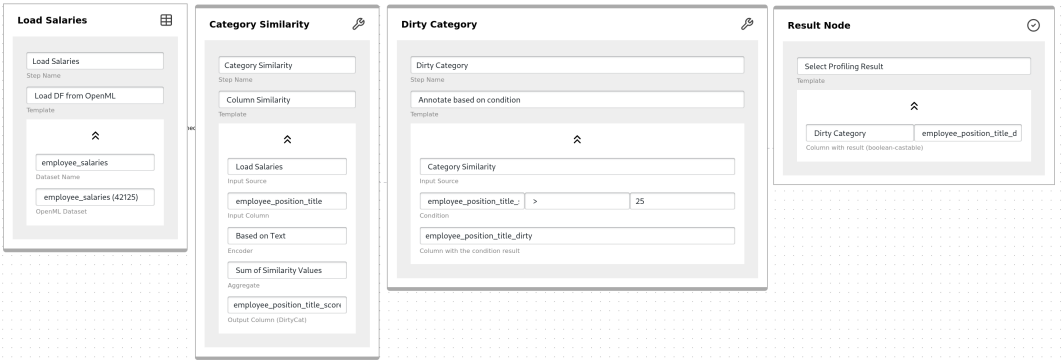

Figure 29: Employee Salaries: Profiling Workflow for Consistency

First, an n-gram based similarity encoder is run over the data (provided by the `dirty_cat` library). With the configured parameters, this step will sum the degree of similar but not exactly the same categories.

Finally, each category that has a degree of 25 or more very similar categories is annotated as dirty.

Outcomes

Of the Employee Positions,
2317 entries are marked as bad.
This is 25% of all entries.

Of the Overtime Fees,
2917 entries are marked as bad.
This is 32% of all entries.

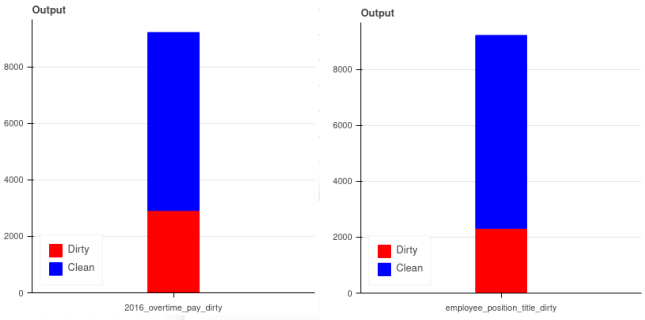

Figure 30: Employee Salaries: Profiling Outcomes

5.1.2 Cleaning Workflows

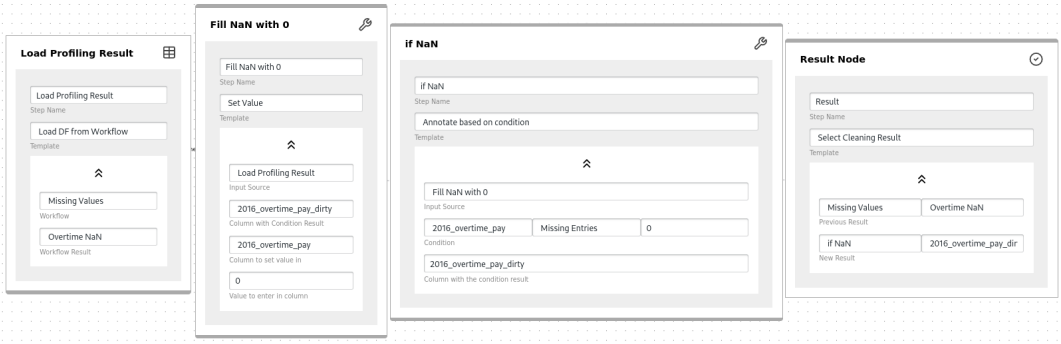

Figure 31: Employee Salaries: Cleaning Workflow for Completeness

Completeness For completeness, each missing value in the overtime column is replaced with a 0.

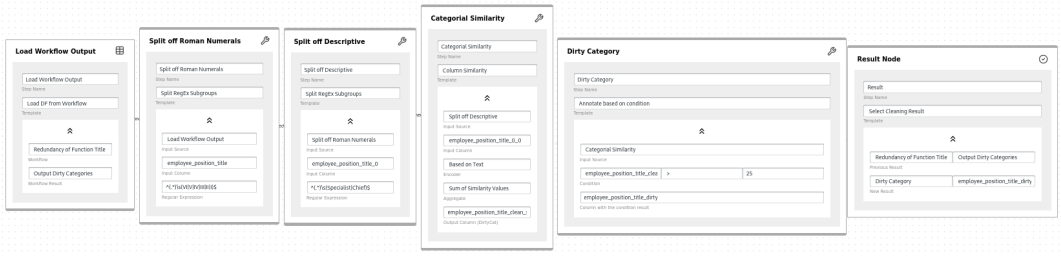

Figure 32: Employee Salaries: Cleaning Workflow for Consistency

Consistency For consistency, most columns that are highly similar end with a Roman numeral. Using a regular expression, the numerals are separated.

Remaining columns that are highly similar end with either `Specialist` or `Chief`. Using a regular expression, these titles are separated.

Outcomes

Of the Employee Positions, 0 entries are marked as bad. This is a 100% improvement compared to 2317.

Of the Overtime Fees, 0 entries are marked as bad. This is a 100% improvement compared to 2917.

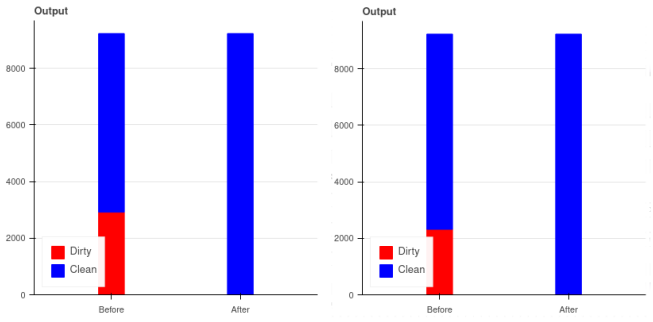

Figure 33: Employee Salaries: Cleaning Outcomes

5.2 MycoDiversity DB (2020)

5.2.1 Profiling Workflows

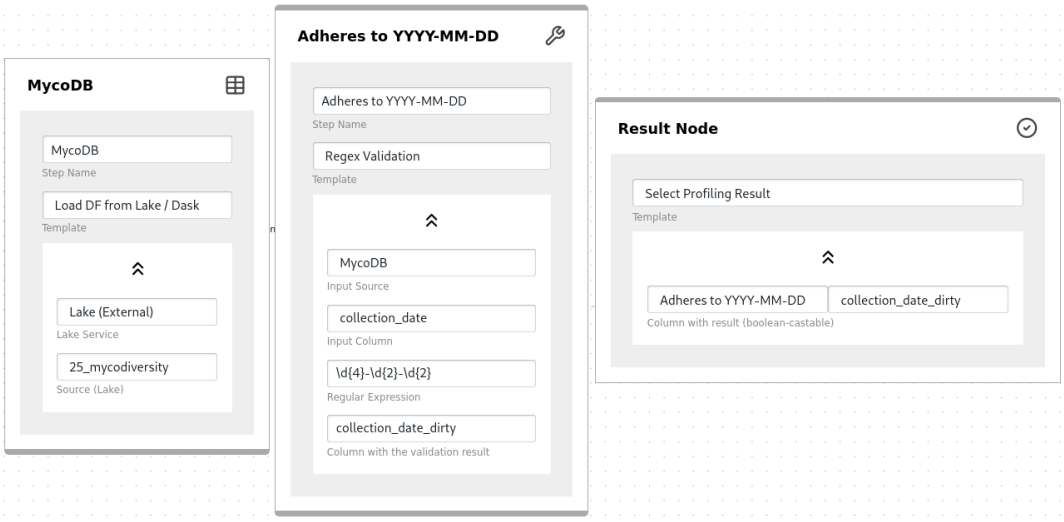

Figure 34: MycoDiversity DB: Profiling Workflow for Consistency

Consistency For consistency, the collection date is considered. Here, each date should be in the format `YYYY-MM-DD`.

Each value that doesn't match a regular expression for `YYYY-MM-DD` is annotated as dirty.

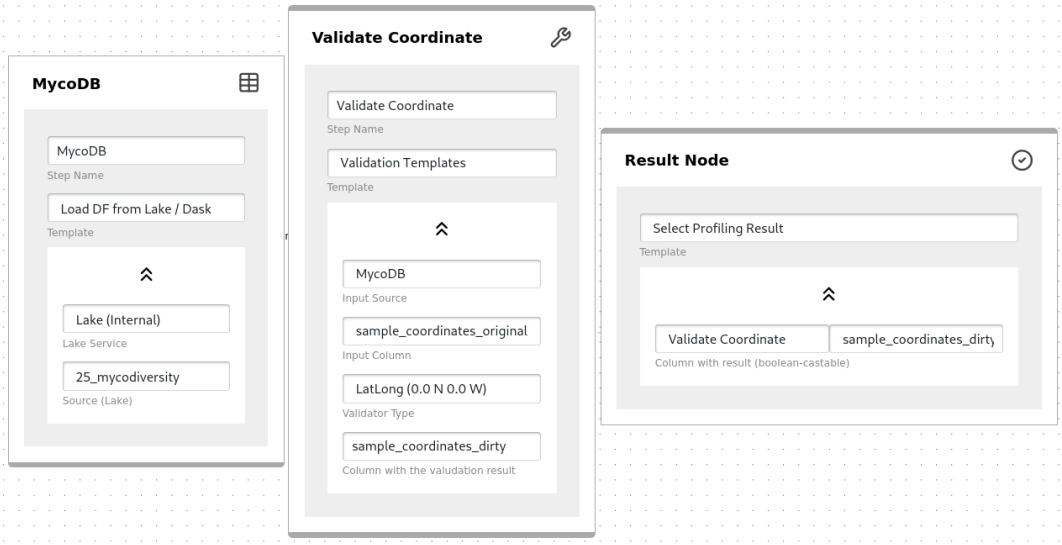

Figure 35: MycoDiversity DB: Profiling Workflow for Validity

Validity For validity, the coordinates of each sample are considered. These should be recorded as latitude-longitude, any other recordings are invalid (and incomplete). Each value that is not in a latitude-longitude format is annotated as dirty.

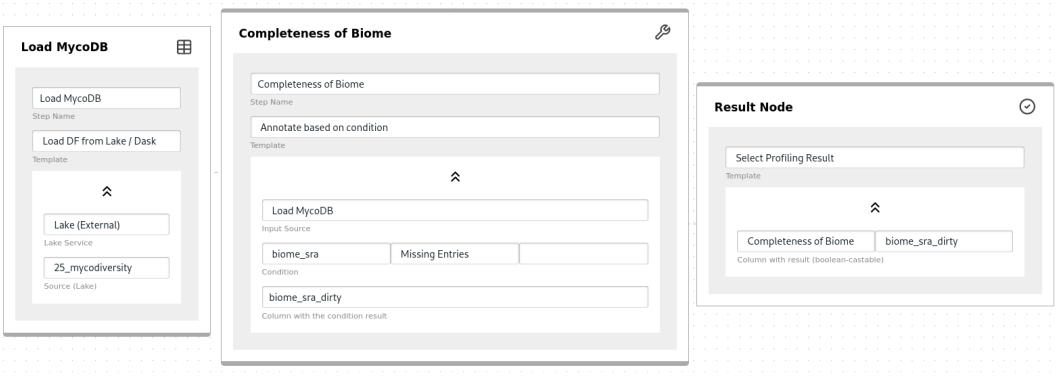

Figure 36: MycoDiversity DB: Profiling Workflow for Completeness

Completeness For completeness, the biome column is considered. If no biome type is recorded, consider the value as missing.

Each value that is missing is annotated as dirty.

Outcomes

Of the Collection Date,
1899 entries are marked as bad.
This is 43.7% of all entries.

Of the Coordinates,
1552 entries are marked as bad.
This is 46.7% of all entries.

Of the Biomes,
1313 entries are marked as bad.
This is 30.1% of all entries.

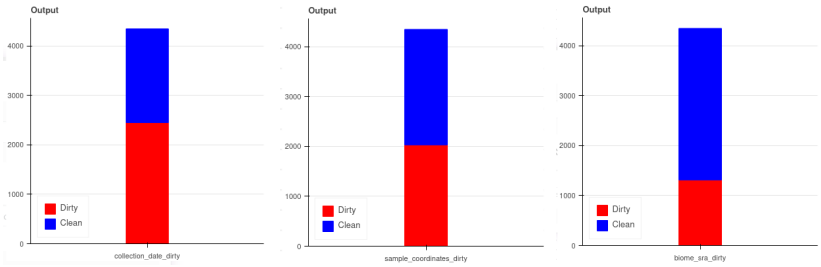

Figure 37: MycoDiversity DB: Profiling Outcomes

5.2.2 Cleaning Workflows

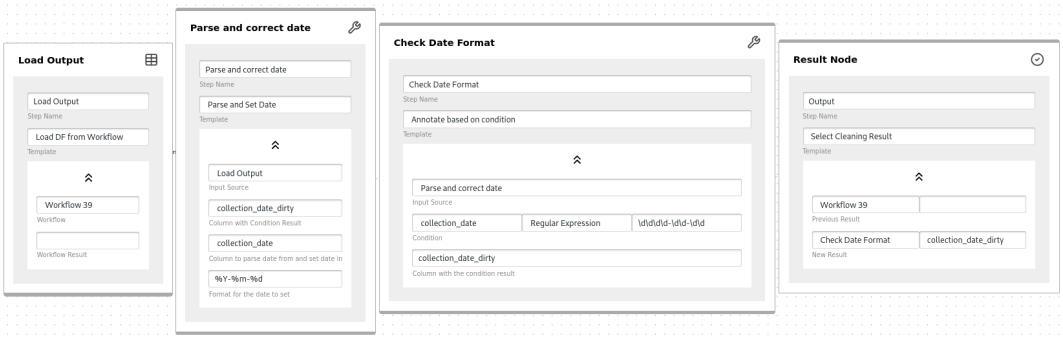

Figure 38: MycoDiversity DB: Cleaning Workflow for Consistency

Consistency For consistency, first all the dates are parsed if day, month and year are given. Then the dates that could be parsed are printed in the correct format.

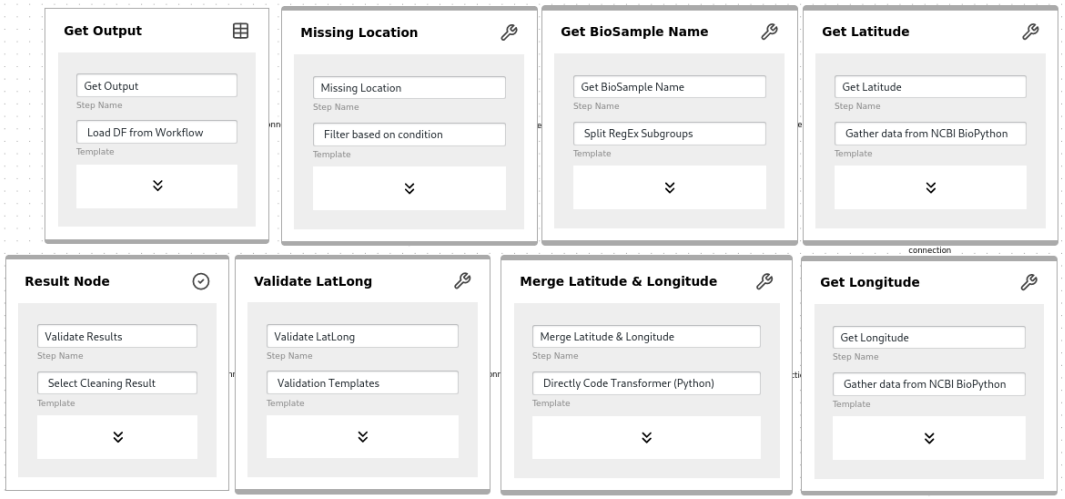

Figure 39: MycoDiversity DB: Cleaning Workflow for Validity

Validity For validity, first all valid rows are filtered out. Then, the NCBI BioSample ID is extracted from a given URL.

Continuing, a template to gather samples through BioPython Entrez [57] was built. This template loads and caches data from either Ensemble or BioSample based on an ID. This template is used to get the latitude and longitude for samples that store these separated.

Finally, the two columns are merged into one valid column.

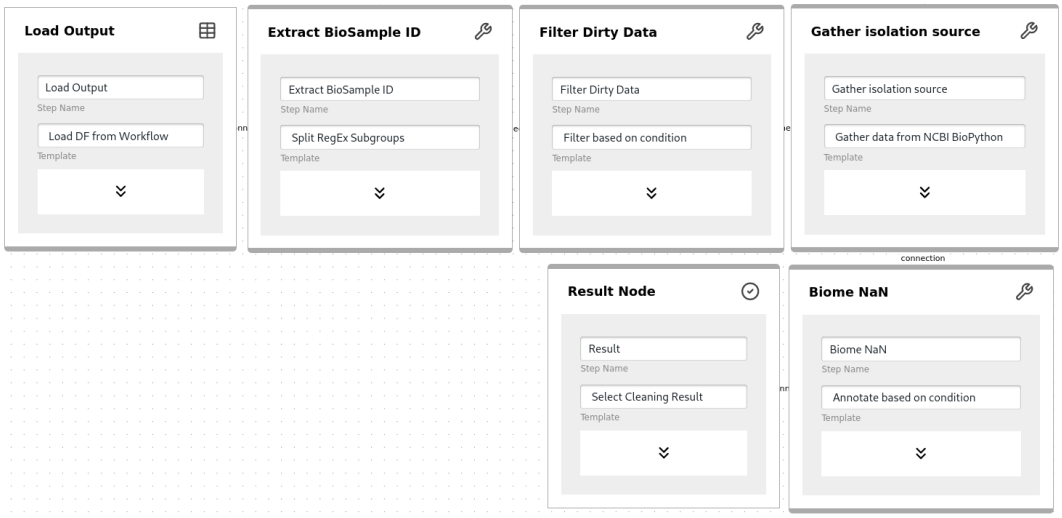

Figure 40: MycoDiversity DB: Cleaning Workflow for Completeness

Completeness For completeness, first all valid rows are filtered out. Then, the NCBI BioSample ID is extracted from a given URL. Then the NCBI BioPython template is used to retrieve isolation source field and store it in the biome column.

Outcomes

Of the Collection Date, 1552 entries are marked as bad.
This is an improvement of 17.8% over 1899.

Of the Coordinates, 923 entries are marked as bad.
This is an improvement of 54.6% over 2031.

Of the Biomes, 157 entries are marked as bad.
This is an improvement of 88.0% over 1313.

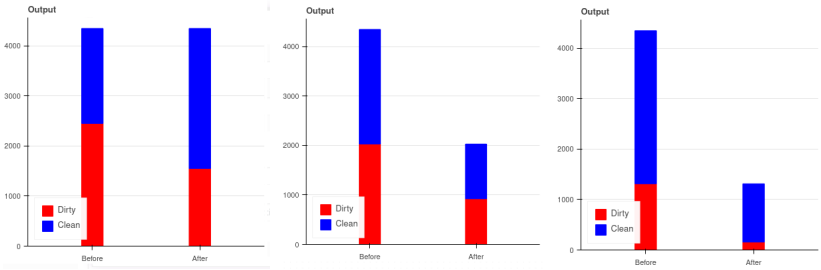

Figure 41: MycoDiversity DB: Cleaning Outcomes

6 Discussion

The results from the case studies show that the system can be used to classify data quality issues and can subsequently clean them. For the case studies, however, multiple processing templates were developed that are specific to either dirty categories or NCBI data. These templates are however reusable for other workflows where similar problems are faced. If the system is used by an end-user with the support of a consultant, the system can be populated with templates that are relevant for the user once. After this initial configuration, the user can create workflows and cleanse data however necessary.

The results from the architectural and interface work show that the initial research objectives are sufficiently completed. Reflecting:

1. **Extensible & turn-key data cleansing**
   An approach using the Dimensions of Data Quality of DAMA UK was chosen for test-set and covered the quality issues with that data. Furthermore, The ABC of Data Quality was added to the framework as an additional normalization option if DDQ does not cover all issues or if a more fine-grained approach is necessary. Other frameworks can be entered if needed.

2. **Low-code development principles**
   Templates to do profiling and cleaning are re-usable and executable through a low-code environment.

3. **Easy development & scaling**
   The framework can be deployed as a singular monolith, either directly on a Python-supporting OS or in containers. For large-scale deployments, the framework can be split into multiple microservices connected through a service-bus.

4. **Allows plug-in development**
   The framework is built as separate microservices in a singular Django project. Each microservice has an API and developers in a certain discipline can work behind that abstraction in the microservice. Furthermore, templates for loading, classification, cleaning and storing of data can be added to the database at runtime or as flat files in the project.

7 Conclusion

This thesis explores an intuitive and customer-oriented way to profile and successively cleanse data to improve data quality. Its main objective is to lay a good foundation for academic work and to explore various architectures to build a system that allows this.

An end-to-end system is built and (parts of) it can be adapted in future projects that may or may not fall in the sphere of data quality research. The system meets the objectives set in the introduction and was validated in two case studies. In the case study on salary data 25% to 32% of entries were classified as dirty, of which all were cleansed. In the case study of the MycoDiversity DB data, 30% to 47% of entries were classified as dirty, of which 18% to 88% were cleansed.

7.1 Future Work

In the system, the necessary functionality for meeting the objectives and completing the case studies was provided. However, for deployment of the application in enterprise applications, further work can be done. In the following paragraphs, future extensions to the system are proposed.

Software Development Kit There is no standardized development interface for the development of the Euridice templates. Consultants can only partially generate a workflow in the system and head to the JupyterLab-instance running in the backend to code the desired cell. This cell is not visible to the rest of the system and can not be tested. The cell must then manually be converted to a Jinja2-template and validated before publishing to the user.

For good developer experience and adaptation by the open source community of this system, options to build a development plug-in for existing IDEs should be explored. Either a plug-in that connects to a development Euridice instance or that can validate and generate previews of templates.

GraphQL Federation (Scout) Scout is an implementation of a custom federation system for the multiple services in the framework. There are off-the-shelf solutions (such as Apollo Federation [58, pp. 8-9]) for GraphQL federation that might be easier to maintain than scout.

In scout, each microservice is called by a unique identifier from the client so that the client can decide where a query should go. In Apollo Federation, the gateway should automatically resolve requests to the correct backend. A difficulty implementing Apollo Federation might be that in the overarching design of Euridice, sharding is possible between microservices for horizontal scaling: each contains only part of the data the GraphQL supergraph might query. For Apollo Federation to be applied, a separate sharding engine needs to be implemented.

Data Support & Storage (Lake) Lake only supports the storage of data as DataFrames in Dask. Herein, it's only possible to store CSV-files. In enterprise applications it might be needed to load data from other kinds of sources into the framework. Mapping of Dask's APIs needs improvement to support more formats. Or, a different backend for data can be used such as Apache Spark [59], for a wider support of data formats and sources. Implementing Apache Spark would also open possibilities to continuously stream and monitor data through the framework.

Workflow Storage (Flow) Flow stores workflow data and graph data in a SQL-database using nodes and adjacency lists. Storing graph data in SQL-databases is considered inefficient and makes performance improvements on graph searching difficult [60]. Implementing the support for a graph database such as Neo4J would greatly improve the performance of the graphing engine and efficiency of Flow. It would also support better research into graphs produced by the framework.

Alternatively, it might be useful to research a mapping of Flow's workflow engine to an existing DAG-based data science system such as Apache Airflow. This would also remove the requirement of an execution engine in the framework.

Execution Engine (Engine) Engine uses a JupyterLab server as a backend. Due to the generic nature of the JupyterLab server there is no possibility for optimizations such as caching of executed jobs and storage of job results. It might be interesting to remove the abstraction of JupyterLab and implement an execution backend directly.

References

[1] M. Bianchini and V. Michalkova, "Data analytics in smes," no. 15, 2019.

[2] M. Chui, B. Hall, A. Singla, and A. Sukharevsky, "Mckinsey global survey: The state of ai in 2021," *McKinsey Quantum Black*, 2021.

[3] J. Ladley and T. C. Redman, "Use data to accelerate your business strategy," *Harvard Business Review*, 2020.

[4] D. Donoho, "50 years of data science," *Journal of Computational and Graphical Statistics*, vol. 26, no. 4, pp. 745–766, 2017.

[5] L. Cai and Y. Zhu, "The challenges of data quality and data quality assessment in the big data era," *Data Science Journal*, vol. 14, p. 2, May 2015.

[6] T. C. Redman, "The impact of poor data quality on the typical enterprise," *Commun. ACM*, vol. 41, p. 79–82, feb 1998.

[7] EUROPEAN PARLIAMENT AND OF THE COUNCIL, "Directive (eu) 2015/2366," 2015. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex:32015L2366.

[8] The Federal Assembly of the Swiss Confederation, "Federal act on data protection (235.1)," 2019. https://www.fedlex.admin.ch/eli/cc/1993/1945_1945_1945/en.

[9] S. H. A. Chau, "Ab-375 privacy: personal information: businesses," 2019. https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375.

[10] FINRA, "Artificial intelligence (ai) in the securities industry," 2020.

[11] E. C. Bank, *Eleventh survey on correspondent banking in euro : 2019*. European Central Bank, 2021.

[12] E. C. Bank, "Report on the thematic review on effective risk data aggregation and risk reporting," 2018.

[13] T. Breur, "Data quality is everyone's business — managing information quality — part 2," *Journal of Direct, Data and Digital Marketing Practice*, vol. 11, pp. 114–123, Oct. 2009.

[14] M. A. Hernández and S. J. Stolfo, "Real-world data is dirty: Data cleansing and the merge/purge problem," *Data mining and knowledge discovery*, vol. 2, no. 1, pp. 9–37, 1998.

[15] R. Wieringa, "Design science as nested problem solving," in *Proceedings of the 4th international conference on design science research in information systems and technology*, pp. 1–12, 2009.

[16] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, p. 377–387, jun 1970.

[17] D. D. Chamberlin, "Early history of sql," *IEEE Annals of the History of Computing*, vol. 34, no. 4, pp. 78–82, 2012.

[18] Wes McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference* (Stéfan van der Walt and Jarrod Millman, eds.), pp. 56 – 61, 2010.

[19] J. Rowley, "The wisdom hierarchy: representations of the dikw hierarchy," *Journal of Information Science*, vol. 33, no. 2, pp. 163–180, 2007.

[20] E. F. Codd, "Further normalization of the data base relational model," *Data base systems*, vol. 6, pp. 33–64, 1972.

[21] R. Ramakrishnan and J. Gehrke, *Database Management Systems*. USA: McGraw-Hill, Inc., 3 ed., 2002.

[22] D. Henderson and S. Earley, *DAMA-DMBOK : data management body of knowledge*. 2017.

[23] R. Y. Wang and D. M. Strong, "Beyond accuracy: What data quality means to data consumers," *Journal of management information systems*, vol. 12, no. 4, pp. 5–33, 1996.

[24] T. C. Redman, *Data quality for the information age*. Artech House, Inc., 1997.

[25] L. P. English, *Improving data warehouse and business information quality: methods for reducing costs and increasing profits*. John Wiley & Sons, Inc., 1999.

[26] N. Askham, D. Cook, M. Doyle, H. Fereday, M. Gibson, U. Landbeck, R. Lee, C. Maynard, G. Palmer, and J. Schwarzenbach, "The six primary dimensions for data quality assessment," *DAMA UK working group*, 2013.

[27] N. D. Lawrence, "Data readiness levels," *arXiv preprint arXiv:1705.02245*, 2017.

[28] L. A. Castelijns, Y. Maas, and J. Vanschoren, "The abc of data: A classifying framework for data readiness," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pp. 3–16, Springer, 2019.

[29] J. Bicevskis, Z. Bicevska, and G. Karnitis, "Executable data quality models," *Procedia Computer Science*, vol. 104, pp. 138–145, 2017.

[30] ISO, *Quality management principles*. 2015.

[31] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[32] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2017.

[33] D. Feitosa, A. Ampatzoglou, A. Gkortzis, S. Bibi, and A. Chatzigeorgiou, "Code reuse in practice: Benefiting or harming technical debt," *Journal of Systems and Software*, vol. 167, p. 110618, 2020.

[34] E. Gamma, R. Helm, R. Johnson, R. E. Johnson, J. Vlissides, *et al.*, *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[35] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pp. 000149–000154, 2018.

[36] N. C. Mendonça, C. Box, C. Manolache, and L. Ryan, "The monolith strikes back: Why istio migrated from microservices to a monolithic architecture," *IEEE Software*, vol. 38, no. 05, pp. 17–22, 2021.

[37] T. Betts, "To microservices and back again - why segment went back to a monolith," *InfoQ*, 2020.

[38] D. H. Hansson, "The majestic monolith," *Signal v. Noise (Basecamp)*, 2016.

[39] H. Kreger and J. Estefan, "Navigating the soa open standards landscape around architecture," *Joint Paper, The Open Group, OASIS, and OMG*, 2009.

[40] Open Group, *Service-Oriented Architecture Ontology, Version 2.0*. the Open Group Library, 2014.

[41] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, *et al.*, "Serverless computing: Current trends and open problems," in *Research advances in cloud computing*, pp. 1–20, Springer, 2017.

[42] B. Wootton, "Microservices-not a free lunch!," *Dosegljivo: http://highscalability. com/blog/2014/4/8/microservices-nota-free-lunch. html*, 2014.

[43] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume 1, A System of Patterns*, vol. 1. John Wiley & Sons New York, 1996.

[44] A. Borysov, M. McLarty, and M. Garrett, "Api showdown: Rest vs. graphql vs. grpc – which should you use?." Panel discussion at QCon Plus - November 2021, Online, 2021.

[45] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[46] M. Marculescu, "Introducing grpc, a new open source http/2 rpc framework," 2015.

[47] B. J. Nelson, *Remote procedure call*. Carnegie Mellon University, 1981.

[48] GraphQL Specification Project, "Graphql," 2021.

[49] "Graphene-Python — docs.graphene-python.org." https://docs.graphene-python.org/en/latest/quickstart/#introduction. [Accessed 08-Aug-2022].

[50] "Introduction · Ariadne — ariadnegraphql.org." https://ariadnegraphql.org/docs/intro.html. [Accessed 08-Aug-2022].

[51] Meta, "Reactjs," 2013.

[52] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Science Conference* (K. Huff and J. Bergstra, eds.), pp. 130 – 136, 2015.

[53] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, *et al.*, *Jupyter Notebooks-a publishing format for reproducible computational workflows.*, vol. 2016. 2016.

[54] Django Software Foundation, "Django."

[55] P. Cerda, G. Varoquaux, and B. Kégl, "Similarity encoding for learning with dirty categorical variables," *Machine Learning*, vol. 107, no. 8, pp. 1477–1494, 2018.

[56] I. Martorelli, L. S. Helwerda, J. Kerkvliet, S. I. Gomes, J. Nuytinck, C. R. van der Werff, G. J. Ramackers, A. P. Gultyaev, V. S. Merckx, and F. J. Verbeek, "Fungal metabarcoding data integration framework for the mycodiversity database (mddb)," *Journal of integrative bioinformatics*, vol. 17, no. 1, 2020.

[57] NCBI Resource Coordinators, "Database resources of the national center for biotechnology information," *Nucleic Acids Research*, vol. 41, pp. D8–D20, Nov. 2012.

[58] P. Stünkel, O. von Bargen, A. Rutle, and Y. Lamo, "Graphql federation: a model-based approach," 2020.

[59] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[60] D. Fernandes and J. Bernardino, "Graph databases comparison: Allegrograph, arangodb, infinitegraph, neo4j, and orientdb.," in *Data*, pp. 373–380, 2018.