FlowETL : An Example-Driven Autonomous ETL Pipeline

Mattia Di Profio

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Bachelor of Science
of the
University of Aberdeen.

Department of Computing Science

April 27, 2025

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

Date: April 27, 2025

Abstract

The Extract Transform Load (ETL) workflow is fundamental for populating and maintaining data warehouses and other locations accessed by analysts for downstream tasks such as machine learning model training or informing enterprise decisions through data-driven insights.

A major shortcoming of modern ETL solutions is the extensive need for a human-in-the-loop, required to design and implement context-specific, and often non-generalisable transformations. While related work in the field of ETL automation shows promising progress, there is a lack of solutions capable of automatically designing and applying these transformations.

This report presents FlowETL, a novel example-based autonomous ETL pipeline architecture designed to automatically standardise and prepare input datasets according to a concise, user-defined target dataset. In essence, FlowETL is an ecosystem of components which interact together to achieve the desired outcome. A Planning Engine uses a paired input-output datasets sample to construct a transformation plan, which is then applied by an ETL worker. Monitoring and logging provide observability throughout the entire pipeline. The results show promising generalisation capabilities across 13 datasets of various domains, file structures, and file sizes.

Acknowledgements

I would like to express my gratitude to all those who have supported me throughout the course of this project. First and foremost, I am sincerely thankful to my supervisor, Dr. Mingjun Zhong, for his invaluable guidance, continuous support, and encouragement. A special thanks also goes to my colleagues and friends for their feedback and support. I would also like to extend my sincere appreciation to the team at Citi, where I had the opportunity to complete my summer internship, especially to my manager Chris Thomson and the entire GFT team for introducing me to the field of Data Engineering, and ultimately inspiring the idea behind this project. Lastly, and most importantly, I am eternally grateful to my family and my girlfriend Summer, for their love and belief in me. Thank you all.

Contents

Acronyms

API Application Programming Interface. 36, 40, 42, 43, 52

BIRD Big Bench for Large-Scale Databases. 42

CSV Comma-Separated Values. 16, 53, 56

DAG Directed Acyclic Graph. 19, 20, 37, 38, 48

DQS Data Quality Score. 19, 29, 31, 43, 47–49, 51

DRH Duplicate Rows Handler. 20, 21

DTN Data Task Node. 16, 19, 20, 31, 37, 41, 52, 57

ETL Extract Transform Load. 3, 5, 6, 8–11, 13, 15, 20, 31, 32, 36, 40, 42, 48–51, 53

GPU Graphics Processing Unit. 39, 40

GT Ground Truth. 39, 42, 44, 48

GUI Graphical User Interface. 9, 13, 52

IR Internal Representation. 15–20, 22, 24–26, 29–32, 36, 38, 39, 41, 47, 51, 57

JSON JavaScript Object Notation. 16–18, 27, 31, 41, 53, 56

LLM Large Language Model. 8, 10, 13, 14, 25, 27, 30, 31, 36, 38–47, 50–53, 58

MAD Median Absolute Deviation. 22

MVH Missing Value Handler. 20, 21

NOH Numerical Outliers Handler. 20, 22, 23

SRDT Success Rate for Data Transformation. 42, 43

TBE Transform By Example. 10, 11, 29

TBP Transform By Pattern. 10, 11, 29

XML Extensible Markup Language. 13, 16, 53

Chapter 1

Introduction

With the increasing prevalence of Big Data, the demand for robust data engineering solutions has grown significantly. Data engineering, otherwise known as data wrangling, involves collecting, organising, and transforming raw data into a format suitable for downstream tasks [13]. A common implementation of these tasks comes in the form of an ETL pipeline, where data is extracted from multiple sources, transformed into a desirable format, and loaded into its final destination for further consumption. The literature estimates that around 80% of a data analyst's time is spent on data wrangling tasks due to absence of a reliable and efficient procedure to transform data automatically [29]. Although many ETL tools have been developed, there is a persistent need for a framework which minimises the developer's efforts while providing a highly resilient and adaptable solution, particularly when handling unstructured data [8]. This report presents FlowETL : a fully autonomous, example-based, ETL pipeline architecture capable of handling missing values, numerical outliers, duplicate entries, as well as standardising a source file at both the schema and instance level. It only requires the developer to supply a subset of the source file, transformed into the target file, from which the pipeline can distil the desired transformations.

1.1 Objectives and Contributions

With FlowETL, the author aims to achieve an ETL pipeline architecture capable of efficiently and correctly addressing the data preparation requirements for most downstream tasks. FlowETL additionally aims to reduce the need for human involvement for ETL workflows design and monitoring, without compromising on runtime resources usage or output correctness. This project aims to contribute to the field of ETL research, with particular focus on automation. Additionally, FlowETL aims to contribute toward bridging the research gap concerned with Large Language Model (LLM)s within ETL pipeline design.

1.2 Report Structure

Chapter 2 gives an overview of the background and related work, where an introduction to the ETL pipeline architecture and its autonomous variants is provided. Additionally, it also introduces the schema matching problem, a fundamental step to the ETL process, and presents the solutions developed for this problem. Chapters 3 and 4 outline the project design and the implementation decisions respectively. Chapter 5 reports the evaluation methodology used to assess the generalisation capabilities of the pipeline and experimental results. Chapter 6 provides a discussion of the strengths and limitations of the project, as well as a comprehensive account of future works.

Chapter 2

Background

2.1 ETL Pipelines

The ETL workflow is an established solution for populating and maintaining data warehouses, often accessed by analysts for downstream tasks such as machine learning, data analysis, or data-driven decision making [29]. It involves three distinct steps, knows as the extraction step, the transformation or enrichment step, and the loading step. The extraction step is conventionally used to perform data integration, which requires combining related data from multiple, often diverse sources into a unified representation [35].

It is often the case that since data is collected from varying sources in the extraction step, it may contain data wrangling issues such as missing values, duplicates, outliers, or be represented in formats which make loading it into a data warehouse infeasible. The transformation step therefore plays a key role in standardising the data and addressing these inconsistencies. Furthermore, this step directly impacts the quality of the data, which might lead to performance losses on analysis or predictive tasks [50, 38]. Additionally, the transformation step may contain some feature engineering or enrichment steps, where the data is enriched using an external metadata repository [25]. Once the data is standardised and any data wrangling issues handled accordingly, it is written to a loading destination for further use.

The ETL pipeline architecture has streamlined the data engineering process, allowing for a scalable and versatile solutions tailored towards the 4 Vs of Big Data : variety, veracity, volume, and velocity [7]. Within the literature, many variants of ETL have been proposed and evaluated. Theses can be categorised into Graphical User Interface (GUI) tools which allow users with limited technical background to develop complex workflows via drag-and-drop interactions and programmable or scripting based tools, which often provide a more robust and configurable approach with a higher technical barrier of entry.

2.1.1 Autonomous ETL Pipelines

Given the substantial overhead associated with developing suitable transformations and enrichment instructions, recent advances in the development of ETL pipelines have attempted to automate the process, paying particular attention to the transformation step.

Mondal et al. [32] propose a system in which a recommender, consisting of multiple machine learning algorithms, interacts with a reporting agent to identify performance bottlenecks and optimise the overall pipeline efficiency.

Pattyam [38] presents a theoretical framework to generate schema mappings from example values in the target schema, indicating that providing reference data to guide automated preparation could be a practical approach.

Devarasetty [9] argues that traditional ETL pipeline architectures are often vulnerable to changing requirements, and that leveraging modern advances in machine learning could allow for dynamic adjustments of workflow designs, while reducing manual effort and improving data quality. The proposed solution uses Random Forests and Auto-encoders for anomaly detection, Natural Language Processing for data tagging, and Recurrent Neural Networks to automatically infer and apply data transformations.

Other solutions such as Data Build Tool (dbt) [1] focus their functionality on the transformation stage of ETL. It operates by directly executing SQL queries on the target database, and is therefore better suited for ELT pipelines such as Fivetran Transformations. This tools allows users to automatically configure the end-to-end ELT process within one environment, but requires initial setup overhead to configure the pipeline.

Recent advances in LLMs have introduced a new family of autonomous pipelines. Bodensohn et al. [5] and Kanagarla [24] have both developed and explored data engineering solutions which leverage LLMs to perform completely autonomous data wrangling. In particular, the authors of the former paper focus on the limitations of current LLMs, and identified challenges such as operational costs and poor generalisation to enterprise data as major bottlenecks towards the adoption of this technology for more concrete applications.

2.2 Overview of Current Autonomous ETL Solutions

Autonomous ETL workflows have been developed from two approaches. The first approach, known as Transform By Pattern (TBP) aims to identify a sequence of transformation steps based on patterns inferred from both the source and target files, while the second approach, Transform By Example (TBE) attempts to automatically infer transformations by analysing instance-level data in the source and target files.

2.2.1 Pattern-Driven Architectures

Jin et al. [23] propose a new approach to data transformation based on input-output data patterns, tailored towards tasks such as data repair and data integration. A TBP program is formally defined by the authors as a triplet (P_s, P_t, T), where P_s and P_t are syntactic regex patterns describing the source and target column for which the corresponding program $T(P_t, P_s)$ is applicable. The authors then leverage a pre-constructed lookup table to infer the transformation program. While the results reported are promising, this approach remains unfeasible. The main limitation is constructing the transformations lookup table due to the extensive amount of time and computational power required to process web tables and other publicly available artifacts leveraged in the development of this tool.

2.2.2 Example-Driven Architectures

The literature is richer regarding example-driven transformation engines, with several solutions having been developed and evaluated.

[1] `https://www.fivetran.com/learn/what-is-dbt`

Foofah [22] is a TBE program developed under the assumption that data should be transformed without any extra semantic information, i.e. abbreviations, and that transformations should not add new information not contained in the input table, such as adding a column header or enriching columns using external sources.

The authors identified two categories of transformations: syntactic transformations, which operate at the instance level, and layout transformations, which affect the overall structure of the table. Foofah is tailored towards the latter, and heavily inspired by the Potter Wheel Project [41]. Foofah frames inference of data transformations as a search problem in a state space graph, where each path from the source to the target table represents a sequence of transformation operations. It uses a heuristic search approach, inspired by the A* algorithm [17], along with pruning strategies to efficiently synthesise transformation programs. According to the authors, Foofah was able to synthesise perfect programs for 90% of test scenarios. While Foofah is a promising tool, its limited capabilities hinder its applicability to more involved data engineering problems, which require both syntactic and layout-based transformations, alongside handling data wrangling issues.

In DataXFormer [33], the user provides the column labels of the input and output values for the desired transformation, then DataXFormer uses its two search engines to find relevant web tables and forms, from which it extracts the needed information. Unlike systems that rely only on examples, it requires column headers to create keyword queries. However, it struggles with tasks that go beyond structured sources like web forms and tables. Additionally, it shares a similar limitation to TBP, namely that it requires an extensive amount of set up. Precisely, 112 million web forms and tables are used during the search phase of the transformation inference stage.

He et al. [18] developed a system that searches an index of 50,000+ functions from websites such as GitHub [2] and StackOverflow [3] to synthesise a transformation program that matches the user-provided examples. This allows for a wide range of both syntactic and semantic transformations, however under-represented domains may be unsupported if no relevant functions exist in the index. Furthermore, synthesising correct transformations remains challenging for ambiguous or under-represented inputs, making TBE a system with high recall but low precision.

In more recent work, Singh and Gulwani [45] developed a system which uses a domain-specific language for semantic string transformations and syntactic operations. It works by taking input-output examples and searches through a space of candidate transformations, then a ranking heuristic is used to select the most promising transformation. As users provide more examples, the system iteratively refines its output, making it a semi-autonomous solution. A major limitation is the size of the possible transformations set, which can be extremely large even for simple examples, posing scalability challenges. Additionally, the system struggles to handle infinite domains, such as arithmetic transformations on numerical instances.

2.3 Schema Matching

Schema matching is a core step in the dataset standardisation process, and is often part of a larger ETL workflow [3]. Schema matching consists of finding a satisfactory mapping between columns of a source dataset with columns of a target dataset. Across the literature, the notion of a *Match*

[2] `https://github.com/`

[3] `https://stackoverflow.com/questions`

operator is mentioned extensively, with Rahm and Bernstein [40] defining the Match operator as a function which produces a mapping between elements of the two schemas that semantically correspond to each other, and Giunchiglia et al. [16] describing it as an operator which produces a mapping between nodes of a bipartite graph.

The schema matching problem has been modelled in the literature as a bipartite graph matching problem [14]. Namely, the schema matching problem can be represented by an undirected bipartite graph $G = (X,Y,E)$, where X and Y represent the source and target schemas and E represent the set of weighted edges between the two schemas, with each edge (x,y) having a weight $w \in [0,1]$ based on a similarity score between x and y. Within this abstraction, a mapping M is then defined as a a subset of E. Additionally, Gal [14] identified three classes of matches:

- $1:1$, where M is a subset of pair-wise disjoint edges of E
- $1:n$, where $x \in X$ maps to multiple $y_1, y_2, \cdots \in Y$, known as Replication.
- $n:1$, where multiple $x_1, x_2, \cdots \in X$ map to a single $y \in Y$, known as Decomposition.

2.3.1 The Stable Matching Problem

Abstracting the schema matching problem to a bipartite graph matching one makes the family of algorithms tailored towards the Stable Matching Problem a suitable candidate for a solution, with the Gale–Shapley algorithm serving as a representative implementation.

The Gale-Shapley algorithm [15] finds a stable matching (if one exists) M between two sets, X and Y, through a series of proposals. In this context, a proposal is defined as an assignment of a node $x \in X$ to a node $y \in Y$.

Each free $x \in X$ proposes to their most-compatible un-matched $y \in Y$. Members of Y accept proposals if they are unmatched to any other x or prefer the new proposer over their current match. This process repeats until everyone is matched, resulting in a stable pairing where no (x,y) pair would prefer each other over their assigned partners.

Formally, a match $M = \{(x_i, y_i) : x \in X, y \in Y\}$ between sets X and Y is deemed stable if for all pairs in M, x is y's most compatible node and vice versa [3], [48]. Compatibility in this case is measured in terms of edge weight. If edge $e_1 = (x_1, y_1)$ has weight of 0.45, and edge $e_1 = (x_1, y_2)$ has weight of 0.60, then x_1 is more compatible with y_2 than y_1.

Due to the inherit heuristic nature of the matching process, incompleteness and incorrectness are major risks associated with this problem[3]. Tan [48] further expands by defining a match M unstable if there is a pair $(x,y) \notin M$ such that x and y are more compatible than any other mapping $m \in M$ involving either x or y. Such a pair is said to block the matching M and a stable match cannot be reached.

In some cases, a stable matching may not exist. This can happen when a subset of nodes forms a cyclic preference structure, where each member prefers someone else in the group over their current match, leading to a chain of blocking pairs. Such a subset, known as an odd party [48] disrupts the stability of the entire system, making it impossible to achieve a complete stable matching that includes all participants.

Tan and Yuang-Cheh [49] redefine stability by introducing the concept of a stable partition. A stable partition extends the idea of a matching and accommodates cases where no conventional stable match exists. This structure ensures that even if individual pairwise stability cannot be achieved, a broader form of equilibrium is attainable.

In summary, the Gale-Shapley algorithm and its variants provide a promising approach to solving the schema matching problem, however any robust implementation must take into account any issues which might hinder the existence or reachability of a stable match.

2.3.2 Overview of Current Schema Matching Algorithms and Heuristics

Bernstein et al. [3] argue that one-shot schema matchers can lead to suboptimal results as they only show the best option according to their internal evaluation system, which may not always align with the optimal match requirements.

Moreover, one-shot matchers tend to produce false positives, which require extensive manual work to fix. They instead propose a ranking system based on two factors: schema-based techniques such as lexical similarity between nodes, and the user's previous matching actions. Furthermore, they introduce a method to fine-tune the matching process using user-validated mappings. The system ranks matches iteratively until a satisfactory match is inferred and returned. The unsuitability of this system for larger schemas poses a significant limitation. Additionally, the GUI-based component of the system developed leads to a linear increase in user workload as the size of the matching problem increases.

Gal [14] addresses the limitations of one-shot schema matchers by generating and evaluating the top-K schema mappings simultaneously. A Stability Heuristic positively scores mappings that consistently appear across these K candidates. This approach supports $1:1$, $1:n$ (replication), and $n:1$ (decomposition) mappings. However, it is computationally intensive for large datasets and relies heavily on the quality of initial mappings.

MAXSM [2] propose a heuristic-based approach to Extensible Markup Language (XML) schema matching, which incrementally computes similarities between schema elements using a similarity function $sim(n_1, n_2) \rightarrow [0, 1]$ computed by combining multiple heuristics, such as natural language similarity using WordNet [4], a tree-spanning method to detect structurally similar node clusters, and location path-based heuristics to assess node similarity. It then applies threshold-based decision-making to determine potential mappings. These similarity scores are recorded in a matrix, which informs the final mapping decisions in the output construction phase. While MAXSM poses a promising multi-heuristic approach, it has not yet been implemented nor evaluated.

Similarity Flooding [30] is a graph-based matching algorithm that transforms schemas and data instances into directed labelled graphs. The approach uses an iterative computation algorithm to propagate initial textual similarity scores among adjacent nodes in the graph, until a stable mapping is reached. This method requires minimal application-specific heuristics and is adaptable to various schema types, demonstrating considerable versatility. However, the algorithm is sensitive to the quality of the initial similarities and often requires human intervention to adjust mismatches, which highlights its poor suitability for independent environments, such as autonomous ETL workflows.

Recent advances in LLMs have laid the foundation for new schema matching solutions.

[4] `https://wordnet.princeton.edu/`

Matchmaker[42] proposes a self-improving program which first creates a collection of multi-vector documents from the target schema and then leverages semantic retrieval and reasoning-based candidate generation to obtain a smaller subset of candidate matches for further evaluation. While being a promising tool, Matchmaker requires integration with other tasks and human oversight to effectively generate reliable data usable for downstream tasks, limiting its applicability within completely autonomous workflows.

ReMatch [43] leverages retrieval-enhanced LLMs, which are tailed to support human matchers. It eliminates the need for predefined mappings, model training, or direct access to the source database and instead uses LLMs to perform semantic ranking between schemas. The underlying method involves converting target schema tables and source schema attributes into structured documents, using a text embedding model to identify the top candidate tables for each source attribute based on semantic similarity, and then employing an LLM to rank the most similar target attributes, producing a ranked list of potential matches.

Magneto [28], comprises two key components: a candidate retriever and a re-ranker. The idea at the core of Magneto's design is to first use a simpler, more cost-effective method to retrieve and filter candidate mappings, then use a more advanced and resource-intensive approach to identify the correct matches from a smaller pool of candidates. The authors claim that this strategy reduces the overall schema matching costs while maintaining a high level of accuracy. One limitation of the paper is that Magneto was developed and evaluated solely on medical domain data, making it difficult to assess its generalisability to other domains.

Chapter 3

Design

In this chapter the main components of the architecture, their role, and their interactions are introduced. Namely, these components are the Observers, the Planning Engine, the Messaging System, the ETL Worker Pipeline, and the overarching Reporting Engine. In addition, an outline of the design decisions and alternatives considered for each component is provided.

3.1 Overview of System Architecture

As displayed in Figure 3.1, the Observers are responsible for detecting the source file to be standardised and the target file which informs the data wrangling plan generation process. The Planning Engine is responsible for composing an executable plan of data wrangling tasks which both standardise the source file according to the target and improve its estimated data quality with respect to missing values, duplicates, and outliers. The ETL Worker ingests the source file and computes a plan, which is then applied to to the former. The Messaging System and Report Engine act as supporting component, enabling exchanges of payload and information between components and logging and reporting of runtime metrics, respectively.

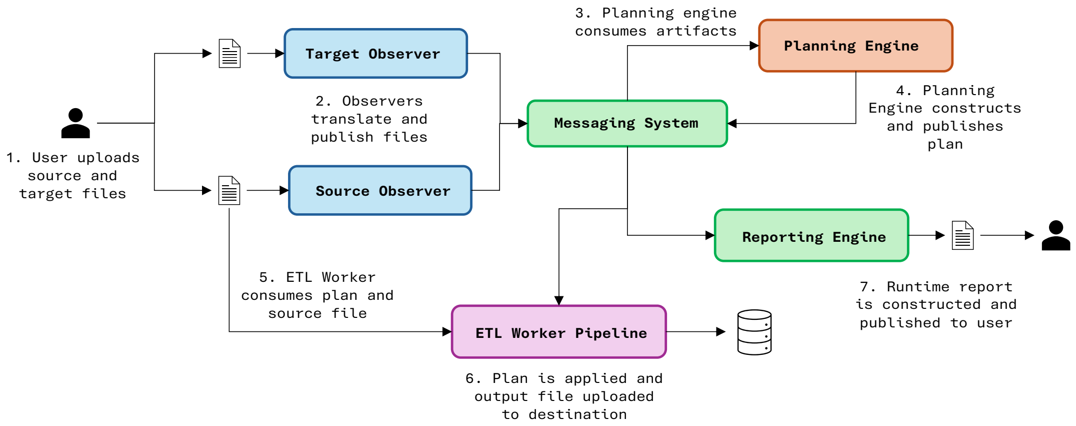

1. User uploads source and target files
Target Observer
Source Observer
2. Observers translate and publish files
3. Planning engine consumes artifacts
Planning Engine
Messaging System
4. Planning Engine constructs and publishes plan
Reporting Engine
5. ETL Worker consumes plan and source file
ETL Worker Pipeline
6. Plan is applied and output file uploaded to destination
7. Runtime report is constructed and published to user

Figure 3.1: Simplified FlowETL system architecture and workflow diagram

3.2 Abstractions

In order for the pipeline to support both structured and unstructured datasets, two core abstractions have been made throughout the design stage. The first abstraction defines a set of internal data types, while the second abstraction translates both structured and unstructured files into an Internal

Representation (IR).

3.2.1 Internal Data Types System

The abstraction introduces generalised data types capable of supporting a wide range of transformations, deliberately limiting the edge cases encountered during type conversion. This design presents a trade-off between the precision of data types and the potential risk for runtime errors.

Additionally, the chosen data types must be able to represent all data types which could be possibly contained within files of type Comma-Separated Values (CSV) and JavaScript Object Notation (JSON). Namely, this means we must concisely model simple entries of type string, float, boolean, etc. as well as more complex structures such as nested objects and lists. Another requirement identified is to find a way to label ambiguous collections of values, which are mostly found in unstructured files.

Several approaches were evaluated to model FlowETL data types. Pandas data types [1] were considered due to the framework's support for data operations. Apache Spark data types [2] were explored for similar advantages. Ultimately, data types were designed and developed from scratch to allow greater flexibility and avoid being constrained by either technology.

The FlowETL data types and corresponding abstracted types are outlined in the Table 3.1

FlowETL Data Type	**Abstracted Types**
number	numerical values
string	characters and strings
boolean	symbols representing truth values
complex	lists, dictionaries
ambiguous	columns containing multiple data types

Table 3.1: Column data types supported internally by FlowETL and the corresponding data types abstracted

3.2.2 Internal Representation

Abstracting both structured and structured files into an IR allows for an pipeline architecture which can be easily adapted to handle other file types, such as XML, by simply providing the conversion logic to read the file's contents into the IR.

Moreover, this abstraction allows for increased maintainability. For instance, each Data Task Node (DTN) can be defined only once and designed to operate on the IR, therefore there is no need for a separate task node variant to handle each of the supported file types.

The IR mechanism works by extending unstructured files into a tabular format inspired by Dataframes, a popular construct in both Apache Spark [3] and Pandas [4]. For CSV files, the algorithm is trivial : the file's contents are translated into a 2-D matrix with the first row denoting the column names and each subsequent row representing a row in the original file. For JSON files, the process is more complex. A major assumption made during the read-in process is that the source data within the JSON file is stored in a list of JSON objects. The extraction algorithm 1 leverages

[1] `https://pandas.pydata.org/docs/reference/arrays.html`

[2] `https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/data_types.html`

[3] `https://www.databricks.com/spark/getting-started-with-apache-spark/dataframes`

[4] `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.htm`

recursion to traverse the possibly nested structure of a JSON file until a value of type `list` is found and returned.

Algorithm 1 Extract First List Function

```
function EXTRACTLIST(collection, key = null)
    if ISINSTANCE(collection, list) then
        return (key, collection)
    else if ISINSTANCE(collection, dictionary) then
        for all (k, value) in collection do
            result $\leftarrow$ EXTRACTLIST(value, k)
            if result $\neq$ null then
                return result
            end if
        end for
    end if
    return null
end function
```

The process of inward translation, from source file to the IR, begins by invoking the `extract_list()` 1 method, which retrieves the first list found along with its associated key. Subsequently, a union of keys from all objects in the list is created to form the headers of the IR. Each object's keys are extended to include all the headers, with placeholder values added for any extended keys. The values for each object are then translated into rows in the IR. When translating outward, from the IR back to the source, the key from the source file is located, and if not found, it is created. For each row, a dictionary is formed with column names as keys and their respective cell values as values, skipping placeholders.

This translation mechanism meets the abstraction requirements outlined while avoiding any loss of information when translating to the internal representation or gain of redundant information when translating back up to the original file type.

3.3 Observers

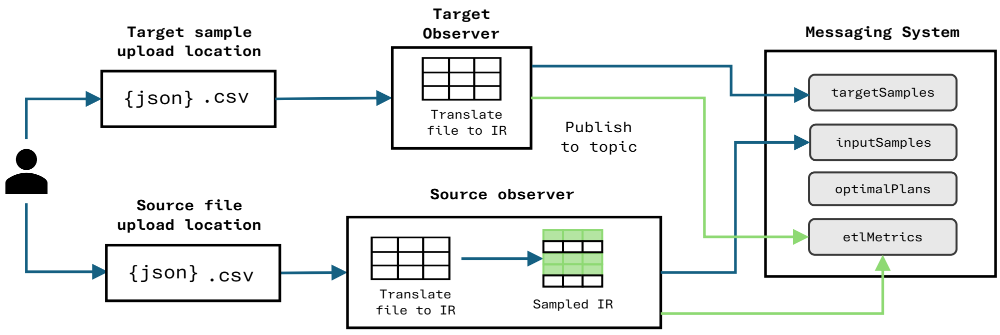


Figure 3.2: Source and Target Observers interaction with the Messaging System. Blue edges denote data artifacts, green edges denote payload containing runtime metrics.

The Observers, shown in Figure 3.2, are designed to prepare the files of interest through a series of pre-processing steps. Namely, these steps are ingesting the source-target files pair, translating them to an IR, and publishing these artifacts to the Messaging System. Through this separation of concerns, the system is kept modular and the behaviour of each observer can be monitored and modified in isolation, which improves maintainability of the system [31].

Both observers additionally record information at runtime about the files processed, such as file name, objects count, file size, and an optional key associated returned by the `extract_list()` 1 method if the source file is of type JSON. These metrics, shown in Figure 3.3 are separately published to a dedicated topic on the Messaging System which to be then consumed by the Reporting Engine throughout the pipeline runtime.

```
{
    "from": observerName,
    "contents": {
        ’filename’: filename,
        ’objects_count’: count,
        ’filesize_mbs’: fileSize
    }
}
```

```
{
    "name": filename,
    "associated_key": key,
    "contents": sampledIR
}
```

Figure 3.3: Observers payloads to Broker. Runtime metrics payload on the left, and source file artifacts on the right

Both Observers listen for any event of type *upload* occurring in the extraction locations. Once the user, or any upstream process, uploads a source file, these components will translate its contents into an Internal Representation using the `to_internal()` 2 method.

Algorithm 2 Inward Translation Algorithm

```
Input: filePath
Output: (associatedKey, IR)
fileType, fileName ← filePath
IR ← [ ]
if fileType is csv then
    IR ← TRANSLATECSV(fileName)
    return (None, IR)
else if fileType is json then
    dictionary ← JSONTODICTIONARY(fileName)
    (associated_key, objects) ← EXTRACTLIST(dictionary)
    if objects is None then
        return (None, None)
    end if
    attributes ← union of objects.keys
    IR ← IR ∪ {attributes}
    for all object in objects do
        row ← EXTEND(object.values, "_ext_")
        IR ← IR ∪ {row}
    end for
    return (associated_key, IR)
else
    return (None, None)
end if
```

If the Source Observer is triggered, it randomly samples a subset of rows from the IR. This step is carried out to reduce latency, serialisation, and processing time, especially during the plan construction phase. While this approach is expected to yield faster information exchange between components [20] it is expected to compromise the quality of the plan computed since it assumes that data in the file is randomly distributed [37], which most of the time is not. The effects associated with a changing sample size on the pipeline's output are further explored in Chapter 5. If the Target Observer is triggered, no sampling is performed, since any target file is expected to be relatively small (5-20 objects at most).

3.4 Messaging System

The following requirements were identified for a robust messaging system. Firstly, it should allow communication between components in isolation to non-participating ones. Secondly, it should be able to scale accordingly to system requirements. Lastly, it should support sequential processing of messages from an offset, as the Planning Engine may cause a performance bottleneck if artifacts are being published by the observers faster than the Planning Engine can handle.

Apache Kafka [5] was identified as a suitable solution to the requirements identified. In particular, Kafka provides a concrete implementation of the Publisher-Subscriber model [19], in which subscribers express interest in receiving messages and publishers publish them to topics without specifying a message's recipient(s). This pattern is decoupled and anonymous, meaning that components can be configured to publish/subscribe to and from topics relevant to their task.

A risk identified during the design stage is that the entire pipeline relies on the messaging system to operate, meaning that information exchange between components is disabled in case of failure. Therefore, the chosen messaging system must provide strong guarantees in regards to availability and reliability. Apache Kafka fulfils this requirement with its replication mechanism and reliability guarantees [36]. While it doesn't eliminate the risk of total failure, it reduces it to a level that is sufficiently low. The Messaging System, depicted in Figure 3.4, has been configured with four pre-defined topics, outlined in Table 3.2.

Topic	**Publisher(s)**	**Subscriber(s)**	**Contents**
etlMetrics	Observers, PE, ETLw	Reporting Engine	Runtime Metrics
inputSamples	Source Observer	Planning Engine	sampled source IR
targetSamples	Target Observer	Planning Engine	target IR
optimalPlans	Planning Engine	ETL Worker	Transformation Plan

Table 3.2: Outline of Messaging System's topics, producers and consumers, and message contents. Abbreviations are PE - planning engine, ETLw - ETL worker.

3.5 Planning Engine

The Planning Engine, shown in Figure 3.9, is responsible for constructing a transformation plan which once applied to the sampled source dataset maximises its Data Quality Score (DQS). In this context, a plan refers to a Directed Acyclic Graph (DAG) composed of DTNs that represent a sequential series of transformation steps. These steps aim to maximise data quality by addressing

[5] `https://kafka.apache.org/`

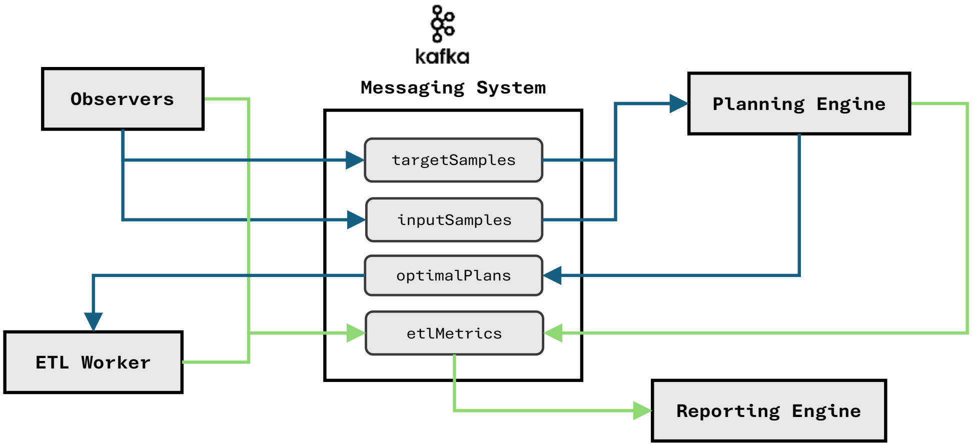


Figure 3.4: Messaging system topics overview. Blue edges denote file data, green edges denote runtime metrics. Incoming edges denote data being published to a topic, outgoing edges denote data being consumed from a topic.

data wrangling issues in the source file changing its structure and contents to follow the format specified in the target file.

The Planning Engine was designed based on the structure of a DAG, with each step in the planning process being confined to one DTN only. Choosing this modular approach enhances both testing and maintainability [12].

Given the broad scope of Data Engineering, the initial set of DTNs has been developed to address common data wrangling challenges associated with data standardisation and quality assurance, such as missing value handling, deduplication, and outliers detection. These issues are a recurring theme in the subfield of Data Integration [35], which concerns processes that combine incoming data from multiple sources and represents a typical application area of ETL tools [25].

3.5.1 Data Task Nodes

A DTN is a construct designed to keep the system modular and extendable. Each node takes in a series of inputs, which always include an and intermediate IR and additional artifacts required by the specific node, which then applies its logic onto the IR and makes it available for downstream tasks.

This consistency enables nodes to be connected into a sequential data wrangling plan and lays the groundwork for integration of additional nodes in the future. A node may additionally take in a `strategy` parameter, which determines how the it approaches the data wrangling issue it was designed to solve. The nodes currently available in the FlowETL ecosystem are the Missing Value Handler (MVH), the Duplicate Rows Handler (DRH), and the Numerical Outliers Handler (NOH).

Missing Value Handling

The MVH node, depicted in Algorithm 3, is responsible for detecting and handling missing values within the IR. Its input parameters are the IR to be operated on, the schema of the IR's column headers, and a third `strategy` parameter which determines whether the missing values will be imputed with an appropriate placeholder for their data type or dropped completely.

Algorithm 3 Missing Value Handler

```
Input: IR, strategy
Output: IR
if strategy is "impute" then
    for cell in IR.values do
        if cell is None or "" then
            cell ← IMPUTE(cell, column.type)
        end if
    end for
else if strategy is "drop.rows" then
    nullRowIndices ← GETNULLROWINDICES(IR.values)
    for index in nullRowIndices do
        IR ← IR \ {IR.values.index}
    end for
else if strategy is "drop.columns" then
    nullColumnIndices ← [ ]
    for column in IR.headers do
        if NULLPERCENTAGE(column) $\geq 0.5$ then
            IR ← IR \ {column}
        end if
    end for
end if
return IR
```

This node has been designed to detect missing values as either empty cells or cells marked with the implementation language's null representation, for example None in Python. Although this approach may naively overlook some missing values if they are not marked with any of the expected options, and given that the goal is to distinguish between missing and non-missing values to preserve meaningful information, the assumption previously made on what constitutes a missing value according to FlowETL is deemed reasonable.

In detail, the strategies available to the MVH node are :

- impute - replace missing values with the appropriate placeholder for the column's data type.
- drop.rows - drop all rows containing at least one missing value
- drop.columns - drop all columns which are 50%+ composed of missing values

Duplicate Rows Handling

The DRH node is responsible for identifying and removing duplicate rows, which are generally redundant for most data analysis tasks [13]. This node does not require a strategy parameter, since all duplicated rows are inherently dropped.

In this context, rows are considered as singular objects, meaning that rows are equal if and only if they contain the same values for the same columns, disregarding column ordering to account for possible shuffling during serialisation and deserialisation of the IR. This approach is inspired by the Duplicate Detection (DuDe) Toolkit [11], which according to its authors, "*supports the search for tuples that represent the same real-world object in a variety of data sources*".

The duplicate checking algorithm 4 operates by using a data structure with constant lookup time, such as a hash table [46]. Each row is hashed into a string representation, which becomes the key in the map. If two rows, A and B, are identical, their hashes will collide, signalling that B is a duplicate of A. Once the algorithm terminates, the map's keys represent the unique rows of the IR, and de-hashing these them restores the IR without duplicate rows.

Algorithm 4 Duplicate Values Handler

```
Input: IR
Output: IR
uniqueRows ← [ ]
seen ← { }
for all row in IR do
    key ← SERIALISE(row)
    if key $\notin$ seen then
        uniqueRows ← uniqueRows ∪ {row}
        seen ← seen ∪ {key}
    end if
end for
IR ← DESERIALISE(uniqueRows)
return unique_rows
```

An alternative approach considered involves hashing the rows, sorting them in $O(nlogn)$ time, where n is the number of non-header rows in the IR, and then performing a linear scan to remove duplicates, which will now be grouped together. Afterward, a second linear scan is needed to de-hash the data. However, for simplicity and a slight runtime advantage, the former approach was chosen, since it requires less time due to the absence of the sorting step but additional memory due to the use of a lookup table.

Numerical Outliers Handling

The NOH node was specifically developed to address outliers in numerical columns. This design choice was made to offer a foundational yet effective approach to data wrangling. Consequently, the pipeline currently does not account for outliers in non-numerical columns, such as string-based outliers (e.g., inconsistent date formats) or boolean outliers (e.g., variation in truth value symbols).

Furthermore, a choice between using learning-based methods [6] and statistical methods [47] was made. The Median Absolute Deviation (MAD), one of the mentioned statistical methods, was chosen due to the fact that the median is more robust against outliers than the mean [52]. However, MAD can also be affected by outliers, especially when the proportion of outliers exceeds 50%, causing the median to lie within the outliers.

The MAD score for a particular point is calculated as:

$$T_{\min} = \text{median}(X) - a \cdot \text{MAD}$$

$$T_{\max} = \text{median}(X) + a \cdot \text{MAD}$$

$$\text{MAD} = b \cdot \text{median}\left(|X - \text{median}(X)|\right)$$

where median and MAD are the corresponding statistics of the outlier scores, the values of a

and b are suggested to be 1.48 and 3.0 respectively by Singh and Upadhyaya [44], and X represents the n original numerical values. Here, $T_{\min}$ and $T_{\max}$ establish a boundary for normal data points, and observations outside this range are considered to be significantly different from the majority of the data, making them outliers. The algorithm for outlier detection and handling is outlined below 5.

Algorithm 5 Outlier Handler

```
Input: IR, strategy
Output: IR
for all column in IR.headers do
    if column.type is "number" then
        column ← IR.values[column.index]
        median ← MEDIAN(column)
        mad ← MEDIAN(ABS(column − median))
        outlierMask ← COMPUTEMASK(column, median, 3.0 × mad)
        outlierIndices ← INDICESWHERETRUE(outlierMask)
        if strategy is "impute" then
            for all i in outlierIndices do
                IR[i + 1][column.index] ← median
            end for
        else if strategy is "drop" then
            IR ← IR.headers ∪ rows ∉ outlierIndices
        end if
    end if
end for
return processed_IR
```

In summary, the NOH node first determines the data type of each column. If the column is identified as having type of `number` an outlier mask of the column is generated by applying statistical methods to detect anomalous values. Flagged outliers are then handled through by either imputing the outlier values with the median of the column, or removing the entire row containing the outlier, depending on the value of the `strategy` parameter.

3.5.2 Schema Inference

The schema inference step is at the core of the entire planning process, given that most operations are applied differently according to the type of the column or cell acted on.

Two solutions to the schema inference problem were considered. The first one leverages Apache Spark's `inferSchema` [6] keyword attribute within the `read()` method, while the second approach involved defining a custom schema inference method. The latter was chosen for its flexibility in customising inference logic to suit FlowETL's internal data types.

The `infer_schema` algorithm 6 was designed with two assumptions in mind. First, any cell that can be converted to a floating-point number can be treated as a numeric value. This caused issues for boolean-like columns containing 1s and 0s, which were incorrectly classified as numbers. To counter this issue, a second assumption was made. Namely, if a column contains exactly two values, then it likely encodes a boolean statement. This covers all possible uses of

[6] https://spark.apache.org/docs/3.5.4/sql-data-sources-load-save-functions.html

truth symbols such as "Y,N,true,false, . . . ", however it poses a risk of mislabelling if the column only contains two values due to sampling.

The schema inference algorithm 6 begins by creating a local copy of each column in the input IR. It then iterates over all non-missing values—ignoring missing cells, as they are assumed to provide no relevant information on their column's type. For each value, the inferred type and the cell's hashed value are collected, with the former being determined via simple language-specific checks in the `infer_cell_type` 7 algorithm.

Algorithm 6 Infer Schema

```
Input: IR
Output: schema
schema ← {}
for column in IR.headers do
    typeCounts ← {}
    valueCounts ← {}
    for all cell in column do
        if cell is null or empty then
            continue
        end if
        cellType ← INFERCELLTYPE(cell)
        typeCounts[cellType] ← typeCounts[cellType] + 1
        key ← SERIALISE(cell)
        valueCounts[key] ← valueCounts[key] + 1
    end for
    if valueCounts.length == 2 then
        inferredType ← "boolean"
    else
        recordedTypes ← typeCounts.keys
        if recordedTypes.length > 1 then
            inferredType ← "ambiguous"
        else
            inferredType ← recordedTypes[0]
        end if
    end if
    schema[column.name] ← inferredType
end for
return schema
```

Once the column has been processed, the algorithm uses the frequency of inferred types and the count of distinct values to determine the column's type. For instance, if multiple types are detected (e.g., `[number, number, complex]`), the column is considered ambiguous. If exactly two distinct values are found, it is likely a boolean column, as per the earlier assumption. This process is repeated for every column to generate the final output schema for the IR.

3.5.3 Schema Matching

The main requirements identified for any schema matching solution is that it should support both one-to-one, many-to-one, and one-to-many matches, while leveraging syntactic and semantic similarities between columns in the source and target schemas.

Algorithm 7 Infer Cell Type

```
Input: cell
Output: type
if ISINSTANCE(cell, list) or ISINSTANCE(cell, dictionary) then
    return "complex"
end if
if FLOAT(cell) equals cell then
    return "number"
else
    return "string"
end if
```

Additionally, this step relies solely on column names and types, disregarding the actual values within the columns. This choice comes from the possibility that values, types, or structures of cells in potentially matching columns could be altered by the transformation logic, meaning that the schema matching step should remain agnostic to any chances to a column's values. As a result, this step is performed beforehand, and the resulting IR is prepared to support the LLM inference process discussed later. Three different approaches were evaluated as a possible solution the schema-matching step.

The first solution considered, SMUTF (Schema Matching Using Generative Tags and Hybrid Features), is an open-source tool that leverages the XGBoost algorithm and sentence transformers[7]. While it demonstrated promising performance, averaging an F1 score of 0.77 during evaluation, its training methodology (using only 16 source-target table pairs) raised concerns about its ability to generalise to more complex scenarios. Additionally, the model offered limited control over the internal matching process and lacked support for many-to-one and one-to-one matching scenarios, making it unsuitable given the requirements.

The second approach involved using an incremental, heuristic-driven approach which produces multiple candidate matches between schemas[14]. The authors argue that common tools tend to generate a single mapping between elements of two schemas, assuming it to be the optimal solution. While such tools offer efficient outputs, they often lack flexibility in the matching process and depend on the assumption that their underlying models are well-trained and evaluated on extensive datasets. In contrast, the top-k schema matching approach formulates the problem as a variant of the bipartite graph matching problem[8]. It first computes all possible edges between elements of the two schemas, and assigns a similarity measure between 0 and 1 to each edge.

The algorithm then identifies the top-k mappings through an iterative process, where in each iteration the edge with the lowest score from the current mapping is discarded and replaced by the next best-scoring edge not present in any previous mapping. A heuristic known as Stability Analysis is then applied to promote edges that occur frequently across the top-k mappings and demote less frequent ones. This refined ranking is used to recompute the final top-1 mapping. While this approach supports more advanced matching operations such as replication (one-to-many mappings) it lacks support for decomposition (many-to-one mappings) of attributes, limiting its suitability for the task.

[7] `https://github.com/fireindark707/Python-Schema-Matching`

[8] `https://discrete.openmathbooks.org/dmoi3/sec_matchings.html`

The third approach considered follows the the Gale-Shapely algorithm [9], a concrete implementation of the Stable Matching Problem described in the Chapter 2. One limitation identified with this algorithm is its restriction to one-to-one matches, making it unsuitable for more complex matching scenarios such as replication and decomposition, as seen in [14]. To address this, a modified version of the algorithm was designed to support $n:1$ and $1:n$ matches, $n \in [0,2]$.

At a high level, the algorithm works as follows. Firstly, the sets of source attributes X and target attributes Y are extracted from the source and target IR ingested by the Planning Engine. A bipartite graph is constructed such that for every node x in X, a directed edge (x,y) is created for all nodes y in Y, assigning the weight as a score between 0 and 1. The second step of the algorithm iteratively improves upon a starting mapping between X and Y until a stable mapping is reached, or a timeout reached in instances where a stable match is non-reachable.

Computing the Similarity Graph

The weight heuristic is designed to assign a value between 0 and 1 to each edge in the graph, where a score of 1 denotes perfect similarity and 0 indicates complete dissimilarity. This heuristic accounts for both syntactic and semantic similarities between column names. Syntactic similarity is measured using the Levenshtein edit distance, which quantifies the minimum number of single-character edits required to transform one string into another [10]. Semantic similarity is computed by treating each attribute as a sentence and generating dense vector representations that capture their semantic content. The cosine similarity between the source and target vectors is then calculated[10].

The final edge weight is obtained by averaging the syntactic and semantic similarity scores, followed by normalisation to ensure the result falls within the $[0,1]$ range. Edges with weights below a threshold of 0.5 are discarded, under the assumption that such low-similarity mappings are not significant. In alignment with the original Gale-Shapley algorithm, each node on the source side of the bipartite graph sorts its incident edges in descending order of weight. The pseudocode responsible for generating the similarity graph is outlined below 8.

The Schema Matching Algorithm

Once the bipartite graph is constructed, it is fed into a modified Gale-Shapley algorithm designed to support varying matching requirements, which operates by iteratively attempting to establish a stable match between nodes in sets X and Y through a sequence of proposals and rejections, both tracked by the algorithm 9.

For each unmatched $x \in X$, a proposal is made to x's highest-ranked y node which has not rejected x in previous iterations. If y is already matched, a challenge process is initiated, where x competes with the node, x', currently matched to y. Possible outcomes include (1) y matching with both x and x', implying a potential joining of source columns, (2) replacement of x' by x if the latter has a stronger similarity with y, or (3) rejection of x if x' remains the preferred match. Rejected $x \in X$ nodes update their rejection list and re-enter the proposal cycle if further y candidates remain.

To prevent infinite loops or redundant iterations, the algorithm includes a stalling detection mechanism triggered if the number of unmatched x nodes does not decrease between iterations. If a stall is detected, execution is halted and a failure is returned. An example is shown in Figure 3.5

[9] https://medium.com/data-science/gale-shapley-algorithm-simply-explained-caa344e643c2

[10] https://huggingface.co/tasks/sentence-similarity

Algorithm 8 Compute Similarity Graph

```
Input: X, Y, threshold = 0.5
Output: G
model ← LoadModel()
xEmbeddings ← zip(X, model.encode(X))
yEmbeddings ← zip(Y, model.encode(Y))
G ← {}
for v in Y do
    for u in X do
        semanticSim ← model.similarity(xEmbeddings[u], yEmbeddings[v])
        syntacticSim ← EditDistance(u, v)
        avgSim ← AVG(semanticSimScore, syntacticSimScore)
        if avgSim > threshold then
            G[u].add((v, avgSim))
        end if
    end for
end for
for x in X do
    G[x] ← G[x].sort(descending)
end for
return G
```

```
{
  "X.header3" : "Y.header2",
  "X.header1,X.header2" : "Y.header1",
  "()" : "Y.header3",
  "X.header4" : ()
}
```

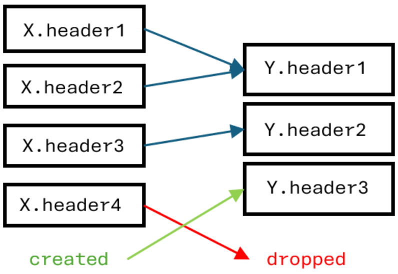


Figure 3.5: Example output schema map and corresponding diagram

The time complexity of this algorithm is comparable to that of the original Gale-Shapley algorithm. While the original implementation has a time complexity of $O(n^2)$ for sets of equal size n, the modified version, adapted to handle source and target sets of potentially differing sizes, has an estimated time complexity of $O(\max(n,m)^2)$, where n and m denote the sizes of the source and target sets, respectively. This schema matching approach has several limitations. Firstly, it operates solely on schema metadata, without accounting instance-level data. As a result, potentially valuable information embedded in the actual data values is not utilised. Additionally, the algorithm only supports flat schema structures and does not handle nested or hierarchical representations, such as those found in semi-structured files like JSON, where values may themselves be objects or sub-schemas. These nested structures are currently abstracted away and excluded from the matching process, however solutions to this limitation are discussed in Chapter 6.

A fourth solution evaluated aims to leverage LLM inference to compute a mapping between source and target schemas. An in-depth outline of the prompt architecture Figure 5.2 and a comparison with the modified Gale-Shapley schema matching algorithm are provided in Chapter 5.

Algorithm 9 Schema-Matching Algorithm

```
Input: source, target, diff
Output: matchedAttributes
X, Y $\leftarrow$ COMPUTEGRAPH(source, target)
unmatchedX $\leftarrow$ [] for x in X
matchedAttributes $\leftarrow$ [] for y in Y
rejectedMatches $\leftarrow$ []
while unmatchedX $\neq$ [] do
    x $\leftarrow$ unmatchedX[0]
    for y in X[x] do
        if y.name $\notin$ rejectedMatches[x] then
            if matchedAttributes[y].length == 0 then
                assign x to matchedAttributes[y]
                remove x from unmatchedX
                break
            else if matchedAttributes[y].length == 1 then
                defender $\leftarrow$ matchedAttributes[y]
                if ISWITHIN(x.score, defender.score, diff) then
                    add x to matchedAttributes[y.name]
                    remove x from unmatchedX
                    break
                else if x.score > defender.score then
                    matchedAttributes[y.name] $\leftarrow$ x
                    add defender to unmatchedX
                    add y.name to rejectedMatches[defender]
                    remove x from unmatchedX
                    break
                else
                    add y.name to rejectedMatches[x]
                    break
                end if
            else
                defender1, defender2 $\leftarrow$ matchedAttributes[y.name]
                x.score $\leftarrow$ score from Y[y.name] for x
                if defender1.score + defender2.score < x.score then
                    matchedAttributes[y.name] $\leftarrow$ x
                    add defender1, defender2 to unmatchedX
                    add y.name to rejectedMatches[defender1]
                    add y.name to rejectedMatches[defender2]
                    remove x from unmatchedX
                    break
                else
                    add y.name to rejectedMatches[x]
                    break
                end if
            end if
        end if
    end for
end while
return matchedAttributes
```

3.5.4 Planning Computation and Evaluation

This step involves exhaustively evaluating all possible transformation plans and selecting the one that maximises the DQS evaluation metric, using the artifacts available to the Planning Engine.

Each Airflow task node consumes an IR as input and produces a modified IR as output. This modular behaviour enables plan construction by varying the sequence in which nodes are composed. The current implementation adopts a brute-force approach, evaluating all feasible node orderings to identify the optimal plan. Although computationally inefficient, this method ensures comprehensive plan exploration. Currently, the system supports three distinct nodes and a total of six node-strategy combinations. Given that one node-strategy pair is selected from each of the three available groups, the total number of possible plans is 36.

The engine subsequently generates all possible plans and applies each to a copy of the sampled source IR. For each valid (non-failing) plan, it computes the DQS for the resulting IR using Algorithm 10. The engine maintains a mapping of the form `{ plan : achieved DQS }`. Once all candidate plans have been evaluated, the plan yielding the highest data quality score is selected and returned. To optimise the brute-force approach, this step is designed to terminate as soon as a sufficiently good plan is found, with a DQS greater than 0.95. The idea is that once a satisfactory solution is identified, the computational and time costs of searching for a better plan are not justified by the potential gains in data quality.

If no viable plan can be found for the sampled IR, a default plan is returned, specifically `"impute missing values → remove duplicate rows → impute outliers"`. While this may not be the optimal plan for all datasets, it is a reliable, non-failing option that ensures no errors or bottlenecks occur further down the pipeline. However, certain plans will inevitably fail. For example, plans that address outliers before handling missing values are prone to failure, as the outlier node isn't designed to handle null values. This design follows the principle of separating concerns. Further improvements to this plan generation strategy are discussed in Chapter 5.

3.5.5 LLM Inference of Transformation Instructions

The schema matching stage precedes the transformation logic inference component of the Planning Engine. Given one or two source columns x_1 and x_2 and a corresponding target column y, this component aims to infer a transformation function $f(x_1,x_2) \rightarrow y$ which combines the inputs to obtain the output column. A key constraint in this process is the exclusion of any external auxiliary data, such as dictionaries, databases, or knowledge bases. As a result, the transformation logic should be inferred solely from the available schema mappings, the structure of the source and target tables, and values within the involved columns.

The following options were considered:

1. Using the TBP approach, based on the work introduced in Auto-Transform [23], involves inferring regular expression-like patterns from column values and referencing a repository of previously validated pattern transformations to infer the desired mapping. Although a promising approach, constructing a repository of pattern made the TBP method unfeasible.

2. Using a TBE approach, similar to that used in Foofah [22], which synthesises transformation programs from paired input-output tables. Similar to TBP, this approach seemed promising

Algorithm 10 Compute Data Quality

```
Input: IR, schema
Output: dataQuality
cellsCount ← IR.headers.length × IR.length
missingCount ← COUNTMISSINGVALUES(IR)
missingRatio ← missingCount / cellsCount
outliersCount ← COUNTNUMERICALOUTLIERS(IR, schema)
numericalValsCount ← COUNTNUMERICALVALUES(IR, schema)
outliersRatio ← outliersCount / numericalValuesCount
duplicateRowsCount ← IR.length-1 − COUNTUNIQUEROWS(IR)
duplicateRatio ← duplicateRowsCount / rowCount
dataQuality ← AVG(missingRatio + duplicateRatio + outliersRatio)
return dataQuality
```

due to the fact its inputs are comparable to the ones used in FlowETL. However, its capabilities are limited, as it only supports table restructuring and lacks the ability to infer or apply transformations to the data itself.

3. A third approach is inspired by the work of Raman and Hellerstein [41], in which an informed search through a graph of intermediate states is proposed, where each state represents a cell value resulting from an applied operation. In the solution considered for FlowETL, each column type defines a set of permissible actions, for example, strings support insertions and deletions, while numeric columns support mathematical operations. The search aims to apply these transformations and evaluate how close each intermediate state is to the target, similar to the A* search algorithm[11]. However, due to the vast size of the state space, this approach proved impractical.

4. The final approach explored the use of an LLM fine-tuned for code generation. In this method, the available artifacts were provided as part of a structured prompt, guiding the LLM to generate an output following the construct of $f(x_1,x_2) \rightarrow y$. This output was then compiled into a function, which could be applied to the source IR to transform it into the target one.

The fourth approach was selected due to the powerful capabilities and recent advancements of LLMs on the task of code logic generation [51]. The prompt was structured as shown in Figure 3.6, following the practice of zero-shot learning, where a model performs a task without having been explicitly trained on similar examples and instead using its generalisation capabilities and data encountered during training. This prompting approach, also referred to as *task instructions* [27], has been selected due to their ease of understanding and straightforward design process.

3.5.6 Payload Construction

The exit point of the Planning Engine is the publishing step, where both the computed transformation plan and the engine runtime metrics are sent to the Messaging Broker. The detailed payload is outlined in Figure 3.7.

[11] https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

```
Task description : "You are given two tables represented as 2D lists:
    an input table and an output table. Your task is to write a Python
    function that transforms the input table into the output table"

Your function must:
- Perform data transformations to match the output table
- Use the provided mapping and schemas to guide your logic.
- Leave any cell with the value ’_ext_’ unchanged.
- Convert numeric strings to float before using them.
- If a cell is ‘A|B‘, you may split on ‘|‘ and use both values.
- Assume column renaming and reordering is already done.
- Be named ‘transform_table‘.

Column mapping rules (context): [...]

Constraints :
- Return only valid, executable Python code - no comments
- Your response will be evaluated by ‘exec()‘
- Your logic should generalise to similar tables. Do not hardcode, do
    not provide samples, and do not randomly generate cell values.

Operate on the following artifacts:
    Input Table: [...]
    Output Table: [...]
    Input Schema: {...}
    Output Schema: {...}
    Column Mapping: {...}
```

Figure 3.6: LLM prompt structure for inference of transformation logic

A key assumption underpinning this approach is that if the Planning Engine identifies an optimal plan based on a representative sample of the source file, then this plan is expected to generalise effectively to the entire source dataset.

The plan is structure as follows :

- `Source File` - name of the file targeted by the plan.
- `Reconstruction Key` - an optional key used to convert the IR back into JSON format.
- `Schema Map` - a source-target columns mapping as defined by the target file.
- `Plan Steps` - a sequence of DTNs designed to maximise the DQS on the sample IR.
- `Logic`: A string containing valid executable instructions generated by the LLM, which is compiled into a Python function and invoked onto the IR by the ETL worker.
- `IR Schema`: The schema inferred from the sample IR, stored to prevent redundant recomputation within the ETL worker.

A key observation is the importance of maintaining the plan structure outlined below consistently throughout the pipeline, as components such as the ETL worker are specifically designed to interface with it. Any modifications to the plan structure should therefore be carefully managed to ensure compatibility.

3.6 ETL Worker Pipeline

The ETL Worker component is responsible for ingesting the optimal transformation plan previously computed by the Planning Engine and applying it to the entire source file.

```
{
    "plan": [
        "sourceFileName.ext"
        {"associatedKey": key},
        {"schemaMap": (...)},
        planStep1,
        ...,
        planStepN,
        {"logic": "..."}
    ],
    "IRschema": {...}
}
```

```
{
    "contents": {
        "bestPlan": [...],
        "maxDQAchieved": 0.999,
        "plansComputed": 24,
        "plansFailed": 0
    },
    "from": "planningEngine"
}
```

Figure 3.7: Planning engine payload to the Messaging System. The optimal plan is shown on the left, while the runtime metrics are on the right.

This component has been designed to follow the standard ETL practices, which divide the process into the three core steps of (1) extracting the source file's contents, (2) transforming the raw data into the desired form, and (3) loading the transformed data to the intended location, which is often a database or a data warehouse[50].

It has additionally been designed, like other components, to operate as long as the Messaging System is alive. In particular, the pipeline continuously polls the `optimalPlans` topic until a plan is detected and the ETL process triggered. The ETL process is as follows :

1. Extract the file name and transformation instructions from the published plan.
2. Translate the entire source file into an IR
3. Conduct a preliminary pre-ETL data quality assessment on the IR, using the method `compute_data_quality()`. See Algorithm 10 for details.
4. Parse and apply the transformation instructions
5. Conduct a post-ETL data quality assessment on the transformed IR
6. Translate the transformed IR back to original file type and load to loading destination.

The design decision of separating the plan generation step from the application step enables for many ETL workers to be instantiated. In essence, the plan can be computed once by the Planning Engine and distributed to as many ETL Worker instances as required, depending on the size the source file to be processed. Again, the architecture's ability to scale on demand is one of the core requirements of the ideal pipeline architecture outlined in the literature[29].

3.7 Reporting Engine

The Reporting Engine provides an interface for the user to track the progress of each component in the pipeline. It works by continuously polling the Messaging System's `etlMetrics` topic and populating itself with the payload returned by each component, making it a self-populating, self-managed report artifact.

The class is equipped with the following attributes and methods, shown in Figure 3.8. Attributes are dictionaries which store the metrics returned by the component which collected them. The `is_complete()` method returns `true` if the report is full, which should only occur after the pipeline has run from start to finish. The `run()` method acts as the driver, and the `compile()` method simply formats the attributes into a tabular, formatted string. Lastly, the `save()` method writes the report to the appropriate location, ready to be consumed by the user.

By structuring the reporting logic in this way, the output is guaranteed to remain consistent, meaning that users can extend the capabilities of the system to do further analysis on the compiled report.

For example, key metrics could be extracted and used for downstream tasks such as anomaly detection within the pipeline's behaviour. Such a feature has been highlighted extensively in the background literature as being critical to the monitoring, maintenance, and improving the pipeline's performance [39, 1, 32].

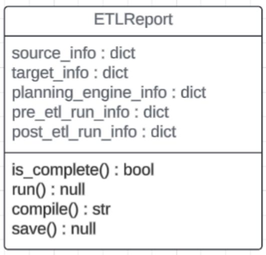


Figure 3.8: UML class diagram of the Reporting Engine's underlying Report artifact.

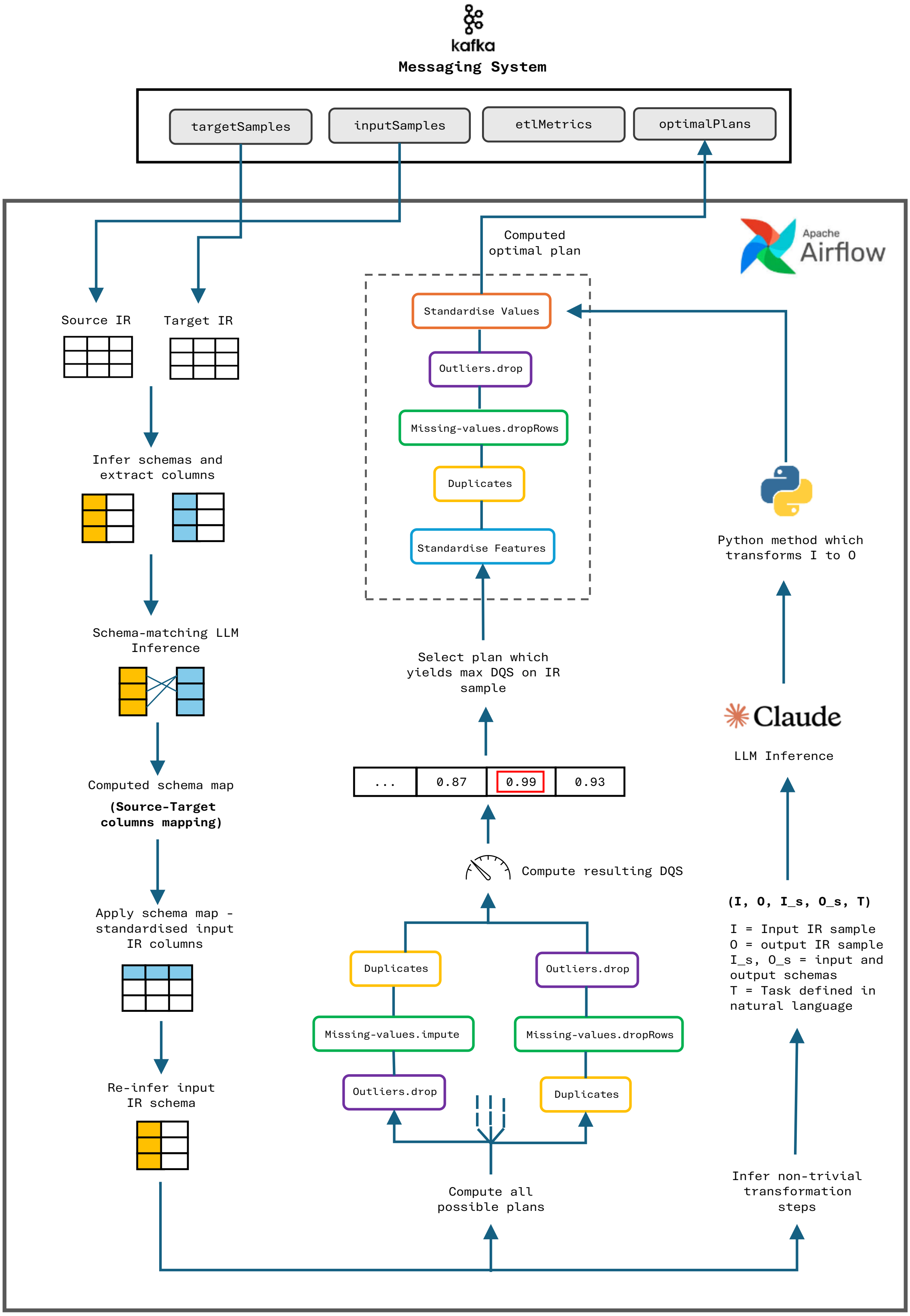


Figure 3.9: Overview of the planning engine and data flow from consumption to plan generation and publication.

Chapter 4

Implementation

This chapter gives an in-depth outline of the implementation of this project's design, in addition to providing an overview of the solutions considered and the factors which influenced the implementation process. The chapter is divided into subsections corresponding to the individual components of the architecture.

4.1 Infrastructure Setup with Docker

Apache Kafka was chosen as the implementation of the Publisher/Subscriber model introduced in Chapter 3.

Two options were considered when setting up a local Kafka broker. The first involved installing Apache Kafka directly on the development machine, while the second leveraged a pre-built Docker image of Kafka[1].

Initially, the native installation approach appeared appealing due to Kafka's compatibility with Windows, the operating system underlying the development environment. Although functional, this setup proved cumbersome, requiring manual startup of the broker and recreation of topics with each application launch. This repetitive overhead significantly hindered the development workflow.

The second approach involved installing Docker and retrieving the Kafka image from Docker Hub[2], a central repository for containerised application images, enabling the creation of a Kafka broker as a Docker container. Containers package all necessary components—application code, runtime, libraries, and dependencies—ensuring consistent execution across different environments.

In a similar fashion, an Apache Airflow instance [3] was setup. While local installation was initially considered, Airflow is primarily designed for Linux-based environments. To facilitate setup on a Windows system, the Windows Subsystem for Linux (WSL)[4] was employed as a compatibility layer. However, this approach involved a complex and time-consuming configuration process, and was discarded as a consequence.

A custom `docker-compose.yml` file was written to define the infrastructure orchestration for the development environment. This configuration allowed for precise control over services initialisation and tear-down, including defining service dependencies (e.g., ensuring Airflow started

[1] https://hub.docker.com/r/apache/kafka

[2] https://hub.docker.com/

[3] https://hub.docker.com/r/apache/airflow

[4] https://ubuntu.com/desktop/wsl

after Kafka) and pre-configuring Kafka topics. Additionally, the compose file facilitated the installation of required Python packages within the Airflow container and the specification of environment variables such as Application Programming Interface (API) keys for LLM inference, and login credentials for the Airflow web interface.

A noted drawback of this approach was the need to rebuild the entire infrastructure upon any modification to the compose file. This process typically took around 10 minutes, presenting a development bottleneck. However, given the rarity of these changes and the improvements to the services orchestration process brought along by this approach, this was deemed an acceptable trade-off.

Another development bottleneck encountered was setting up a connection between the components running within the Docker Engine-the Kafka broker and Airflow instance-and the components running locally, such as the Observers, the ETL worker, and the Reporting Engine. A connection was setup leveraging Docker Networks[5], where container ports were mapped to localhost ports, allowing seamless communication between containerised services and components running locally.

4.2 Observers

The Python programming language was chosen to implement the Observers, due to its simplicity and adoption in tools such as Apache Airflow and Apache Kafka. Although Java was also evaluated as a potential implementation language, the lack of native support in Airflow was identified as a significant limitation, therefore making it unsuitable for this project.

The observers leverage the `watchdog` package [6], which allows for configurable monitoring of the file system. This package comes with a `FileSystemHandler` class, which detects and reports any events within the folder being monitored.

The Observers operate in an event-driven manner, monitoring and filtering events to identify those specifically indicating a file upload. Upon detection of such events, the translation to IR and publishing logic begins. Given that both Observers share common properties and behaviours, the implementation leverages the Object-Oriented Programming (OOP) principle of Inheritance[7]. This implementation choice enhances code reusability and improves testability the Observers. A `Base Observer` superclass was developed, from which the Source and Target Observers stem from.

The communication between the Observers and the Kafka broker was setup using the `kafka-python` [8] package, a Python API to publish from and subscribe to Kafka topics. See Figure 4.1 for an example.

An instance of a `KafkaProducer`, configured to publish to one of the pre-defined topics, was embedded within each Observer class through Composition, another OOP paradigm [9], allowing the Observers to send payload without having to instantiate the `KafkaProducer` class themselves.

Managing events such as insertions and deletions to files already uploaded within the watched

[5] https://docs.docker.com/engine/network/

[6] https://pypi.org/project/watchdog/

[7] https://www.codecademy.com/resources/blog/what-is-inheritance/

[8] https://kafka-python.readthedocs.io/en/master/index.html

[9] https://realpython.com/inheritance-composition-python/

```
producer = KafkaProducer(
    bootstrap_servers=’localhost:9092’,
    key_serializer=lambda k: k.encode(’utf-8’))
    value_serializer=lambda v: json.dumps(v).encode(’utf-8’)
)
producer.send(
    topic=’optimalPlans’,
    key=payload_key,
    value=payload_value
)

consumer = KafkaConsumer(
    ’inputSamples’,
    bootstra_servers=[’kafka:9091’]
)
consumer.poll()
```

Figure 4.1: Code snippet for instantiating a Kafka producer and consumer through the `kafka-python` package, as well publishing to a topic using the `send()` method and consuming from a topic using the `poll()` method.

directory was a persistent issue throughout the implementation stage. A temporary solution consisted of implementing a lookup table which stored a file’s name and the number of objects contained within. Each observer would then check if the file was extended, and only then the extra objects would be published for processing. This mechanism was discarded for simplicity purposes.

Another implementation detail is within the `extract_sample()` method, used by the Source Observer to extract a subset of objects from possibly large source files. Originally, a sampling percentage parameter `p` was implemented with a default value of 25%. This posed an issue due to the variable size of the payload published to the `inputSamples` topic. A viable mitigation was setting `p = 0.05` (i.e., 5%) with an upper-bound on the sample size of 50 objects, meaning that extremely large files did not result in unmanageable payload sizes.

Each Observer is executed in its own dedicated process, implemented using Python’s `threading` library[10]. While Python’s Global Interpreter Lock (GIL) [11] prevents true parallel execution of threads within a single process, this approach enabled concurrent task scheduling and responsiveness, effectively simulating parallelism for Input Output (IO)-bound operations.

4.3 Planning Engine

Apache Airflow was selected as the framework for implementing the Planning Engine due to its inherent support for sequential task execution, which aligns with the requirements outlined in the previous chapter. Furthermore, Airflow provides built-in mechanisms for failure handling, a user-friendly web interface, and an internal messaging system that facilitates communication between DTNs, making it a robust and suitable solution for this component.

4.3.1 Airflow DAG and Task Nodes

Each node in an Airflow DAG is referred to as a *task node*, which may depend on an upstream task or have dependants of its own, as shown in Figure 4.2. Each task node in Airflow can additionally execute different types of command, such as bash commands using the Airflow’s `BashOperator`,

[10] `https://docs.python.org/3/library/threading.html`

[11] `https://wiki.python.org/moin/GlobalInterpreterLock`

or Python code through the `PythonOperator`, used extensively across the Planning Engine.

Airflow task nodes communicate using its internal messaging mechanism, XComs, which enabled the exchange of data via key-value pairs. Additionally, XComs supports a maximum payload size of 48 KB[12], making it a convenient tool for inter-task data sharing. However, this limitation also presented a constraint during development, as any payload transmitted through XComs had to remain within this threshold.

An issue encountered during development arose after the `applySchemaMap` step, which frequently resulted in modifications to the structure of the IR, causing subsequent nodes to fail. To address this, an additional schema inference step was introduced to ensure that the schema artifact reflected any transformations applied to the IR throughout the preceding DAG steps.

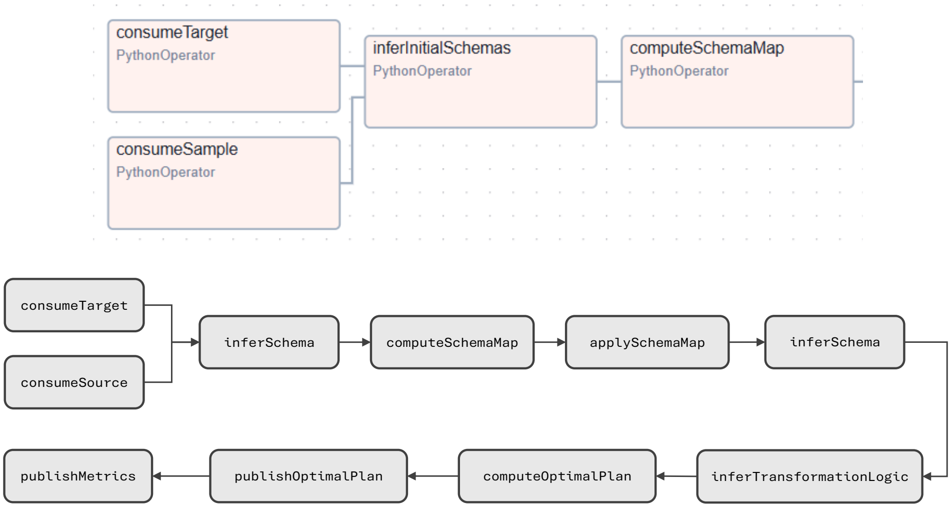


Figure 4.2: Screenshot of a subset of nodes within the Airflow DAG workflow, executing from left to right (top) and diagram of the entire Airflow DAG (bottom)

4.3.2 Schema Matching

To implement the schema matching step, composed of the `computeSchemaMap` and `applySchemaMap` nodes, two distinct approaches were developed and compared. The first approach relies on algorithmic schema matching and combined several Python modules installed via pip[13]. The second approach leveraged LLMs to achieve the same objective, offering an alternative strategy that is explored in greater detail in Chapter 5.

The first stage in the algorithmic process involves constructing the similarity graph between the source and target columns. For this, semantic similarity scores are computed using the `SentenceTransformer` class from the `sentence-transformers` module[14], while syntactic similarity scores are computed using the ratio function from the `FuzzyWuzzy` library[15], which calculates the Levenshtein edit distance between two strings. Both similarity metrics return a numerical value between 0 and 1, and their average is used as the weight of the edge connecting a source column x to a target column y.

[12] `https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/xcoms.html`
[13] `https://pypi.org/project/pip/`
[14] `https://pypi.org/project/sentence-transformers/`
[15] `https://pypi.org/project/fuzzywuzzy/`

An example of the logic used to compute these weights is shown in Figure 4.3, where a model with the identifier "all-MiniLM-L6-v2", a pretrained embedding, is instantiated, and the source and target headers are encoded to compute pairwise cosine similarities. One limitation identified is that the notion of similarity can become ambiguous, particularly when comparing individual words without contextual information.

The second stage of the algorithmic process involves feeding the similarity bipartite graph computed in the previous step into an extended implementation of the Gale-Shapley algorithm described in the previous chapter. Once the mapping is established, it is applied to the IR using the `standardise_features()` method.

Other approaches were also considered, such as using the `semantic-text-similarity` package[16], which offers an interface to fine-tuned BERT models for semantic similarity computation. However, this alternative was not adopted due to its requirement for Graphics Processing Unit (GPU) acceleration, which may not be available in all target deployment environments. More powerful models with similar requirements were also considered unsuitable due to the potential performance bottlenecks they could introduce.

```
model = SentenceTransformer("all-MiniLM-L6-v2")
sentence_similiarity = model.similarity(
    model.encode(source), model.encode(target)
)
levenshtein_distance = fuzz.ratio(u, v)/100
weight = (levenshtein_distance + sentence_similiarity)/2
```

Figure 4.3: Pairwise source-target column headers weight calculation using `sentence-transformers` and the Levenshtein edit distance heuristic

4.3.3 LLM Inference

As introduced in Chapter 3, implementing value standardisation required using the source and target IR, as well as additional artifacts such as their respective schemas and schema-map, to infer a Python function which correctly encodes the Ground Truth (GT) for a source file. An LLM solution was chosen, as discussed in Chapter 3. Two key implementation decisions were made: the first regarding the selection of the most suitable LLM for the task, and the second concerning the method for performing inference with the chosen model.

A survey was conducted to select an LLM from a series of promising options chosen from the Big Code Models leader-board on Hugging Face[17]. This leader-board lists the top-performing open-source models tailored for code generation. At the time of implementation, the best model was Alibaba's Qwen2.5-Coder-32B-Instruct[18], which achieved a *HumanEval*[19] score of 83.2 on Python-only datasets. In addition to this, other models were considered, including closed-source options such as OpenAI's GPT-4 and Anthropic's Claude-3.7-Sonnet. Both models have shown strong performance on code generation tasks, with Claude requiring more precise prompts[34].

The three models were then evaluated on a set of predefined code inference tasks, using the same paired source-target sample instances employed in the Evaluation stage.

[16] `https://pypi.org/project/semantic-text-similarity/`

[17] `https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard`

[18] `https://huggingface.co/Qwen/Qwen2.5-Coder-32B-Instruct`

[19] `https://huggingface.co/datasets/openai/openai_humaneval`

The evaluation methodology was as follows: all models were prompted using the same structure outlined in Chapter 3. The correctness of each model's output was then measured using Levenshtein distance between the target dataset and the result obtained by applying the LLM-generated code block to the source dataset.

An average accuracy score across the three models was computed by averaging the individual scores each model achieved on all 13 dataset pairs. In addition to accuracy, other factors such as the level of required human intervention (e.g., bug-fixing the generated code) and the cost of running LLM inference were also assessed. The results of this evaluation are summarised in Table 4.1.

Model ID	**Average Accuracy**	**Operational Costs** $	**Intervention**
gpt-4	0.89	0.05-0.08	medium
Claude-3.7-Sonnet	0.92	0.04-0.06	low
Qwen2.5-Coder-32B	0.85	0.03-0.05	medium

Table 4.1: Results of comparison between LLMs on the task of code generation based on a paired source-target internal representation

It was acknowledged that the results of the experiment were non-definitive, and that there remains a risk of suboptimal model selection, meaning that a more comprehensive comparison-perhaps including more models-would be necessary to increase confidence in the choice of model.

Based on the initial survey, Claude-3.7-Sonnet and Qwen were selected for further extrinsic evaluation, which involved integrating each model into the broader system and assessing how well the pipeline generalized to unseen datasets.

The first approach explored involved downloading Qwen locally and running inference. However, due to the model's large size (approximately 10GB), extensive download time, and the need for significant computational resources—particularly GPU acceleration—this option was deemed impractical. An alternative method considered was using the Hugging Face Inference API, though this introduced financial costs, as detailed in Table 4.1.

Ultimately, Claude-3.7-Sonnet was selected for integration, as its higher accuracy, reduced need for manual intervention, and improved generalisation potential outweighed the operational costs of interfacing with the Anthropic API and the Hugging Face API. Additionally, using an API-based approach eliminated the need for local hosting, streamlining the development process.

4.4 ETL Worker and Reporting Engine

To establish a dependency between the Messaging System's lifetime and the ETL worker, a subroutine was implemented to continuously poll the broker for incoming plan payloads. While this approach is more resource-intensive, it effectively achieves the desired behaviour. An alternative could be the use of a scheduling mechanism.

The ETL worker was initially designed to be implemented using Apache Spark, allowing to leverage its distributed processing capabilities. However, due to significant overhead and complexity encountered during setup, the implementation was migrated to a Python-based solution. While this approach lacks native support for distributed computation, it streamlines the implementation process and sufficiently meets the functional requirements outlined in the previous chapter.

The Reporting Engine was also developed with Python. As illustrated in Figure 4.1, an instance of the KafkaConsumer class was integrated into the Reporting Engine via composition. This design enables the engine to consume incoming metrics payloads published by other components during the execution of FlowETL.

In addition, a comprehensive logging system was implemented throughout the project to enhance observability and facilitate debugging. Each logger—built using Python's logging package[20]—adheres to a standardised message format. Specifically, each log entry includes: (1) the logger's identifier and severity level (INFO, WARNING, ERROR); (2) a timestamp to facilitate runtime tracing; and (3) the message body detailing the logged event. Additionally, each component within the pipeline writes the logged events to a time-stamped file, storing relevant log reports together for a more organised and manageable architecture.

4.5 Testing

Testing the pipeline was approached from two prospectives. At a more granular level, unit testing methodologies were employed to verify the operational integrity of individual components, while a more holistic evaluation, carried out in the next chapter, concerns the generalisation capabilities of the pipeline, and ensuring that no runtime or logical errors occur during its execution. In order for tests results to be satisfactory, they were required to be correct, replicable, and consistent. These criteria were crucial in ensuring that dependency of each component on upstream components working correctly was fulfilled.

Unit tests focused on each DTN, with their respective strategy, and the infer_schema() method, while components involving the LLMs were tested holistically alongside the entire pipeline. For each DTN, a dedicated unit test was designed. An input IR was supplied and an assertion used to ensure that the DTN with the configured strategy, modified the IR as required. To test the infer_schema() method, a similar approach was adopted. A series of python lists, representing the columns of an IR, were supplied to the method, and the output type asserted against the expected type.

An issue arose with the json.dumps() [21] method, since it did not preserve key ordering when serialising and deserialising file contents. As a result, the column order would be randomly shuffled on each execution, making any positional assumptions redundant when working with the JSON files. As a mitigation, a custom assertion 11 was implemented. The assertion works by hashing both IRs into a string representation, sorting them lexicographically, then checking that there is a character-wise match.

Algorithm 11 Custom IR Equality Assertion

```
Input: inputIR, outputIR
Output: None
hashedInputIR ← SORT(HASH(inputIR))
hashedOutputIR ← SORT(HASH(outputIR))
if hashedInputIR $\neq$ hashedOutputIR then
    raise failureException()
end if
```

[20] https://docs.python.org/3/library/logging.html

[21] https://docs.python.org/3/library/json.html

Chapter 5

Evaluation

This chapter outlines the experimental methodology adopted to evaluate the project. In summary, the intrinsic component of the evaluation compares two approaches for schema inference, as well as exploring how altering the sampling percentage within the Source Observer changes the output of the Planning Engine. The extrinsic component looks at comparing FlowETL against another ETL tool with respect to output quality. The result are reported and analysed, and limitations of the experimental design are also discussed.

5.1 Methodology

The evaluation corpus was constructed by collecting 13 datasets (7 in CSV format, 6 in JSON format) from Kaggle.com[1], ensuring diversity with respect to their domain and size (i.e., entries count). This approach was intended to evaluate FlowETL's generalisation capabilities across a varied set of inputs, enabling a comprehensive assessment of both the pipeline's correctness and performance.

For each dataset, a human-defined target was created to capture all schema-matching scenarios (e.g., direct mapping, replication, duplication) and a diverse set of transformations, including merging, formatting, and formula application. Additionally, a GT transformation plan was defined for each dataset, representing the correct sequence of operations required to convert the source into its target form.

The datasets contained an insufficient number of data wrangling issues, due to the fact that Kaggle services pre-cleaned datasets ready for downstream tasks. To account for this shortcoming, a polluter script was developed and executed on each dataset, effectively transforming each dataset to contain around 40% missing values, 20% duplicated rows, and 5-10% numerical outliers. The resulting loss of quality and increased number of data wrangling issues is highlighted in Table 5.1.

To evaluate the Planning Engine, the *PlanEval* metric was designed. The core idea of behind this metric is inspired by another metric, Success Rate for Data Transformation (SRDT) [21] which aims to capture the percentage of correctly generated transformations.

Additionally, the authors of ELT-Bench, the evaluation framework which encompasses SRDT, factored-in the average cost (in terms of token usage and API calls) and the number of execution steps required per task within the results. [26] proposes a new testing framework, Big Bench for Large-Scale Databases (BIRD), which focuses on evaluating how well LLMs can solve Text-to-SQL tasks. Within the BIRD framework, the authors define Valid Efficiency Score (VES)

[1] `https://www.kaggle.com/`

Dataset	Pre M%	Pre D%	Pre O%	Pre DQS	Post M%	Post D%	Post O%	Post DQS
Amazon Stock	0.0	0.0	1.5	1.0	40.2	16.5	2.3	0.8
Chess Games	0.0	2.1	0.7	0.99	40.1	16.7	2.1	0.8
E-commerce Transactions	0.0	0.0	1.5	0.99	41.9	18.6	3.0	0.79
Financial Compliance	0.0	0.0	1.5	0.99	39.3	12.9	1.6	0.82
Netflix Users	0.0	0.0	0.0	1.0	40.2	16.5	4.0	0.8
Pixar Films	0.0	3.6	0.9	0.99	41.0	19.4	1.2	0.8
Smartwatch Readings	1.9	3.6	2.8	0.98	41.2	17.6	3.3	0.79
Amazon Reviews	1.9	0.1	2.5	0.92	40.9	16.2	8.7	0.78
Flight Routes	0.0	0.0	0.4	0.99	40.1	19.1	3.0	0.79
News Categories	2.1	0.5	0.0	0.99	41.3	14.2	0.5	0.82
Recipes	0.0	0.0	0.0	1.0	40.0	29.9	12.1	0.73
Social Media Posts	0.0	1.7	12.2	0.94	44.6	17.6	12.2	0.75
Student Grades	1.8	0.0	0.0	0.94	40.9	16.6	1.0	0.81

Table 5.1: Data wrangling percentage across evaluation dataset before (pre) and after (post) artificial pollution. Abbreviations are M - missing values, D - duplicate rows, O - numerical outliers, DQ - data quality

metric, which optimistically computes the correctness of the generated Structured Query Language (SQL) query against the ground truth. Optimism in this context refers to positively scoring an output if and only if it matches the ground truth perfectly, without penalising any incorrect plan components.

PlanEval currently focuses exclusively on evaluating the SRDT, as the associated LLM token usage and API call costs are kept fixed and minimal through a consistent sampling strategy and a fixed planning engine architecture, limiting LLM invocation to exactly two calls per evaluation. Additionally, constraining the input to a maximum of 50 rows per prompt ensures that token consumption—and thus cost—remains within a predictable and controlled range.

Within this project, A plan $P_f = \{m1, m2, ...mN, t1, t2, ...tM\}$ produced for a particular source file f consists of schema matching steps denoted by m and transformation steps denoted by t, acting at the structural and instance level respectively.

A ground truth plan GT_f consists of the required set of matching and transformation operations required to transform the source file into the target one. The evaluation metric $PlanEval(P_f, GT_f) \rightarrow [0,1]$ rewards correct operations within a plan and ignores incorrect ones, producing a normalised score ranging from 0.0 (completely incorrect plan) to 1.0 (perfect plan).

This metric additionally handles cases where the number of operations in P_f differs from the number of operations in GT_f. In particular, if P_f contains missing operations, these are implicitly scored with a 0.0. Similarly in the case where P_f contains extra operations, which might have been hallucinated by the LLM, they are also scored with 0.0, making *PlanEval* an optimistic scoring metric. In detail, the score for a plan is computed as follows:

- compute the maximum achievable score *maxS* for P_f as $1.0 \times n$ where n is the number of the operations in GT_f
- set the initial score for P_f as $s = 0.0$
- for each operation $op \in P_f$, if $op \in GT_f$ and op is correct, increment s by 1.0, if not correct increment by 0.5, otherwise do not increment.
- The $PlanEval$ score for P_f is computed as $s \div maxS$

5.2 LLMs vs Type Checking for Schema Inference

The objective of this experiment was to evaluate whether the advantages of leveraging a LLM for schema inference outweigh the associated drawbacks. Specifically, the experiment aims to determine if the enhanced generalisation capabilities of LLMs justify the inherent trade-offs, including increased monetary cost and potential variability in output consistency. The methods compared were Planning Engine's internal `infer_schema()` method, which uses a series of conditional checks to determine a column's type, and Claude-3.7-Sonnet as the LLM.

For each dataset in the evaluation corpus, 20 rows were randomly sampled and a schema computed manually, constructing the GT for this experiment. A schema for each sample was then computed using the `infer_schema()` method, and another sample computed by prompting the LLM with the prompt shown in Figure 5.1.

```
"Infer a schema for the following table [...].
The possible types a column can have are :
- ’number’ for numerical columns , both integer and decimal
- ’string’ for strings and characters
- ’boolean’ for columns encoding a truth statement
- ’complex’ for columns whose cells contain list or dict values
- ’ambiguous’ for columns without a distinct type."
```

Figure 5.1: LLM prompt structure for the schema inference experiment

A pessimistic metric was used to score the schema computed for each file. For a given schema $S = \{column : type\}$, the metric carries out a pairwise comparison between the type returned by either of the two methods and the GT. If the type is incorrectly inferred, the column contributes -1 to the total, therefore a perfectly inferred schema scores 0, while a completely wrong schema will score $-n$, where n is the number of columns in the schema.

For the use of an LLM to be justified, the model is expected to achieve perfect schema inference, while the type-checking method should exhibit significantly lower performance. The results are summarized in Table 5.2.

Dataset	Infer_Schema Penalty	LLM Penalty
Amazon Stock	0	0
Chess Games	0	-1
E-commerce Transactions	-1	-1
Financial Compliance	0	0
Netflix Users	0	0
Pixar Films	-1	0
Smartwatch Readings	-1	-1
Amazon Reviews	0	0
Flight Routes	0	0
News Categories	0	0
Recipes	0	0
Social Media Posts	0	0
Student Grades	0	0
Total Penalty	-3	-3

Table 5.2: Total Penalty of each method on the schema inference experiment

The results show that the both methods received penalities of -3 across 13 datasets and 132 columns. The results conclude that LLM schema inference comes with an increased monetary cost and no significant performance gain. A plausible explanation is that using an LLM for such trivial tasks may be redundant. The results of this experiment informed the design process, resulting in type-checking being adopted within the Planning Engine for schema inference.

5.3 Algorithmic vs LLM-based Schema Matching

This experiment aimed to compare two versions of the Planning Engine.

The first version (v1) achieves the schema matching step through the use of a custom implementation of the Gale-Shapely algorithm, extended to support one-to-many and many-to-one matches and described in detail in chapter 3, as well as providing the LLM with a detailed example within the prompt detailing how the transformation logic inference step should be carried out.

The second version (v2), uses the same LLM for both steps, approaching the two tasks separately. The transformation inference prompt differs such that the example has been replaced by a set of rules and restrictions to guide the LLM, as outlined in Anthropic's documentation to prompting their Claude-3.7-Sonnet model [2], while the prompt for the schema matching step is described in Figure 5.2

```
"You are given a list of source column headers and target column
    headers. Your task is to infer a plausible mapping between them.

Rules:
- At most two source attributes can be merged into one target attribute
    : represent this as ‘("s1", "s2"): ("t1")‘
- A source attribute may also be split into two target attributes:
    represent this as ‘("s1"): ("t1", "t2")‘
- If a column is dropped, mark it with an empty tuple: ‘("s1"): ()‘
- If a column is created, use an empty tuple as the key: ‘(): ("t1")‘
- Columns can appear at most once in each mapping.

Additional constraints:
- Do not include placeholder values such as ‘None‘
- Ensure no duplicate keys in the output
- Return a valid Python dictionary using tuple keys and values
- Do not add any comments, explanations, or surrounding text

Input:
Source columns = {source_headers}
Target columns = {target_headers}

Return your answer inside a Python code block in the following format:
    ‘‘‘python{{ ("source1",): ("target1",), ... }}‘‘‘"
```

Figure 5.2: LLM prompt structure for the schema matching experiment

For each dataset, both versions of the Planning Engine were used to output a plan, which was then scored using *PlanEval*. The results are shown in Figure 5.3 and Table 5.3.

Although leveraging an LLM for schema matching introduces a fixed monetary cost, the consistent improvements in plan quality and correctness justify its use. The observed performance gains are likely attributable to the LLM's extensive pre-training on diverse datasets, enabling it to

[2] https://docs.anthropic.com/en/prompt-library/function-fabricator

accurately infer mappings—including ambiguous ones. Furthermore, refinements to the prompt structure likely enhanced the model's effectiveness.

Additionally, the experiment highlights a limitation of the algorithmic approach used for schema matching, namely that edge weights in the bipartite graph are derived from averaged semantic and syntactic similarities, which leads to poor generalisation in absence of context [53].

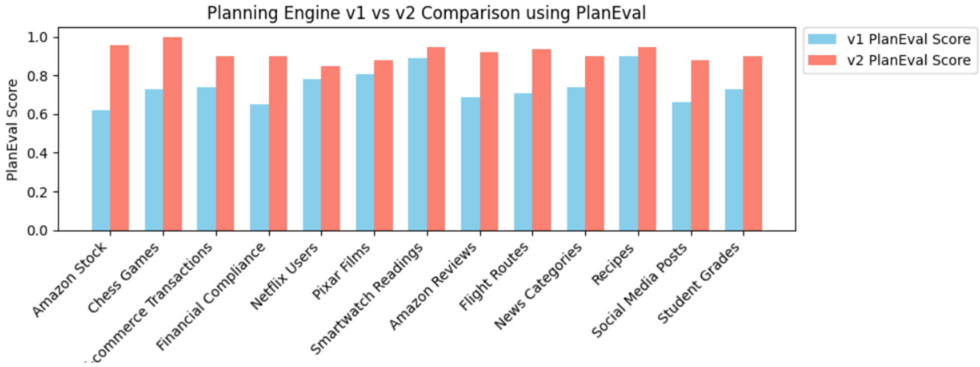


Figure 5.3: Graphical results from the comparison between the Planning Engine using algorithmic schema matching and example-based prompting (v1) VS the Planning Engine using LLMs for both schema matching and transformation inference, without example-based prompting (v2)

Dataset	**v1 PlanEval Score**	**v2 PlanEvalScore**
Amazon Stock	0.62	0.96
Chess Games	0.73	1.00
E-commerce Transactions	0.74	0.90
Financial Compliance	0.65	0.90
Netflix Users	0.78	0.85
Pixar Films	0.81	0.88
Smartwatch Readings	0.89	0.95
Amazon Reviews	0.69	0.92
Flight Routes	0.71	0.94
News Categories	0.74	0.90
Recipes	0.90	0.95
Social Media Posts	0.66	0.88
Student Grades	0.73	0.90

Table 5.3: Tabular results from the comparison between the Planning Engine using algorithmic schema matching and example-based prompting (v1) VS the Planning Engine using LLMs for both schema matching and transformation inference, without example-based prompting (v2)

5.4 Effects of Increasing Sampling Percentage

This experiment evaluates whether varying the sampling percentage within the Source Observer impacts the output of the Planning Engine. It was hypothesised that as the sampling percentage increases, the execution time of the Planning Engine would also increase, and the likelihood of errors or failures in the computed plan would decrease. This is because a larger sample should provide more information for the LLM to process, but the time required to pass the sample between data task nodes would inherently slow down the planning process.

The experiment was carried out as follows. Firstly, the upper bound of 50 rows was removed from the `to_internal()` method. A series of p values from 5% to %100, with an increasing step of 5% was chosen. To select a dataset for this experiment, the median (around 7000) number of objects/rows across all 13 datasets was computed, and the dataset with the closest object count, namely 'Amazon Stock' was selected. For each sampling value p, the time elapsed from the beginning to completion of the planning engine, the plan quality scored using *PlanEval*, and the maximum DQS achieved on the each sample recorded. The results are summarised in Table 5.4.

Sample %	**Time Elapsed (s)**	**PlanEval Score**	**Max DQS Achieved**
5	69.3	0.85	0.98
10	66.0	0.85	0.96
15	78.2	0.90	0.97
20	70.6	0.90	0.96
25	65.0	0.85	0.95
30	73.1	0.90	0.96
35	71.6	0.85	0.96
40	78.0	0.95	0.95
45	75.2	0.90	0.96
50	81.0	0.95	0.96

Table 5.4: Results of increasing the sampling percentage parameter p

The pipeline failed to handle any `p > 0.5` due to the large payload size. While the results were incomplete, a distinct series of patterns emerged, as shown in Figure 5.4. The elapsed time appeared to increase gradually with the sample size. This behaviour was expected, as many data wrangling tasks, such as handling missing values and removing duplicates, require a linear scan of the IR, causing the time taken to grow linearly with the input size. The *PlanEval* score also scaled linearly with p, likely because larger samples provide more representative data, offering the LLM more context to correctly infer schema matches and transformation steps. The maximum DQS achieved remained consistent, indicating that the sampling percentage is not strongly correlated with p. This conclusion is further supported by the Pearson correlation coefficient analysis[3], reported in Table 5.5

Correlation Pair	**Pearson Correlation**
Time Elapsed vs Sample %	0.62
PlanEval vs Sample %	0.77
Max DQS vs Sample %	-0.48

Table 5.5: Pearson correlation coefficients for the sampling percentage evaluation task

Two major limitations of this experiment's design are the monetary costs associated with running FlowETL, and the Planning Engine's capabilities of only handling smaller IRs which fit within XComs' tolerated payload sizes. This limitation likely contributed to pipeline failures when processing larger datasets. Repeating the experiment with additional datasets, potentially capping p to ensure that the IR stays within the payload limit, could produce a more representative set of results.

[3] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.pearsonr.html`

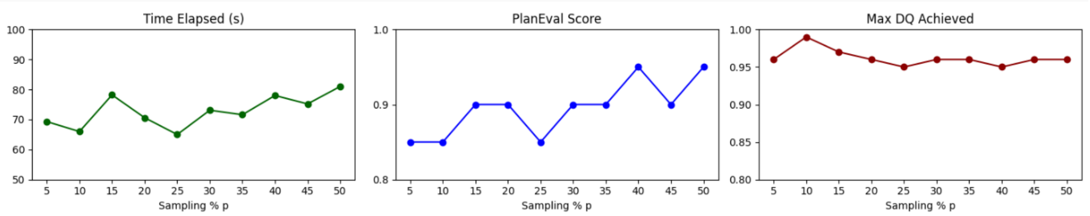


Figure 5.4: Time elapsed (left), *PlanEval* score (middle), Maximum DQS achieved (R) on the "Amazon Stock" dataset for varying sampling percentage p values

5.5 Evaluation Against Competing Tools

Another approach to evaluate FlowETL involved comparing it against other ETL tools available on the market which provided similar functionalities. A major challenge identified during the design of this experiment was the lack of open-source, example-based ETL tools.

Foofah [4] was a promising candidate, however its limited functionality made direct comparison with FlowETL impractical. Bonobo [5] was selected as an alternative. Bonobo is an open-source Python framework designed for creating lightweight, scalable, and maintainable data pipelines using DAGs made up of reusable transformation components. A major issue is that Bonobo is non-autonomous, making it challenging to evaluate FlowETL's planning capabilities against other non-autonomous ETL systems.

The experiment was setup as follows. The author first learned to use the basic functionalities offered by Bonobo. Subsequently, a custom Bonobo transformer method was defined for each evaluation dataset, following the GT previously defined, resulting in 13 different Bonobo workflows being defined for this evaluation. The execution time, data quality, missing values percentage, duplicate rows percentage, and outliers percentage were recorded for each dataset after running their respective pipeline. The results are reported by Table 5.6.

Dataset	**Entries**	**Time (s)**	**DQS**	**M %**	**D %**	**O %**
amazon_stock_data_source.csv	7557	0.16	0.96	0.0	0.127	0.0
chess_games_source.csv	24093	0.68	1.0	0.0	0.0	0.0
ecommerce_transactions_source.csv	650151	11.32	0.98	0.0	0.064	0.0
financial_compliance_source.csv	117	0.01	1.0	0.0	0.01	0.003
netflix_users_source.csv	29827	0.56	0.98	0.045	0.007	0.0
pixar_films_source.csv	36	0.01	1.0	0.0	0.036	0.043
smartwatch_health_data_source.csv	12039	0.17	0.99	0.0	0.028	0.0
news_categories_source.json	280	0.01	1.0	0.0	0.005	0.0
students_grades_source.json	5682	0.24	1.0	0.0	0.0	0.0
social_media_posts_source.json	94	0.01	0.94	0.0	0.017	0.208
recipes_source.json	51361	0.63	1.0	0.0	0.0	0.0
flight_routes_source.json	10695	0.28	1.0	0.0	0.0	0.001
amazon_reviews_source.json	1948	0.06	1.0	0.0	0.001	0.0

Table 5.6: Results gathered after transforming all datasets using Bonobo. The 'time' column only indicates each pipeline's runtime and does not account for development time.

[4] https://github.com/umich-dbgroup/foofah

[5] https://www.bonobo-project.org/

The current runtime measurements for Bonobo are not fully representative, as they exclude the time spent analysing the input datasets, interpreting the ground truth, and gaining familiarity with the Bonobo framework prior to implementing the required transformations.

A more accurate evaluation could involve recruiting developers to implement the required transformations for each dataset, repeating the task for both Bonobo and FlowETL. The hypothesis is that FlowETL would prove easier and faster to use, as specifying the target output for a dataset is expected to require less effort than constructing an equivalent workflow using Bonobo. If the manual implementation takes longer than executing the complete ETL process with FlowETL, it would indicate greater efficiency of the latter. This form of human-in-the-loop evaluation was not originally planned and therefore represents an unaddressed limitation. Therefore, the evaluation focused primarily on FlowETL's ability to handle common data wrangling challenges, alongside correctly inferring and applying a transformation plan.

In addition, the data quality of the transformed output and the corresponding *PlanEval* Score were recorded for each execution. For every dataset, a corresponding target output containing 5 to 7 entries was manually constructed, resulting in 13 source-target dataset pairs. Each pair was processed by FlowETL using a sampling rate defined as $p = \max(\texttt{object\_count} \times 0.05, 50)$. The same evaluation metrics used in the Bonobo experiment were collected to enable a dataset-wise comparison of the pipelines' data wrangling performance. The results are presented in Table 5.7.

Dataset	Time (s)	DQS	M%	D%	O%	PlanEval Score
Amazon Stock CSV	140.2	0.96	0.00	3.41	0.00	0.96
Chess Games CSV	102.3	0.97	0.00	0.00	0.00	1.00
E-commerce Transactions CSV	99.4	0.99	0.00	2.73	0.00	0.90
Financial Compliance CSV	111.9	0.99	0.00	0.00	0.00	0.90
Netflix Users CSV	79.5	0.98	3.76	4.58	0.00	0.85
Pixar Films CSV	95.7	0.98	1.93	3.56	2.80	0.88
Smartwatch Readings CSV	111.6	0.96	0.00	0.00	0.00	0.95
Amazon Reviews JSON	79.0	1.00	0.00	1.32	0.00	0.92
Flight Routes JSON	90.1	0.99	0.00	0.00	1.51	0.94
News Categories JSON	76.2	0.99	0.00	1.73	0.00	0.90
Recipes JSON	88.9	0.94	0.00	0.00	0.00	0.95
Social Media Posts JSON	76.1	0.98	0.00	3.44	4.62	0.88
Student Grades JSON	106.1	1.00	0.00	0.00	0.00	0.90

Table 5.7: Runtime results gathered by running FlowETL on all evaluation datasets

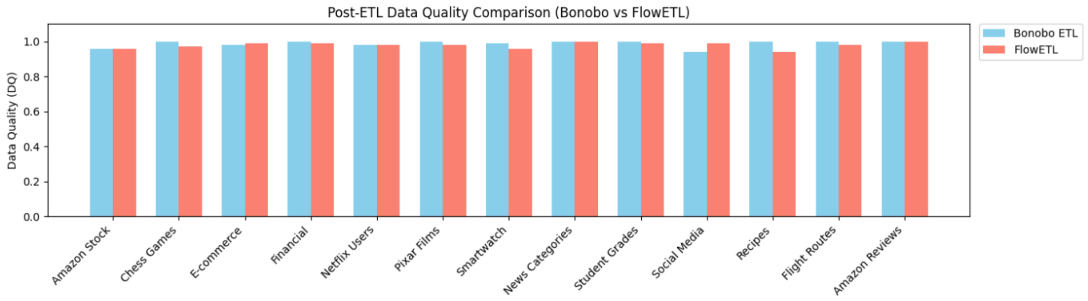


Figure 5.5: Comparison of the DQS achieved on each evaluation dataset by FlowETL and Bonobo

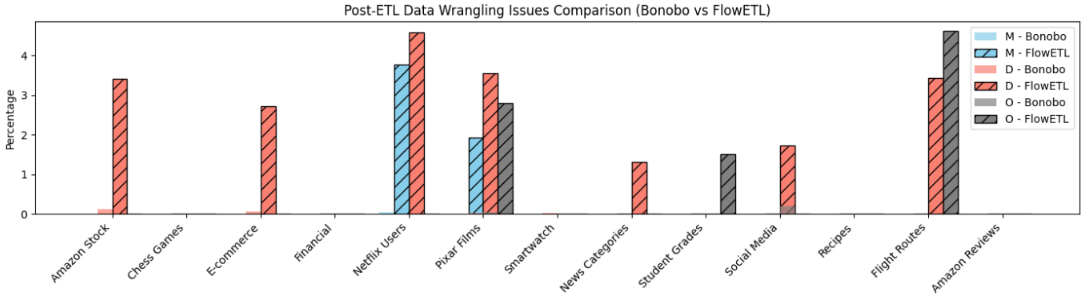


Figure 5.6: Percentage of data wrangling issues present in all datasets post-ETL. Comparison between FlowETL and Bonobo

As the results show, FlowETL achieved post-ETL data quality scores ranging from 0.94 to 1.00, which are comparable to those achieved with Bonobo. However, the overall data quality across all datasets was slightly lower for FlowETL. This can likely be attributed to a higher incidence of unresolved data wrangling issues in the output, as illustrated in Figure 5.5. A contributing factor may be poor generalisation of the Planning Engine's output on the entire dataset. Despite these limitations, FlowETL demonstrated strong generalisation capabilities, consistently producing high-quality outputs and PlanEval scores while autonomously inferring and executing transformation steps. However, it does remain more error-prone than Bonobo, largely due to the inherent variability in LLM-generated outputs.

Chapter 6

Discussion and Conclusions

6.1 Strengths

Following the evaluation, the following strengths were identified with this project.

Firstly, the pipeline seems to generalise to a wide range of domains. As shown by the experimental results comparing the pipeline's output to the ground truth for each file, the pipeline correctly inferred and applied 7 out of 13 plans, while achieving almost perfect results for the other 6.

Additionally, FlowETL correctly handles both structured and unstructured datasets without loss of information. The pipeline also correctly handled data wrangling anomalies in most cases, achieving a DQS between 0.96 and 1.0 for all evaluation datasets. The pipeline also computes a plan in constant time with respect to the source file size, thanks to the sampling step performed by the Source Observer.

Furthermore, evaluation results suggest that FlowETL significantly reduces time required for the design, implementation, and testing of transformation instructions compared to traditional tools such as Bonobo. This shows that the pipeline reduces manual involvement without sacrificing data quality or performance.

6.2 Limitations

Three major limitations were identified throughout the project's development.

The first limitation concerns the scalability of the inward translation mechanism from source file to an IR. As observed in the evaluation of the Planning Engine's performance against varying sampling percentages, larger files introduce a slight, but noticeable, increase latency and communication overheads between components. This gradually increases the Planning Engine's runtime, until a failure point is reached when the payload size exceeds manageable thresholds.

Additionally, if the source file is too large to be entirely read locally by the Source Observer or the ETL worker, runtime errors due to insufficient resources available are likely to occur. A possible solution is to leverage existing solutions with built-in partition mechanisms, designed to process large files by distributing workload. This actionable step proses a re-implementation of the ETL worker using Apache Spark and translating the the file into a Spark Dataframe. This would enable for a transformation plan to be computed once within the Planning Engine and distributed across multiple workers to apply the plan on the source file concurrently [4].

The second limitation concerns the monetary cost associated with executing the pipeline. As outlined in Chapter 4, downloading LLMs locally was unfeasible due to the extensive hardware requirements. The selected alternative requires interfacing with the chosen LLM through API requests, which incurred a marginal financial cost. During development, the cost averaged around $0.06-0.08 per execution and remained constant for all file types and sizes. This limitation posed restrictions at both the development and evaluation stages of the project, however this cost was deemed to be a worthy investment to achieve a more performant and generalisable pipeline.

The last limitation identified concerns the lack of support for data enrichment using external sources. As a consequence, FlowETL assumes that the desired transformations can be inferred solely by modifying analysing the target dataset. Providing a data-context or supplementary resources to the planning engine could enable the LLM to consider enrichment strategies as well, further enhancing the architecture's capabilities.

6.3 Future Work

Several areas for future work became evident throughout the development of this project.

As described in Chapter 3, FlowETL's architecture has been conceived to be expandable and configurable in order account for evolving needs and requirements.

Firstly, the underlying LLM used for both the schema matching and transformation logic inference tasks could be injected within the architecture rather than be reserved to the author's choice. This would enable enterprises working in security-critical sectors where data-privacy is key to incorporate their in-house model. While this mitigates any data governance and privacy risks, it may reduce performance and generalisation capabilities if the selected LLM is not fine-tuned for code generation or similar tasks, therefore there is a trade-off between data management and generalisation capabilities.

Additionally, the architecture could be configured with custom-defined DTNs implementation, enabling users with the choice of either providing a custom implementation for a particular node or even expand the set of available nodes to cover more data engineering tasks, such as the ones defined in [35].

Another improvement looks at expanding the set of strategies currently available to each DTN. For example, by leveraging machine learning techniques such as Isolation Forest [1] to detect numerical outliers, or using regression models to impute missing values.

Another area of development looks at adding additional data types to the architecture. Examples include date-time, adding more granularity to the `numbers` type and defining `double` and `int` subtypes. Due to the restricted time available however, this extended set of types remained a point of improvement and not a characteristic of the architecture

Additionally, allowing users to pre-configure certain components of the pipeline by supplying a `.yml` configuration file could significantly improve the pipeline's customisability. The user could specify a pre-defined strategy for each data task node, or even supply an entire transformation plan through this new file. In scenarios where a subset of promising plans is identified for a particular dataset a-priori, this file could instruct the pipeline to only consider certain plan possibilities, removing the need to compute and evaluate less-promising or redundant plans.

A further improvement to the system's usability looks at providing a simple GUI to enable

[1] `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html`

seamless upload of source and target files via drag-and-drop mechanism. This would make the system more accessible for non-technical users.

Another improvement, concerned with performance, is to integrate stronger and more robust mechanisms to cache plans previously computed. This would allow for integration of data streaming without re-computing the plan for each micro-batch of data received by the system. Furthermore, the pipeline could be configured to run both on a time-based and event-based fashion, ideal for supporting both batch and stream processing, as outlined in the ideal pipeline architecture described in [29]

Extending support to file types beyond CSV and JSON would enhance FlowETL's applicability across a broader range of domains. Achieving this would require implementing both inward and outward translation mechanisms for each additional file type.

The final improvement concerns the level of complexity supported during the schema matching step for JSON files. At the moment, the pipeline only supports one level of nesting, whereas some scenarios might require schema matching to be executed on multiple levels of nesting, especially with unstructured data such as JSON and XML. To account for this, the translation mechanisms need to be extended to consider multiple nesting levels. Additionally, a more thorough evaluation of the LLMs capabilities in handling such matching requirements needs to be carried out. A promising solution is described in [2], where the authors compute a *path distance* heuristic to handle more complicated schema-matching problems.

6.4 Project Summary and Conclusion

The author designed, developed and evaluated FlowETL : a novel and autonomous ETL pipeline architecture capable of transforming a source dataset into a desired target dataset by inferring a transformation plan from the user-supplied target.

FlowETL has been evaluated across 13 datasets, consisting of both structured and unstructured data from different domains and of different sizes, showing promising performance and generalisation capabilities.

Future work is targeted towards extending the data wrangling capabilities of the architecture, as well as performing a more in-depth evaluation of the underlying schema matching and transformation logic inference steps. Deploying the pipeline in an enterprise environment could provide beneficial insights into the strengths and weaknesses of the proposed architecture.

Appendix A

User Manual

This manual outlines the necessary steps to install and run FlowETL. It is worth mentioning that FlowETL has been developed and tested exclusively on the Windows 11 operating system. Therefore, it is recommended to use Windows 11 for optimal compatibility and performance.

A.1 System Requirements

- Python 3 installed. Download from `https://www.python.org/downloads/`
- Git and Git Bash - required to interact with FlowETL through the command line. Download available at `https://git-scm.com/downloads/win`
- Docker Desktop - required to instantiate the Planning Engine and the Messaging System components within FlowETL. Download available at `https://docs.docker.com/desktop/setup/install/windows-install/`

A.2 Setup Instructions

A.2.1 Obtain the Codebase

Obtain the FlowETL codebase using one of following approaches:

1. Using the Git Bash terminal, create and travel into a new folder. Run the command `git clone https://github.com/MattiaDiProfio/FlowETL.git`
2. If given access rights, visit `https://github.com/MattiaDiProfio/FlowETL.git`, then click "Code" > "Download ZIP".

A.2.2 Provide the LLM Inference API key

Within the base folder for FlowETL, create a `.env` file and add `OPENROUTER_API_KEY="..."`. You should use the API key provided by the author or set up your own one from `https://openrouter.ai/`.

A.2.3 Build FlowETL Instance

1. Start Docker Desktop on your machine
2. Using Git Bash, travel to the directory `docker_services/`.
3. Run the command `docker build -t test:latest .` to create a Docker image using the Dockerfile.

4. Using Git Bash, travel back to the base directory and run the command `pip install -r requirements.txt` to install all the required dependecies.

A.3 Starting the Pipeline

1. Launch Docker Desktop from your machine

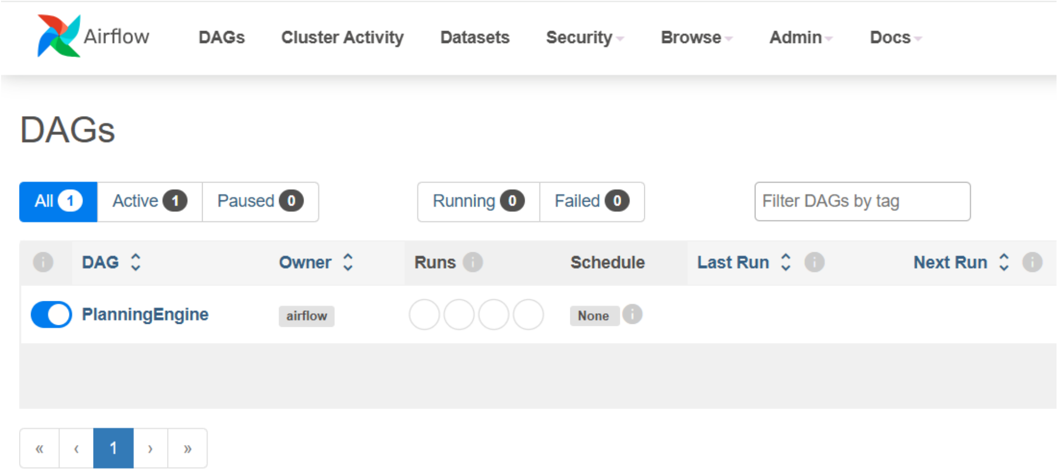
2. Using Git Bash, travel to the directory `docker_services/` and execute the command `docker compose up -d` to start the Planning Engine and Messaging Broker.

3. Wait for 30 seconds, then open up new browser tab (Google Chrome is recommended). Travel to `localhost:8080` and login using "airflow" for both username and password. Once logged-in, you should see the same contents as Figure A.1.

4. Click the DAG name and press the start button. You should see the first two task nodes turning light green, indicating that the Planning Engine has started.

5. To start the Observers, open a new Git Bash terminal and activate the virtual environment by running the command `source venv/Scripts/activate`, then travel to `observers/` and run the command `python driver.py`

6. To start the ETL worker and Reporting Engine, open a new Git Bash terminal for each, travel to the base directory of your FlowETL codebase, activate the virtual environment. Run the command `python etl_worker.py` in one terminal and `python reporter.py` in the other.



DAG	Owner	Runs	Schedule	Last Run	Next Run
PlanningEngine	airflow	○○○○	None		

Figure A.1: Screenshot of Apache Airflow dashboard

A.4 Stopping the Pipeline

1. Stop all processes running on Git Bash by clicking on each terminal and pressing `CTRL + C` on your keyboard.

2. Stop the docker services either via Docker Desktop or by travelling to the directory `docker_services/` and running the command `docker compose down`.

A.5 Using the Pipeline

1. Given a source file to be transformed, define a Target file according to the following rules. Refer to the `evaluation_datasets` folder for 13 examples of both CSV and JSON files.

    - Column headers/keys can be renamed, merged, dropped, split, or unchanged.
    - Columns/keys must not contain missing values or outliers
    - Rows/objects should not be duplicated
    - It should be representative of the transformations to be applied
    - It should be concise (no more than 10 rows/objects)

2. Upload the source file to the `input/source` folder

3. Upload the target file to the `input/target` folder

4. Wait around 2 minutes. A transformed copy of the source file will be found in the `output` folder. Meanwhile, you can check the Planning Engine's progress through the Airflow dashboard, as seen in Figure A.2

5. Information about the runtime will be found in the `observers/logs`, `logs`, and `etl_logs` folders.

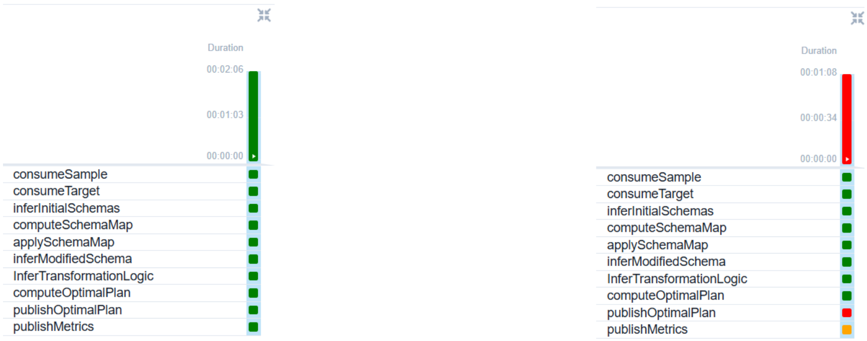


Figure A.2: Comparison of a successful execution (left) and failed execution (right) of the Planning Engine

A.6 Troubleshooting and Common Errors

- Failure during LLM inference - could be caused by either an invalid API key or insufficient credits. Check with the author, or check your Open-Router account if you are supplying your own key.

- ETL worker cannot apply plan due to syntax or logic error. Could be caused by the non-deterministic nature of the LLM. Try to modify the target file if there are some under-represented transformations.

It is advised to check the execution logs for any errors or anomalies, in particular the Planning Engine's individual task nodes. Click on any task node and travel to the `Logs` or `XCom` tags for more information about the execution of the node.

Appendix B

Maintenance Manual

This manual provides additional information to configure, extend, or evaluate the project. An annotated file tree describing the codebase organisation is additionally outlined in Figure B.1

B.1 Contributing

The supported ways of contributing to this project are listed below. If you wish for any changes to be integrated within the FlowETL codebase, open a pull request for `https://github.com/MattiaDiProfio/FlowETL.git`

1. **Adding/Modifying DTNs & Strategies** : Add more DTNs and strategies within the `generate_plans` method in the `docker_services/dags/planning_utils.py` file.

   Change the following methods to modify or add strategies :

   - `missing_value_handler` for handling missing values
   - `duplicate_values_handler` for duplicate rows handling
   - `outlier_handler` for numerical outliers handling

2. **Extending the Inward Translation Mechanism** : Modify the `to_internal` method found in `docker_services/dags/planning_utils.py`, `observers/observer_utils.py`, and `evaluation/bonobo_etl_experiment/bonoboutils.py` to handle more file types by extending the conditional statement (i.e., `...if filetype == 'xml'...`) and defining the translation logic from source to IR. Additionally the `load` method must be modified to reflect these changes.

B.2 Additional Configurations

This subsection provides guidance on configuring FlowETL's internal parameters and running evaluation subroutines.

1. **Hyper-parameter Tuning** :

   - Sampling percentage `p` - found in the `extract_sample` method within the file `observers/observer_utils.py`. Note that `p` should be between 0.01 and 1.0 and be kept low for larger files (5000+ objects).
   - Sample size upper bound `cap` - found in the `extract_sample` method within the file `observers/observer_utils.py`. This should be kept below 100 for optimal performance.

- LLM task prompts - found within `docker_services/dags/airflowDAG.py` file in the `infer_transformation_logic` and `compute_schema_map` methods respectively.

2. **Running Unit Tests** : Activate the virtual environment `venv` from within your FlowETL base directory, then run `python unit_tests.py`.

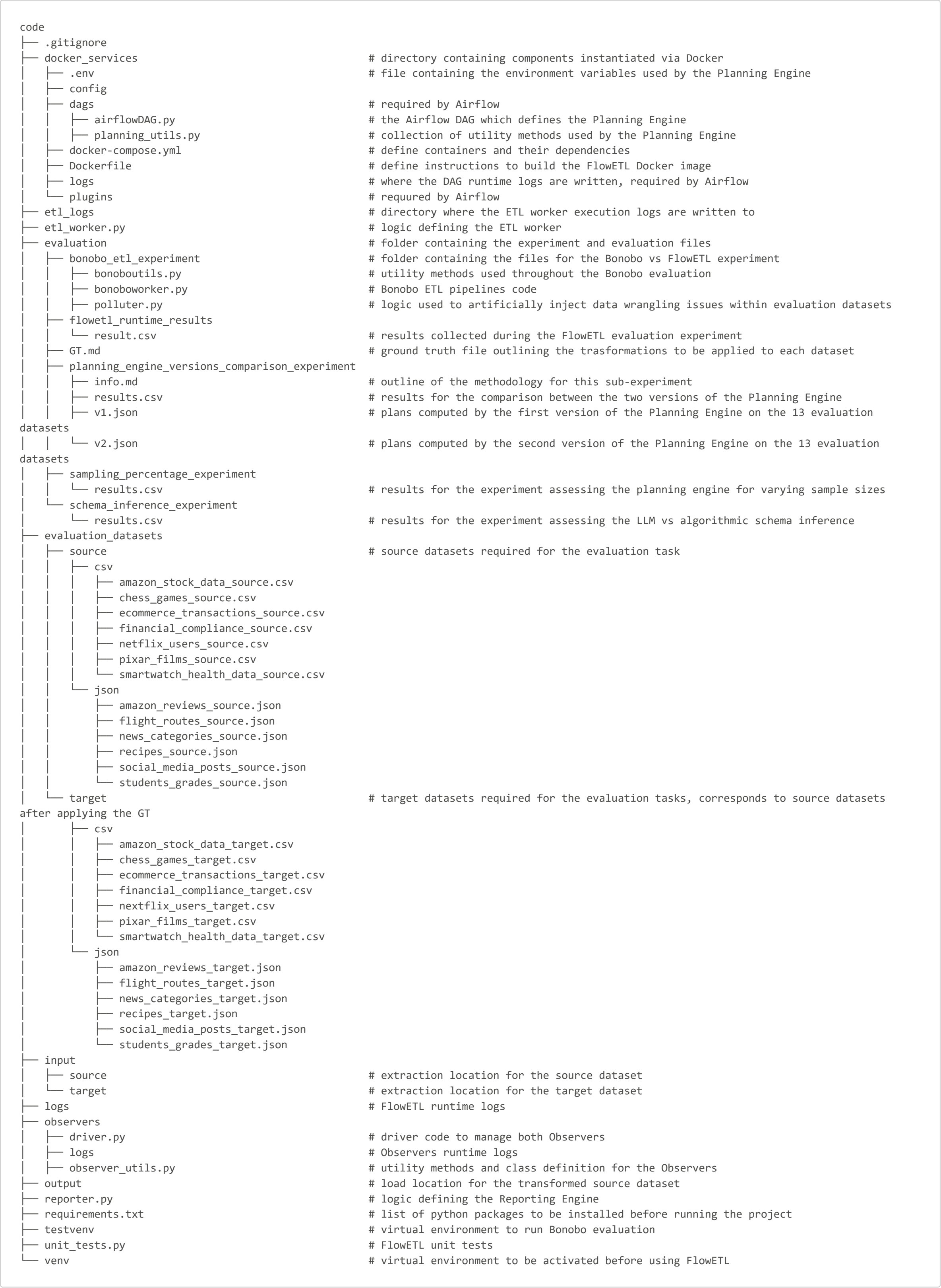
3. **Running Bonobo Evaluation** : Activate the virtual environment `testvenv`, then travel to the folder `evaluation/bonobo_etl_experiment` and modify the `bonoboworker.py` file to set the source dataset filepath. Finally, run the worker with `python bonoboworker.py`

B.3 Bug Reports

Two minor bugs have been identified during the evaluation stage of this project.

1. **LLM Hallucination** : In some instances, the model used for both inference tasks (Claude-3.7-Sonnet), hallucinates on the transformation logic inference step within the *inferTransformationLogic* node of the Planning Engine. Try modifying the prompt hyper-parameters or validate the target dataset against the requirements listed in Appendix A.5.

2. **Reporting Engine Runtime** : In cases of an upstream component failure, such as the ETL Worker, the Reporting Engine will continue polling the Kafka broker endlessly and requires to be manually terminated. Implementing a subroutine which checks the uptime of other components is a possible solution.



```
code
├── .gitignore
├── docker_services                                     # directory containing components instantiated via Docker
│   ├── .env                                            # file containing the environment variables used by the Planning Engine
│   ├── config
│   ├── dags                                            # required by Airflow
│   │   ├── airflowDAG.py                               # the Airflow DAG which defines the Planning Engine
│   │   ├── planning_utils.py                           # collection of utility methods used by the Planning Engine
│   ├── docker-compose.yml                              # define containers and their dependencies
│   ├── Dockerfile                                      # define instructions to build the FlowETL Docker image
│   ├── logs                                            # where the DAG runtime logs are written, required by Airflow
│   └── plugins                                         # requured by Airflow
├── etl_logs                                            # directory where the ETL worker execution logs are written to
├── etl_worker.py                                       # logic defining the ETL worker
├── evaluation                                          # folder containing the experiment and evaluation files
│   ├── bonobo_etl_experiment                           # folder containing the files for the Bonobo vs FlowETL experiment
│   │   ├── bonoboutils.py                              # utility methods used throughout the Bonobo evaluation
│   │   ├── bonoboworker.py                             # Bonobo ETL pipelines code
│   │   ├── polluter.py                                 # logic used to artificially inject data wrangling issues within evaluation datasets
│   ├── flowetl_runtime_results
│   │   └── result.csv                                  # results collected during the FlowETL evaluation experiment
│   ├── GT.md                                           # ground truth file outlining the trasformations to be applied to each dataset
│   ├── planning_engine_versions_comparison_experiment
│   │   ├── info.md                                     # outline of the methodology for this sub-experiment
│   │   ├── results.csv                                 # results for the comparison between the two versions of the Planning Engine
│   │   ├── v1.json                                     # plans computed by the first version of the Planning Engine on the 13 evaluation
datasets
│   │   └── v2.json                                     # plans computed by the second version of the Planning Engine on the 13 evaluation
datasets
│   ├── sampling_percentage_experiment
│   │   └── results.csv                                 # results for the experiment assessing the planning engine for varying sample sizes
│   └── schema_inference_experiment
│       └── results.csv                                 # results for the experiment assessing the LLM vs algorithmic schema inference
├── evaluation_datasets
│   ├── source                                          # source datasets required for the evaluation task
│   │   ├── csv
│   │   │   ├── amazon_stock_data_source.csv
│   │   │   ├── chess_games_source.csv
│   │   │   ├── ecommerce_transactions_source.csv
│   │   │   ├── financial_compliance_source.csv
│   │   │   ├── netflix_users_source.csv
│   │   │   ├── pixar_films_source.csv
│   │   │   └── smartwatch_health_data_source.csv
│   │   └── json
│   │       ├── amazon_reviews_source.json
│   │       ├── flight_routes_source.json
│   │       ├── news_categories_source.json
│   │       ├── recipes_source.json
│   │       ├── social_media_posts_source.json
│   │       └── students_grades_source.json
│   └── target                                          # target datasets required for the evaluation tasks, corresponds to source datasets
after applying the GT
│       ├── csv
│       │   ├── amazon_stock_data_target.csv
│       │   ├── chess_games_target.csv
│       │   ├── ecommerce_transactions_target.csv
│       │   ├── financial_compliance_target.csv
│       │   ├── nextflix_users_target.csv
│       │   ├── pixar_films_target.csv
│       │   └── smartwatch_health_data_target.csv
│       └── json
│           ├── amazon_reviews_target.json
│           ├── flight_routes_target.json
│           ├── news_categories_target.json
│           ├── recipes_target.json
│           ├── social_media_posts_target.json
│           └── students_grades_target.json
├── input
│   ├── source                                          # extraction location for the source dataset
│   └── target                                          # extraction location for the target dataset
├── logs                                                # FlowETL runtime logs
├── observers
│   ├── driver.py                                       # driver code to manage both Observers
│   ├── logs                                            # Observers runtime logs
│   ├── observer_utils.py                               # utility methods and class definition for the Observers
├── output                                              # load location for the transformed source dataset
├── reporter.py                                         # logic defining the Reporting Engine
├── requirements.txt                                    # list of python packages to be installed before running the project
├── testvenv                                            # virtual environment to run Bonobo evaluation
├── unit_tests.py                                       # FlowETL unit tests
└── venv                                                # virtual environment to be activated before using FlowETL
```

Figure B.1: Cobase file structure with description of each file's role and contents

Bibliography

[1] Syed Muhammad Fawad Ali. Next-generation etl framework to address the challenges posed by big data. In *DOLAP*, 2018.

[2] Mirza Beg, Laurent Charlin, and Joel So. Maxsm: A multi-heuristic approach to xml schema matching. *Journal of Computer Science and Technology*, 2006.

[3] Philip A Bernstein, Sergey Melnik, and John E Churchill. Incremental schema matching. In *VLDB*, volume 6, pages 1167–1170. Seoul, Korea, 2006.

[4] Matei Zaharia Bill Chambers. *Apache Spark, The Definitive Guide*, chapter 2, A Gentle Introduction to Spark, pages 24–25. O'Reilly, 2018.

[5] Jan-Micha Bodensohn, Ulf Brackmann, Liane Vogel, Matthias Urban, Anupam Sanghi, and Carsten Binnig. Llms for data engineering on enterprise data. *Proceedings of the VLDB Endowment. ISSN*, 2150:8097, 2024.

[6] Azzedine Boukerche, Lining Zheng, and Omar Alfandi. Outlier detection: Methods, models, and classification. *ACM Comput. Surv.*, 53(3), June 2020. ISSN 0360-0300. doi: 10.1145/3381028. URL `https://doi.org/10.1145/3381028`.

[7] José Camacho, Gabriel Maciá-Fernández, Jesús Díaz-Verdejo, and Pedro García-Teodoro. Tackling the big data 4 vs for anomaly detection. In *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 500–505, 2014. doi: 10.1109/INFCOMW.2014.6849282.

[8] Sara B Dakrory, Tarek M Mahmoud, Abdelmgeid A Ali, et al. Automated etl testing on the data quality of a data warehouse. *International Journal of Computer Applications*, 131(16): 9–16, 2015.

[9] Narendra Devarasetty. Automating data pipelines with ai: From data engineering to intelligent systems. *Revista de Inteligencia Artificial en Medicina*, 9(1), 2018. URL `https://redcrevistas.com/index.php/Revista`.

[10] AnHai Doan, Alon Halevy, and Zachary Ives. 4 - string matching. In AnHai Doan, Alon Halevy, and Zachary Ives, editors, *Principles of Data Integration*, pages 95–119. Morgan Kaufmann, Boston, 2012. ISBN 978-0-12-416044-6. doi: https://doi.org/10.1016/B978-0-12-416044-6.00004-1. URL `https://www.sciencedirect.com/science/article/pii/B9780124160446000041`.

[11] Uwe Draisbach and Felix Naumann. Dude: The duplicate detection toolkit. In *Proceedings of the International Workshop on Quality in Databases (QDB)*, volume 100000, page 10000000, 2010.

[12] Mahmoud Efatmaneshnik and Michael Ryan. A study of the relationship between system testability and modularity. *INSIGHT*, 20(1):20–24, 2017. doi: https://doi.org/10.1002/inst.

12140. URL `https://incose.onlinelibrary.wiley.com/doi/abs/10.1002/inst.12140`.

[13] Tim Furche, George Gottlob, Leonid Libkin, Giorgio Orsi, and Norman Paton. Data wrangling for big data: Challenges and opportunities. In *Advances in Database Technology—EDBT 2016: Proceedings of the 19th International Conference on Extending Database Technology*, pages 473–478, 2016.

[14] Avigdor Gal. Managing uncertainty in schema matching with top-k schema mappings. In *Journal on Data Semantics VI*, pages 90–114. Springer, 2006.

[15] David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American mathematical monthly*, 69(1):9–15, 1962.

[16] Fausto Giunchiglia, Pavel Shvaiko, and Mikalai Yatskevich. Semantic schema matching. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 347–365. Springer, 2005.

[17] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2): 100–107, 1968.

[18] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek Narasayya, and Surajit Chaudhuri. Transform-data-by-example (tde) an extensible search engine for data transformations. *Proceedings of the VLDB Endowment*, 11(10):1165–1177, 2018.

[19] Hans-Arno Jacobsen. *Publish/Subscribe*, pages 2208–2211. Springer US, Boston, MA, 2009. ISBN 978-0-387-39940-9. doi: 10.1007/978-0-387-39940-9_1181. URL `https://doi.org/10.1007/978-0-387-39940-9_1181`.

[20] Xiaolin Jiang, Hossein Shokri-Ghadikolaei, Gabor Fodor, Eytan Modiano, Zhibo Pang, Michele Zorzi, and Carlo Fischione. Low-latency networking: Where latency lurks and how to tame it. *Proceedings of the IEEE*, 107(2):280–306, 2019. doi: 10.1109/JPROC.2018.2863960.

[21] Tengjun Jin, Yuxuan Zhu, and Daniel Kang. Elt-bench: An end-to-end benchmark for evaluating ai agents on elt pipelines. *arXiv preprint arXiv:2504.04808*, 2025.

[22] Zhongjun Jin, Michael R. Anderson, Michael Cafarella, and H. V. Jagadish. Foofah: Transforming data by example. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 683–698, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450341974. doi: 10.1145/3035918.3064034. URL `https://doi.org/10.1145/3035918.3064034`.

[23] Zhongjun Jin, Yeye He, and Surajit Chauduri. Auto-transform: learning-to-transform by patterns. *Proceedings of the VLDB Endowment*, 13(12):2368–2381, 2020.

[24] Krishna Kanagarla. Data engineering with generative ai automating pipelines and transformations using llms like gpt. *Available at SSRN 5107348*, 2025.

[25] Veit Köppen, Björn Brüggemann, and Bettina Berendt. Designing data integration: the etl pattern approach. *UPGRADE: the European Journal for the Informatics Professional*, (3): 49–55, 2011.

[26] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench

for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36, 2024.

[27] Yinheng Li. A practical survey on zero-shot prompt design for in-context learning. *arXiv preprint arXiv:2309.13205*, 2023.

[28] Yurong Liu, Eduardo Pena, Aecio Santos, Eden Wu, and Juliana Freire. Magneto: Combining small and large language models for schema matching. *arXiv preprint arXiv:2412.08194*, 2024.

[29] Anthony Mbata, Yaji Sripada, and Mingjun Zhong. A survey of pipeline tools for data engineering. *arXiv preprint arXiv:2406.08335*, 2024.

[30] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings 18th international conference on data engineering*, pages 117–128. IEEE, 2002.

[31] Tom Mens and Michel Wermelinger. Separation of concerns for software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(5):311–315, 2002. doi: https://doi.org/10.1002/smr.257. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.257`.

[32] Kartick Chandra Mondal, Neepa Biswas, and Swati Saha. Role of machine learning in etl automation. In *Proceedings of the 21st International Conference on Distributed Computing and Networking*, ICDCN '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450377515. doi: 10.1145/3369740.3372778. URL `https://doi.org/10.1145/3369740.3372778`.

[33] John Morcos, Ziawasch Abedjan, Ihab Francis Ilyas, Mourad Ouzzani, Paolo Papotti, and Michael Stonebraker. Dataxformer: An interactive data transformation tool. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 883–888, 2015.

[34] Lincoln Murr, Morgan Grainger, and David Gao. Testing llms on code generation with varying levels of prompt specificity, 2023. URL `https://arxiv.org/abs/2311.07599`.

[35] Alfredo Nazabal, Christopher KI Williams, Giovanni Colavizza, Camila Rangel Smith, and Angus Williams. Data engineering for data analytics: A classification of the issues, and case studies. *arXiv preprint arXiv:2004.12929*, 2020.

[36] Gwen Shapira Todd Palino Neha Narkhede. *Kafka, The Definitive Guide*, chapter 6, Reliable Data Delivery, pages 115–134. O'Reilly, 2017.

[37] Shagofah Noor, Omid Tajik, and Jawad Golzar. Simple random sampling. *International Journal of Education & Language Studies*, 1(2):78–82, 2022.

[38] Sandeep Pushyamitra Pattyam. *Data Engineering for Business Intelligence: Techniques for ETL, Data Integration, and Real-Time Reporting*. Hong Kong Science Publishers, USA, 2021. Independent Researcher and Data Engineer.

[39] Erhard Rahm and Philip A Bernstein. A survey of approaches to automatic schema matching. *the VLDB Journal*, 10:334–350, 2001.

[40] Erhard Rahm and Philip A Bernstein. On matching schemas automatically. *VLDB journal*, 10(4):334–350, 2001.

[41] Vijayshankar Raman and Joseph M Hellerstein. Potter's wheel: An interactive data cleaning

system. In *VLDB*, volume 1, pages 381–390, 2001.

[42] Nabeel Seedat and Mihaela van der Schaar. Matchmaker: Self-improving large language model programs for schema matching. *arXiv preprint arXiv:2410.24105*, 2024.

[43] Eitam Sheetrit, Menachem Brief, Moshik Mishaeli, and Oren Elisha. Rematch: Retrieval enhanced schema matching with llms. *arXiv preprint arXiv:2403.01567*, 2024.

[44] Karanjit Singh and Shuchita Upadhyaya. Outlier detection: applications and techniques. *International Journal of Computer Science Issues (IJCSI)*, 9(1):307, 2012.

[45] Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. *arXiv preprint arXiv:1204.6079*, 2012.

[46] Steven S. Skiena. *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2nd edition, 2008. ISBN 1848000693.

[47] Abir Smiti. A critical overview of outlier detection methods. *Computer Science Review*, 38: 100306, 2020. ISSN 1574-0137. doi: https://doi.org/10.1016/j.cosrev.2020.100306. URL `https://www.sciencedirect.com/science/article/pii/S1574013720304068`.

[48] Jimmy J.M Tan. A necessary and sufficient condition for the existence of a complete stable matching. *Journal of Algorithms*, 12(1):154–178, 1991. ISSN 0196-6774. doi: https://doi.org/10.1016/0196-6774(91)90028-W. URL `https://www.sciencedirect.com/science/article/pii/019667749190028W`.

[49] Jimmy J.M. Tan and Hsueh Yuang-Cheh. A generalization of the stable matching problem. *Discrete Applied Mathematics*, 59(1):87–102, 1995. ISSN 0166-218X. doi: https://doi.org/10.1016/0166-218X(93)E0154-Q. URL `https://www.sciencedirect.com/science/article/pii/0166218X93E0154Q`.

[50] Panos Vassiliadis, Alkis Simitsis, and Eftychia Baikousi. A taxonomy of etl activities. In *Proceedings of the ACM Twelfth International Workshop on Data Warehousing and OLAP*, DOLAP '09, page 25–32, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605588018. doi: 10.1145/1651291.1651297. URL `https://doi.org/10.1145/1651291.1651297`.

[51] Jianxun Wang and Yixiang Chen. A review on code generation with llms: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pages 284–289. IEEE, 2023.

[52] Jiawei Yang, Susanto Rahardja, and Pasi Fränti. Outlier detection: how to threshold outlier scores? In *Proceedings of the International Conference on Artificial Intelligence, Information Processing and Cloud Computing*, AIIPCC '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450376334. doi: 10.1145/3371425.3371427. URL `https://doi.org/10.1145/3371425.3371427`.

[53] Kaitlyn Zhou, Kawin Ethayarajh, Dallas Card, and Dan Jurafsky. Problems with cosine as a measure of embedding similarity for high frequency words. *arXiv preprint arXiv:2205.05092*, 2022.