POLITECNICO DI TORINO

Master Degree in Data Science and Engineering

Master Thesis

Optimizing ETL Processes: automation for SSIS Packages

Candidate:
Mihaita Andrei Boboc
330836

Academic Advisor:
dott. Alessandro Fiori
Company Advisor:
dott. Filippo Balla

academic year 2024-2025

Contents

Abstract

Extract, Transform, Load (ETL) processes are an essential part of building data warehouses. They allow data from many different systems to be collected, cleaned, and stored in a form that can be used for analysis. In Microsoft SQL Server Integration Services (SSIS), these processes are usually created by manually configuring packages. Although this approach provides flexibility, it is repetitive, slow and prone to errors, especially when the system must handle a large number of sources or when several developers are working on the same project. This thesis proposes a metadata driven automation framework written in Python to simplify this task.

Instead of separately building each SSIS package by hand, the system utilizes metadata stored in structured, developed by hand, Excel files and automatically generates packages in XML format. The metadata describes source and target tables, grouping information and referential constraints. This information allows the system to generate packages for the staging layer (L0 level), the operational data storage layer (L1 level) and a pre load step, a special level used to check referential integrity between fact and dimension tables.

The solution is built around reusable XML templates, which are filled dynamically by the Python scripts. In this way, the packages follow a common structure defined by the company's integration framework. Tests carried out on real scenarios show that this method reduces development time, improves consistency, and lowers the risk of human error. Future changes in table structures or business rules can also be managed more easily, as they only require updates to the metadata files. The results obtained confirm that metadata driven automation can improve ETL development in SSIS, make it more reliable, easier to maintain and scalable.

Chapter 1

Introduction

In today's era of digital transformation, data represents an important strategic resource for companies, both to support decision making and to optimize operational processes.
The growth of heterogeneous data sources and the exponential increase in data volume have made of central importance to adopt dedicated infrastructures and tools for data management, such as **Data Warehouses** and **ETL (Extract, Transform, Load)** processes. These technologies allow to consolidate data from various systems, ensuring data quality, consistency and historical tracking, which are fundamental properties for reliable and strategic data analysis.

Among the main challenges companies have to take is the increasing complexity of ETL processes. In particular, the manual development and maintenance are often time consuming, subject to errors and difficult to scale in large and evolving environments. To solve these challenges, it is essential to adopt approaches which introduce automation and standardization.

The following thesis work is relevant to this context and it arises from the internship experience at Mediamente Consulting, a consultancy company specialized in digital transformation. The main objective was to develop an automatic system for the generation of SSIS (SQL Server Integration Services) packages, based on the company's standardized integration framework, reducing manual workload and guaranteeing efficiency and scalability.

To reach such a goal, a Python based program was created, capable of generating automatically XML code, which SSIS packages are based on, starting from hand compiled metadata Excel files. The project has developed a program

capable of covering diverse levels of the ETL integration framework:

- **Staging Area - L0 level**: layer responsible for data ingestion from different heterogeneous sources, with a copy procedure, incremental load and monitoring.
- **Operational Data Store - L1 level**: for data validation and historical tracking through control logics.
- **Pre Load step**: manages the case where a fact table record does not have a referential match in the corresponding dimension table, allowing referential integrity.

The first chapters of this thesis introduce the historical context of reference, with a panoramic view of Data Warehouse architectures, data models and the important difference between OLTP and OLAP systems.

The following chapter focuses on ETL processes and the main challenges they introduce. After, it is introduced and explained the standardized integration framework used by the company.

The next chapter is central in this thesis as it is dedicated to the description of the automation system developed. Firstly, the metadata Excel files are introduced and explained how they are constructed. Next, the base logics of the Python program are presented along the modules developed and how the construction of the SSIS package is performed.

In the end, three different scenarios are presented along with the results of the execution of the generated packages to show the obtained results, which proves the functionality of the program in this context.

Last chapter gathers conclusions and suggest different possible future approaches, such as the extension of the developed system to other framework of the company, integration of generative AI in the creation of the metadata files and generalization of the system to cloud based tools.

Chapter 2

Data Warehouse and its state of the art

The term **Data Warehouse** refers to a centralized system that aggregates a large quantity of historical structured data that originates from various types of sources. It is specifically designed to support analysis processes and strategic decision-making within a company.
Unlike transactional databases, which are optimized for daily operations, a data warehouse has a structure such that it stores historical data over time, which allows for an analysis during a long period of time. Before being stored, the data coming from heterogeneous sources is cleaned and transformed into congruent data, which is suitable for tools that use data to perform analysis.

According to William H. Inmon [11], often considered the father of Data Warehousing, a Data Warehouse is defined by the following distinguishable characteristics:

- **Subject-Oriented**: the data is organized around key business subjects, such as customers, sales or products.
- **Integrated**: Data originates from multiple sources which are different in terms of format and are then integrated into the Data Warehouse
- **Time-Variant**: the data presented in the Data Warehouse is associated to specific time periods, allowing the tracking of changes and the evolution of information over time.
- **Non-Volatile**: once data has entered into the data warehouse, becomes immutable, meaning that it cannot be updated or deleted, ensuring integrity over time.

Ralph Kimball provides a complementary definition, which can be summarized in the statements: *a copy of transaction data specifically structured for query and analysis* [15]. Many other sources summarize the definition of a data warehouse under the same four key properties previously stated [4] [1].

2.1 From Data Mart to Data Lakehouse

It is crucial to understand that Data Warehouses are important within a company but they are not the only options in terms of data archiving.
We are now going to see the various possibilities to store structured data:

2.1.1 Data Mart

Data Marts are databases that are subject-oriented, meaning that they contain data related to a specific area or domain, such as customers or products. While they follow the same principles of a Data Warehouse, they only contain a subset of the information. This structure allows for faster access to needed information and a more targeted analysis. It is also good to note that, in some architectures, a Data Warehouse can be composed of multiple data marts, where each one of them has a focus on a particular area of the company.[9] [2] [20].

2.1.2 Data Lake

Data Lakes are centralized repositories that allow the storage of vast volumes of *raw data* in its original format, which can be structured, semi-structured or unstructured. Unlike Data Warehouses, which need a defined schema and structured data, data lakes follow the "schema-on-read" approach, which means that the data can be ingested without prior transformation and the structure is applied only when the data is queried. Advanced analytics, such as machine learning and real-time analytics, are supported given this allowed flexibility. Figure 2.1 show a simplified process view of how data is ingested, stored, and subsequently utilized within a data lake. However, the other side of the coin is that without proper governance and metadata management, data lakes risk becoming the so called "Data Swamps", which are poorly organized and difficult to navigate through [22].

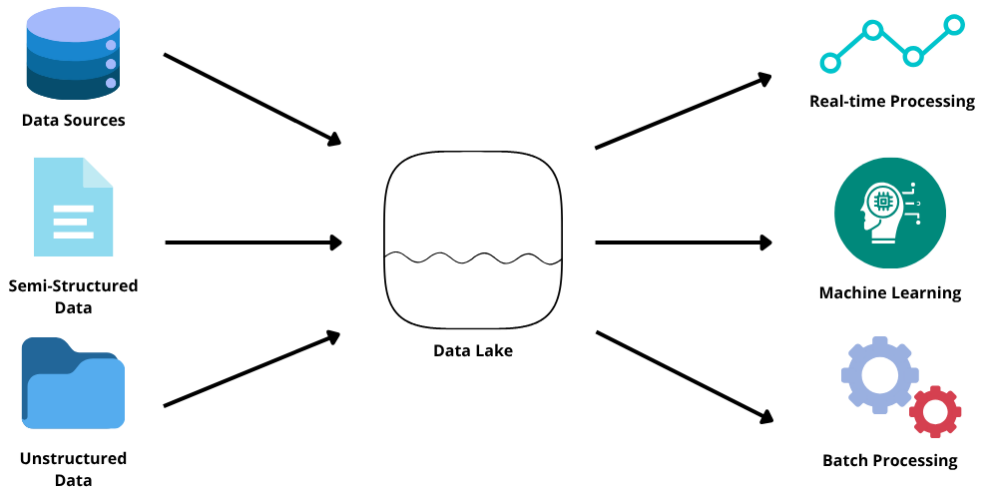

Figure 2.1: Example of a Data Lake

"If you think of a data mart as a store of bottled water, cleansed, packaged and structured for easy consumption, the data lake is a large body of water in a more natural state. The contents of the lake stream in from a source to fill the lake, and various users of the lake can come to examine, dive in, or take samples" [5].

2.1.3 Data Lakehouse

The most modern data architecture currently in use is the **Data Lakehouse**, which is a hybrid solution that merges the benefits of the Data Warehouse with the benefits of Data Lakes.
It introduces the following features:

- Data reliability ensured by the support for **ACID** transactions.
  - **A**tomicity: means that each transaction either is fully completed or it has no effect.
  - **C**onsistency: means that each transaction brings the database from a valid state to another valid state.
  - **I**solation: there is no interference among transactions.
  - **D**urability: once a transaction is committed, the changes made by it are permanent.
- A unified storage formats.

- Governance and indexing supported by a common metadata layer.
- Ability to handle both batch and streaming data in the same environment.

As a result, data management is then simplified by combining the reliability and performance of data warehouses with the scalability and flexibility of data lakes [3] along with the capability to work with both structured and unstructured data in the same data platform.

2.2 Data Warehouse modelling (Star, Snowflake and Data Vault schemas)

The process of defining how data is stored, organized and accessed within a database is called **Data Modelling**. It plays a crucial role in shaping how the business data is logically represented for analytical use.
The goal is to translate real-world companies' business requirements into a formal design that supports

- efficient querying.
- data integration.
- consistency across the system.

For this purpose several modelling approaches have been developed. Each model offers different trade-offs and there is no universal best choice.
We are now going to see more in detail these data models.

2.2.1 Star Schema

One of the most widely used data modelling techniques for the design of Data Warehouse is the Star Schema. As suggested by the name, this model technique resembles a star: a large central table, called *fact table*, that is surrounded by several smaller tables, known as *dimension tables*.
The fact table stores quantitative business metrics such as sales revenue, order quantities or transaction amounts. These metrics, also called measures, are typically associated with multiple perspectives, such as time, product, customer, location or others, where each one is represented by a corresponding dimension table. Figure 2.2 shows an example of how conceptually is modelled a star schema.

A peculiarity of the star schema is the asymmetry property: only the fact table presents multiple foreign key references linking it to various dimension tables. In contrast, dimension tables are not connected to each other and are generally denormalized, meaning that the tables contain redundant or repeated information to reduce the complexity of joins and improve query performance. To be able to identify the records in the fact table, the primary key is composed, in most cases, from the foreign keys of the dimension tables. However, since some of these keys may involve complex attributes, such as fiscal code, artificial primary keys are introduced, called **surrogate keys**. These are just some simple, auto incremented, numeric fields that serve as primary keys with no business meaning. Their use simplifies joins and improves efficiency by avoiding long and variable string based joins [16].
The star schema remains a foundational technique in dimensional modelling as is particularly efficient in large-scale business contexts where queries are focused on aggregating data across multiple dimensions [7][11]

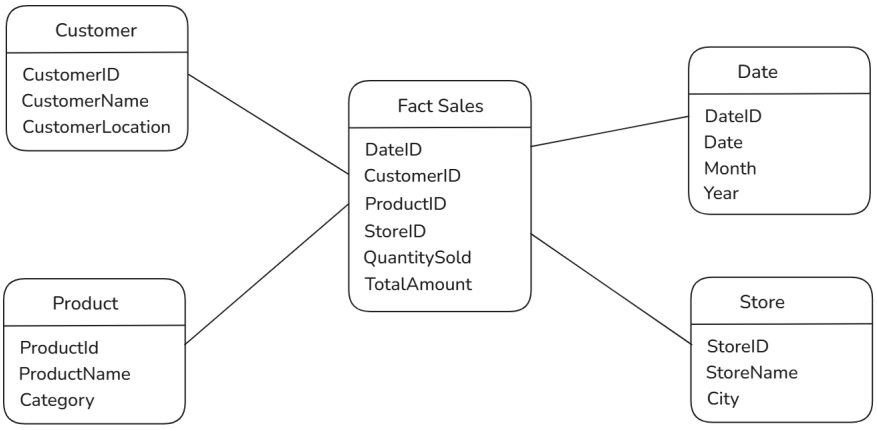

Figure 2.2: Star schema example

2.2.2 Snowflake schema

The **Snowflake Schema** resembles the star schema in its overall structure, containing a central fact table linked to multiple dimension tables. Figure 2.3 illustrates the snowflake schema starting from the star schema of the previous example. However, there is a distinguishable feature of the snowflake model: dimension tables are *normalized*. This aims to reduce data redundancy and improve storage efficiency by organizing dimension data into multiple levels of hierarchy [16].

For instance, a *Date* dimension can be decomposed into separate tables for *Year*, *Month* and *Day*, all linked through foreign keys that maintain the logical relationships among these components. This structure allows for more refined and maintainable hierarchies within the data model [7].
The primary advantage of this model schema is its efficient use of storage which comes from minimizing duplicated data within dimension tables. However, this comes at the cost of increases query complexity as retrieving data often requires more joins across related tables. These additional joins can lead to longer query execution times and may reduce performance, especially in systems where fast query execution is critical [16] [11].

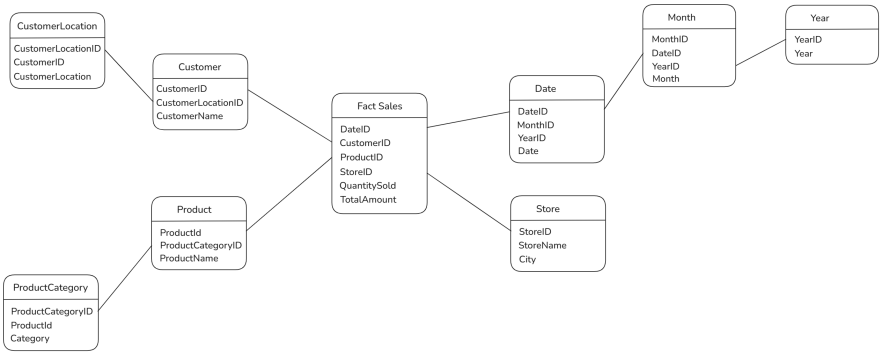

Figure 2.3: Snowflake schema example

2.2.3 Data Vault schema

The **Data Vault** is a modern data modelling approach that has been designed to address the limitations of the star and snowflake schemas. Proposed by Dan Linstedt in the early 2000s, this data model is intended for agile data warehousing and is especially effective in scenarios involving frequent schema evolution and large volumes of historical data [17].
The schema is composed of three main entities:

- **Hubs**: it stores unique business keys, such as "CustomerID", and represent core business entities.
- **Links**: capture the relationships between hubs, such as "Customer" purchased "Product".
- **Satellites**: store descriptive attributes and historical changes associated with hubs or links, such as names of customers or description of products.

This structure separates relationships, business keys and contextual data, making the model highly scalable and extensible. Since satellites are timestamped and versioned, the model supports historical data tracking and data lineage, which align with regulatory and audit requirements [7].
The Data Vault model is highly normalized, trading off query simplicity for flexibility and data traceability. This results in a model that is well-suited for enterprise data warehouse that consolidate data from multiple operational systems and require consistent and reliable data ingestion over time [16].

2.3 Difference between OLTP and OLAP

At this point it is essential to distinguish between **Online Transaction Processing (OLTP)** and **Online Analytical Processing (OLAP)** systems. Even though both involve the storage and the processing of data, they serve fundamentally different purposes within an organization and are optimized for distinct use cases.
OLTP systems are designed to handle day-to-day transactional operations in real-time. They are commonly used for activities such as processing purchases, recording sales, managing customer interactions, and handling financial transactions. These systems usually adopt highly normalized and entity-relationship models to ensure data consistency and integrity during frequent updates, inserts and deletions [16]. OLTP systems require high availability, low latency and robust ACID compliance to be able to ensure that every transaction is correctly recorded, even when systems fail.
In contrast, **OLAP systems** are focused on the analysis of historical and aggregated data, serving as a foundation for strategic decision-making. OLAP systems are optimized for executing complex, read-intensive queries that may involve aggregations, slicing, dicing and drill-down operations across multiple dimensions, such as time, geography or category of a product [7]. These systems are often denormalized and rely on multidimensional data models, such as star and snowflake schemas.
A key operational difference is the way each system manages data quality and structure. In OLTP, temporary data imperfections (such as an ongoing update or partially completed transaction) are tolerated, provided the final integrity of the system is preserved. On the opposite, OLAP systems require clean and consolidated data, as the presence of nulls or inconsistencies can compromise the accuracy of aggregated analyses. As such, data cleansing and transformation operations are an integral part of preparing data for OLAP environments [7].
The user profiles for each system also reflect these differences. OLTP tools are

typically used by operational users, such as customer service representatives, sales agents, and clerical staff, who interact with data in real time. OLAP tools, on the other hand, are geared toward analytical users, including business analysts, data scientists, and executives, who rely on historical data to generate reports, perform forecasts, and drive strategic initiatives [19].
OLTP and OLAP systems often coexist within modern enterprises. Data from OLTP systems is extracted, transformed and then loaded into OLAP systems, such as data warehouses or data lakehouses, where it can be leveraged for business intelligence.
Table 2.1 summarizes OLTP vs OLAP systems.

Table 2.1: OLTP vs OLAP systems

Aspect	OLTP	OLAP
Function	Daily management	Decision making support
Orientation	Application base oriented	Subject oriented
Frequency	daily	Scheduled or on demand
Data	Recent and detailed	Aggregate, historical and multidimensional
Access	Read and write	Read only
Users destination	Operational users	Analysts or decision makers
Dimensions	From 100 MB to GB	From 100 GB to TB
Example	Order processing	Business performance reporting

2.4 Cloud and On-Premise solutions

Data warehouses systems can follow two main architectural models: **on-premise** and **cloud-based**. Each approach offers distinct advantages and trade-offs in terms of scalability, control, cost and operational complexity.

2.4.1 On-Premise Solutions

On-premise architectures refer to systems where the hardware, software and networking infrastructure are fully owned, installed and maintained by the organization itself. This model provides the maximum control over data, compliance and customization of the system, which becomes the preferred choice in industries with strict regulatory requirements.
However, on-premise systems involves significant capital expenditure for hardware acquisition and operational expenses for system maintenance, upgrades and staff. Scalability is also inherently limited by the physical infrastructure as it requires the acquisition and installation of new hardware, along with adequate power supply and physical space [10] [24].

2.4.2 Cloud Solutions

On the contrary, **cloud based architectures** has transformed the way organizations build and operate data warehouses. The cloud offers on-demand access to virtually unlimited computational and storage resources, which enable more scalability, availability, reliability and reduces the overhead of managing physical infrastructure.
Cloud providers offer a variety of service models which are categorized by the level of abstraction and customer responsibility:

- **Infrastructure-as-a-Service (IaaS):** are services which provide low-level computing resources such as virtual machines, storage, and networking. Users manage the operating system and applications, offering flexibility at a lower cost, but with higher operational responsibility.
- **Platform-as-a-Service (PaaS):** are services which deliver a higher level of abstraction, supplying managed platforms where users can develop, deploy, and maintain their applications without managing the underlying infrastructure. This model is popular for ETL pipelines and custom analytics workflows.
- **Software-as-a-Service (SaaS):** offers complete applications delivered via the web, typically through subscription models. In this case, the

provider manages the full stack, and users interact directly with the software, without concern for infrastructure or platform management.

Many of the commercial and open-source Data Warehouse platforms now operate within cloud service models. For example:

- **SaaS**: Google Big Query, Amazon Redshift, Snowflake
- **PaaS/IaaS**: Microsoft Azure Synapse Analytics, Apache Hive, ClickHouse

These platforms vary in their level of abstraction and flexibility: SaaS solutions offer ease of use and rapid deployment with minimal configuration, while PaaS and IaaS platforms offer greater customization, typically requiring more setup and operational oversight [6].

Chapter 3

ETL pipeline foundations

3.1 ETL Concepts and Challenges

Given the vast variety of data sources, such as SQL and NO-SQL databases, Excel files and other heterogeneous sources, moving data from a source to a Data Warehouse necessitate a series of operations able to standardize, unify and ensure coherence of the data. These operations are essential to transform raw information into a consolidated and consistent and suitable for analysis dataset.
For this to be achievable, organizations rely on **ETL (Extract, Transform, Load) pipelines**, which define a structured process for data integrations. ETL pipelines contain a series of operations designed in a standardized way such that enables the passage of data from a raw state to a new, unified, consolidated and high quality data state.
The pipeline, as Figure 3.1 shows, consists of three steps [16]:

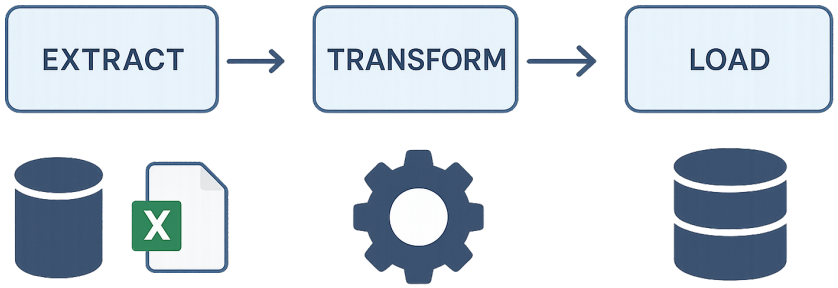

Figure 3.1: ETL procedure

- **Data Extraction**: in this initial step, the goal is to gather data from different sources systems with minimal impact on the source systems itself and preserving data integrity during the transfer.
- **Data Transformation**: the second step consists of transforming the gathered data. Transformations consist of cleaning, normalization, data enrichment and general transformations to match the structure of the target system. Common operations include handling missing values, removing duplicates, converting types, performing aggregations and applying business rules.
- **Data Loading**: in the final step the transformed data is loaded into the target system, the Data Warehouse. This step can be done either in batch or near-real-time.
  - *Batch loading*: this term indicates collecting and loading large volumes of data at scheduled intervals, for example hourly or daily. This is a good approach when data latency is not critical as it introduces a delay between the data being generated and the availability for the analysis.
  - *Near-real-time loading*: differently than the batch loading, this method involves updating the Data Warehouse with small amounts of data in a frequent or continuous way, often within seconds or minutes from the creation of the data. This method is essential when up-to-date information are required, for example for fraud detection systems or operational dashboards.

Despite all the positive features of ETL pipelines, their implementation and maintenance introduce a lot of challenges such as system performance, operational reliability and data quality.
Among the primary difficulties there is the management of heterogeneous data sources, which often differ in format, schema and quality. A simple, yet common, example is the date format: in Europe the format *dd/mm/yyyy* is widely used, while in the United States is more common the format *mm/dd/yyyy*. Additionally, date formats can appear in shorter formats, such as *dd/mm/yy* or more explicit formats, for example *01-January-2025*.
Integrating all of these sources, and formats, can require extensive configuration and custom logic, especially when dealing with unstructured or semi-structured data formats [25].
Errors such as duplicates, missing values or schema mismatches can be introduced during the phases of an ETL pipeline, as a consequence ensuring data

quality and data consistency along the pipeline represent another challenge that need to be addressed as they can lead to incorrect business insights. Data validation and data cleansing becomes essential [16].
Scalability is another challenge, as ETL systems must be designed to handle increasing data throughput, more complex transformations and business logic, without making the system less efficient.
The continuous update of data sources and business requirements requires, in most cases, updates to the ETL logic. As a consequence, maintaining and versioning ETL systems over time, can become time-consuming and prone to errors.

3.2 ETL and ELT

ETL processes, even if considered the classic approach, are not the only option that can be adopted in the world of data integrations. In recent years an alternative has emerged: **ELT processes (Extract, Load, Transform)**, which are increasingly adopted in cloud-based and modern data environments.
In contrast to ETL processes, ELT processes reorder the workflow by performing the transformation phase after the loading has happened as, in this model, the data is extracted and then immediately loaded in the data warehouse in its raw form.
Transformations are executed within the data warehouse itself by leveraging its computational capabilities, especially in cloud-based or parallel processing environments. This step inversion allows ELT processes to be particularly effective when handling large scale or semi structured data [23].

- **ETL processes** are well-suited for on-premise environments and for when data must be validated or enriched before the analytical phase as it offers strict control over the transformations. However, it may become less efficient when data volumes grow significantly.

- **ELT processes** have the benefit of a simplified pipeline architecture and scalability, especially in the cloud environment. More flexibility is allowed thanks to the transformations happening post data load and it also allows the raw data to be used again in future. ELT processes may be less efficient in traditional on-premise systems, where there is the lack of the necessary performance for intensive transformations [23].

A natural question that may arise is regarding how to choose the right approach. The decision should be guided by factors such as data volume, the type of infrastructure available, complexity of the business logic and data

governance requirements. ETL choice remains the preferred one when data needs to be controlled, cleaned and validated before the analysis while ELT processes is a better choice in modern, cloud based architectures, when the workload can be distributed within the data warehouse itself [23].
Table 3.1 shows a recap of the difference between ETL and ELT.

Table 3.1: ETL and ELT confront

Aspect	ETL (Extract, Transform, Load)	ELT (Extract, Load, Transform)
Transformation Location	in external ETL engine	In the Data Warehouse
Ideal Architecture	On-premise	Cloud-based
Data Handling	Data is transformed before loading	Raw data is loaded
Performance	Dependent on external engine resources	Leverages power of scalable DWH engines
Flexibility	Lower as transformations are pre-defined and rigid	Higher as transformations can be delayed and adapted as needed
Latency	Suitable for batch processing	Suitable for near-real-time and large-volume processing
Data Governance	Strong pre-load control and validation	Requires stronger control after data is already loaded
Development Complexity	Requires dedicated ETL tools and transformation logic	Simplifies architecture, but depends on DWH capabilities
Best Suitability	Regulated industries, stable processes, strong pre-load requirements	Dynamic environments, large datasets, data lakehouse architectures

3.3 Company data integration framework

The company Mediamente Consulting has developed a Data Integration framework with the goal to standardize and simplify the development and maintenance of ETL pipelines.
The framework is developed in three main layers, as is shown in Figure 3.2, with the addition of a metadata layer to keep track of the data and the status along the pipeline.

- **Staging area**: to store data from different sources without performing any transformation.
- **Operational data storage**: data is prepared and transformed to be loaded in the data warehouse
- **Publication data layer**: data is finally loaded into the data warehouse and it is ready to be used for analysis.

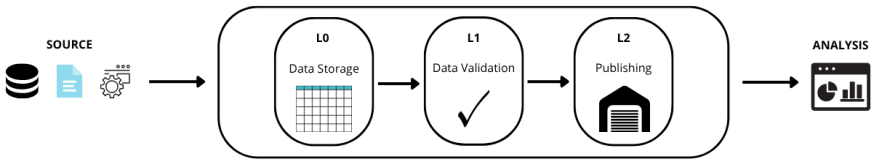

Figure 3.2: ETL framework steps

We are now diving into the details of this framework and see every step.

3.3.1 L0 - Staging Area

The staging area contains a one to one replica of the raw data from the different sources.
Data ingestion happens in batches on a periodical schedule for each table, or object, of the source data inside a **job**. A job is an independent iteration of the load operations, and successive steps, for a single output table. It is managed by the metadata tables, in the metadata layer, called FLOW_MANAGER and TABLE_MANAGER. Each job is identified by a feature called *jobid*, a numeric value with the format *YYYYMMDDHHmmss*.

There are two methods of performing this steps: Full Extraction and Initial Load with Delta Computation.

Full Extraction in this method *all* data is transferred from the source to the target each time. It is easier to perform and it guarantees data completeness but it is not efficient when managing large amount of data.

Initial Load with Delta Computation: this second method consists of a first phase where a first Data Warehouse population happens and then an *incremental* extraction happens: periodic update of the Data Warehouse with capture of the changed data only (new, deleted or updated). More specifically, this method is characterized by the possibility to be performed in two different ways:

- **Minus:** two tables named *STG* and *DLT* are created where:
  - **STG**: table 3.2 shows the schema of a STG table.

Table 3.2: STG table schema

FIELD_1	...	FIELD_N	JOB_ID	INS_TIME

It contains a one to one copy of the batch of read data. There are two additional columns, *JOB_ID* and *INS_TIME*, which represent respectively the id of the actual job in which the batch of data was extracted and the time it was inserted in datetime format.

  - **DLT**: A similar schema to the STG table but with one additional column, as table 3.3 shows.

Table 3.3: DTL table schema

FIELD_1	...	FIELD_N	JOB_ID	INS_TIME	FLG_NEG

This table captures only the change that has happened in the source data, such as added, updated and removed records. To do this the *JOB_ID* column come in help as it is used to get the batch of data of the actual job and the previous one. Once this is done the *Current_DataBatch - Last_DataBatch* and *Last_DataBatch - Current_DataBatch* set operations are computed, getting respectively the new data and the data that was updated or removed.
The *FLG_NEG* column is a numeric field that can contain the value 0 or 1. When a record is inserted for the first time the value 0 is assigned, while when a record is deleted the record is still inserted, in the batch with the current *JOB_ID*, but with value equal to 1.

When a record is updated the old record is inserted, in the current batch of data, with value 1 and the updated one with value 0. Figure 3.3 shows the workflow of the *Minus* operation

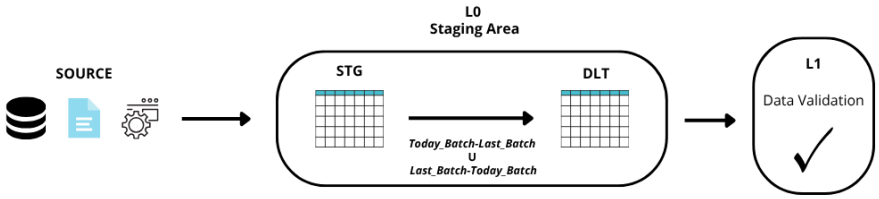

Figure 3.3: L0 with MINUS operation

- **Change Data Capture:** differently with respect to the *Minus*, in the following method only a DLT table is created while the STG table is skipped as a direct copy of the data it is not performed anymore. Source's tables, or objects, are monitored to intercept automatically the delta of data, with respect to the last batch of extracted data. This method *necessitates* the date of update, or a status field, in the source and the owner of the sources must have a log table which keeps track of removed records.
  It is the suggested method when tables with big quantity of data need to be replicated, as it avoids the need to copy data. The only downside is the necessity of the log table as it is not possible to keep track of the removed rows in the Staging Area. Table 3.4 represents the CDC steps, up to the *Operational Data Storage* area.

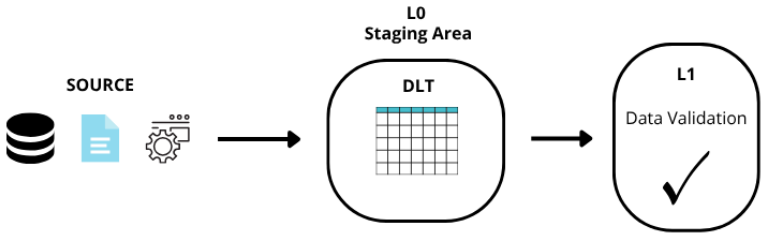

Figure 3.4: L0 with CDC

3.3.2 L1 - Operational Data Storage

The second level represent the core of the data integration framework as it manages complex and computationally intensive procedures of the ETL pro-

cess. The purpose of this phase is to transition the raw data, copied in the previous phase, to a more clean state by performing transformations and checking data quality and business rules. Finally, the L1 is where historical tracking and data harmonization are performed as it enables the integrations of data from heterogeneous systems into a unified model. Figure 3.5 visually shows the steps that this level include, and that we will see in detail.

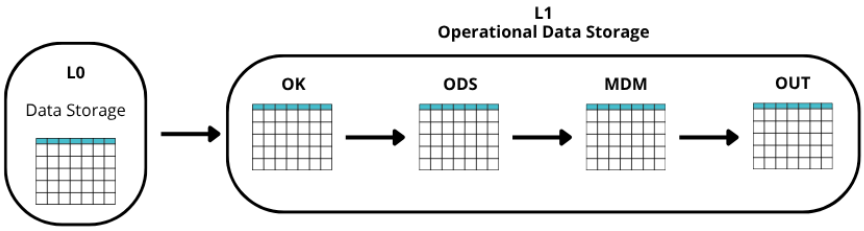

Figure 3.5: L1 - Operational Data Storage - steps

OK Table

The first step of the process consist in the population of the table named *OK*. Data arriving from the DLT table of the previous step, considering only the records where the *JOB_ID* is the most recent one, is subject to data quality controls, such as:

- **Referential Integrity:** for each table that contains at least one foreign key, a join operation is performed between the tables to check relations among tables.
- **Data cleaning:** some transformations are performed to modify the data so that it meets needed requirements. Records having an empty field and the column that cannot contain null values, have the field value in question changed to the default value based on the type of the column [16] and data types are modified to be aligned with the specified one. In addition, general checks are performed to ensure the data is correct, such as integer columns must contain number, duplicates must be removed and anomalies such as a date field containing the date *29/02/2025* should be addressed as the year 2025 is not a leap year.
- **Data Validation:** Data is controlled to be compliant with the requirements needed, such as business rules. For instance, a requirement may be that the *revenue* must lie in the interval $[0, \infty]$.

- **Error Handling:** when records do not respect validation requirements, they are inserted into another table called *E$*. At every iteration the records in this table are re-checked to see if they respect the requirements, if they are met then the rows move to the OK table with the *JOB_ID* of the actual JOB.

ODS Table

ODS represent a table with validated and historicized data. From this point on the table presents a primary key, which was not present in the previous tables thus duplicates were allowed. The population of the table happens with a MERGE sql statement in an incremental update fashion, the possible cases are:

- **Invariant data:** if a record from the OK table is already present in the ODS table then no operation is executed.
- **New records:** if a record from the OK table is not present in the ODS table then the new record from the OK goes into the ODS.
- **Updated records:** if a record from the OK table is already present in the ODS table and a variation of the record from the OK table has been updated then it will update the record in the ODS table.

For historicization to happen two more columns are added: *JOB_ID_UPD* and *UPD_TIME*, of the same format as *JOB_ID* and *INS_TIME*.

MDM Table

Master Data Management (MDM) represents a key step where two main operations are performed: Data Integration and Data Enrichment.

The first operation, **Data integration from different sources** unite together different tables from different sources to generate a unique table, e.g different ODS products table, each one representing the products sold in a specific country are then united together to have a unique product table.

Secondly, **Data Enrichment** is performed on those tables where additional information are added through *join* operations with dimensional tables.
In this step the primary key management is essential, as the join operations are costly. To address this, a *surrogate key* is inserted in this step, which is a unique identifier of a row used for the join operations.

Generally, surrogate keys are defined with compact data types, such as integers, with a growing pattern.

OUT Table

In the final step of the L1, after the integration and enrichment, data are prepared to be published. A table named OUT will contain the data ready for the next step, the L2. **Aggregations operations** and **Denormalization operations** are the main operations in this step .

3.3.3 L2 - Publication Data Layer

This is the final level and it only requires one step, **Publishing**. In this level tables contain data divided into thematic areas, which are usually aligned with business processes. The division of the table will be into:

- **Dimension tables:** tables which contain attributes used to group, browse or constrain data from the fact table.
- **Fact tables:** tables which contain numerical measurements of the business process aligned with. The data is related to each dimension table by a foreign key or a combination of foreign keys.

These tables make up the Data warehouse, which can be configured as:

- **Star schema:** multidimensional representation of data made of a fact table and unnormalized dimension tables.
- **Snowflake schema:** obtained starting from a star schema by normalizing the dimension tables.

Depending on the specific needs of the company one can choose between the two dimensional models as both models have advantages and disadvantages. Though, generally the preferred model is the one that minimizes the number of required joins, the most computationally expensive operation, for analyses.

For an accurate analysis to be performed, the most recent images from the OUT phase must be selected from all the available ones, in the same fashion as in the delta phase.

3.3.4 Metadata Tables

In order to describe and manage metadata, which is descriptive information about the data, an additional layer is present. Company's framework present two metadata tables: FLOW_MANAGER and TABLE_MANAGER.

Flow Manager Table

This table keeps chronological track of ETL iterations, giving detail about the current state and the outcome of each layer for each ingestion flow.

- JOB_ID: Describes ETL iterations instances with a timestamp in the format YYYYMMDDHHmmss.
- IDENTITY: Defines and aggregates flows with a common working area, e.g. *facts* or *dimensions*.
- NUM_LEVEL: Numerical representation of the ETL stage that is performed, i.e. L0, L1 or L2.
- GROUP: This field indicates to which group an ETL instance belongs to, e.g. *customers* or *region*.
- STATUS: Based on the numeric value, it indicates if the state of the current job, as shown in Table 3.4.

Table 3.4: STATUS field values meaning

VALUE	MEANING
0	Job is completed without errors
1	Job is running
2	Job data is ready to be loaded in the next stage
-3	Job is aborted as there are errors

- START_DATE: Indicates, in the data format, the start of the job.
- END_DATE: Indicates, in the data format, the end of the job.

Each record is identified by the primary key composed of the fields *JOB_ID*, *IDENTITY*, *GROUP* and *NUM_LEVEL*.

Table Manager Table

The TABLE_MANAGER table is dedicated to the monitoring of the read operations in the level L0. Each record of this table indicates how many rows have been copied from the source to the STG area for the specific table.
The table presents the following fields:

- IDENTITY: as in the FLOW_MANAGER, it represents flows with a common working area.
- GROUP: Indicates to which group an ETL instance belongs.
- TABLE_NAME: Specifies the name of the table.
- JOBID: Describes ETL iterations instances with a timestamp.
- NUM_ROWS: A numeric value which indicates how many rows have been copied from the sources.
- INS_TIME: Indicates, in date format, the time when the Job has been completed.

the primary key is composed of IDENTITY, JOB_ID, GROUP and TABLE_NAME.

3.4 Overview of Microsoft SQL Server Integration Services (SSIS)

SQL Server Integration Services (SSIS) is a data integration and transformation solution offered by **Microsoft** [18]. This tool is utilized for complex integration problems for organizations, such as copy of data from different sources, cleaning and data extraction based on companies' needs and load them in data warehouses.
The first SSIS version was released along with SQL Server back in 2005 as a substitute to Data Transformation Services (DTS), available with SQL Server 7.0 and SQL Server 2000 [14].
SSIS distinguishes itself from other integration tools for its better integration with Microsoft's ecosystem, which makes the best integration tool for companies which already utilizes SQL Server or other Microsoft technologies.
From a business context, SSIS is an essential tool for automating data management. It can be used to update data warehouses on a daily basis, synchronize data across different systems and perform preliminary data analysis through transformation and data enrichment operations. Its key features include:

- Wide range of built in tasks.
- Graphical tools that provide a visual interface for designing ETL workflows.
- SSIS catalog database for storing and managing packages.

Among the additional advantages that this tool offers there are:

- Easy to use as a low code tool.
- Possibility of integrating custom code.
- Low cost compared to other commercial platforms offering similar services.

The fundamental building block in SSIS is the *package*, which acts as the main operational unit for designing, and executing, ETL processes. A package is a collection of *tasks, control logic, connections* and *workflows* which are organized to extract, transform and load data from one or more sources to a destination. Each package is designed to perform a task, such as copying data, automating recurring processes or executing transformations.
A SSIS package it is typically composed of three principal elements:

- **Control Flow:** Defines the logic sequence of activities in the package. It contains tasks, such as:
  - *Data Flow Tasks:* tasks for data operations.
  - *Execute SQL Tasks:* which allows to execute a specific SQL query or Stored Procedures.
  - *Script Tasks:* which allow to execute personalized code in VB.NET or C#.
- **Data Flow:** this task manages data movement from one or more sources, such as:
  - *OLE DB.*
  - *CSV* or *TXT* files.
  - *XML* and *JSON.*

  to one or more destinations, with same supported types, and data transformations operations, such as:
  - *Data Conversion*: to convert the type of data to be aligned with needs.
  - *Derived Column*: to add specific columns as needed.
  - *Lookup:* to unite data coming from different sources based on a common key.
- **Event Handlers:** allow to manage specific events, such as errors or completion, during the execution of the package.

These packages can be configured with dynamic parameters and monitored through logging tools, providing flexibility and control in enterprise ETL workflows. SSIS offers different tools to handle different scenarios.
Regarding error handling, the *Error Output* enables the redirection of data that does not meet specific requirements to error tables or log files, facilitating analysis and problem resolution.
SQL Server Agent, a service that allows the automatic execution of packages at specific intervals or in response to specific events, manages the automation and scheduling of ETL operations.
ETL development and debugging are carried out using SQL Server Data Tools, an integrated development environment that offers a drag and drop interface for designing workflows and advanced debugging tools.
All of these features make SSIS a robust and efficient solution for managing data integration processes.

Chapter 4

SSIS ETL Pipeline automation

This chapter describes the overall methodology and the implementation of the *SSIS ETL Pipeline automation project*. The chapter first outlines the motivations for this project and its benefits in terms of scalability, maintainability and efficiency. It goes then into details, such as the proposed methodology and technical design, including the metadata files structure and the use of XML templates. In the end, the overall Python code structure for the package generation is presented.

4.1 Motivations for Automation

Within a data driven company the development of ETL processes is a critical but often repetitive task in data warehouse projects.
Usually, in SQL Server Integration Services (SSIS) the ETL workflow is designed manually, while this approach provides flexibility and full control to the developer, it also introduces several challenges:

- Designing packages by hand is *highly* time consuming, especially when the data warehouse must integrate data from multiple sources.
- Each package has to be individually configured with connections, transformations and error-handling logic, which often mean repeating the same development patterns across multiple packages [11].
- Manual repetition can delay delivery and also increase the risk of human

error and inconsistency in implementation, especially when multiple developers work on the same package [21].

Automation of the process addresses these issues by letting the repetitive manual design be generated by a metadata driven structure. Instead of having developers recreate the same package structures over and over again, a program can interpret structured metadata and automatically produce SSIS packages that are consistent with predefined standards [25]. This approach reduces the overall development time, ensures consistency across packages and makes the ETL process more scalable.
Errors that arise from manual configuration are minimized while updates and maintenance are simplified since modification to the metadata can be propagated automatically to regenerated packages [7].

An important motivation for automation in this project arises from the context in which it was carried out. As explained in the previous chapter, Mediamente Consulting company has developed a standardized framework for the ETL processes. This framework ensures that SSIS packages share a common structure and follow strict best practices, but it also means that the design of each package is repetitive. In this context, the benefits of automation are amplified as a metadata driven approach allows SSIS packages to be generated automatically according to the company's framework, ensuring that best practices and standards are followed while drastically reducing the manual development effort.

Ultimately, automating the creation of SSIS packages allows the organization to accelerate data warehouse development cycles, improving reliability and free developers from routine tasks so they can focus on activities of higher value, such as optimization and business logic design [8].

4.2 Methodology and Technical Design

In this thesis, the proposed solution is based on a metadata driven approach. Figure 4.1 shows the overall structure of the process, which relies on structured metadata that describes the source and target systems and mappings. A Python script interprets this metadata and generates fully functional SSIS packages in XML format, using predefined templates compliant with company's framework.
More specifically, as Figure 4.1 shows, the first step is metadata ingestion by a Python automation script, which captures the all the metadata necessary

for the generation of the packages, the next steps.

Secondly, the SSIS package responsible for the initial staging layer, the L0 level, is generated.

Thirdly, the system creates a package designed to handle an exception case: having transactions in the dimension table that have no reference in the respective dimension tables.

Lastly, the system generates the SSIS package that manages the OK and ODS tables within the Operational data management, the L1 level.

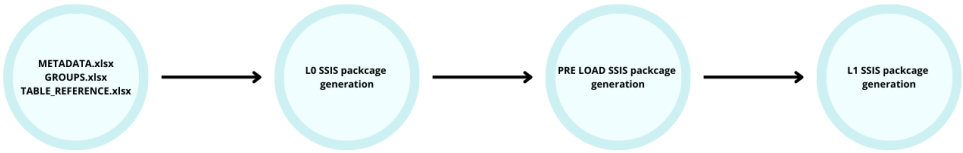

Figure 4.1: Steps for automated ETL develop

4.2.1 Excel Metadata File Structure

The metadata provided in the Excel files is crucial for the automation framework. Instead of relying on developers to repeatedly configure SSIS packages, they have to spend time just to correctly configure these three files which capture all relevant information.

METADATA.xlsx

This file defines the structure of the tables from the sources and the tables found in the level L0 and L1. Each Excel page represent a single table and its tables in the framework.
As the Figure 4.2 and 4.4 shows, the schema of this Excel file consists of:

- **Source table** column with table name defined as:

  [schema].[SRC_*table name*]

  The *schema* is the logical structure that defines how data is organized in a database, including tables, relationships, and constraints, while *table name* is simply the name of the table in the source. All the columns

of the table in question are registered along with the type of the data and the length if the type is of string type. This last Excel column is needed only to be able to automatically generate the query to create the necessary tables.

- **STG table** column with table name:

  [L0].[STG_*table name*]

  L0 represents the schema of the company where the tables regarding the staging layer are saved. We have the addition of the columns *JOBID* and *INS_TIME*.

- **DLT table** where in addition to the previous columns we have the *FLG_NEG* column. This table represent the end of the L0 stage.

  [L0].[DLT_*table name*]

- **OK table** column with table name:

  [L1].[OK_*table name*]

  The schema *L1* represent the schema containing all the information about the operational data storage layer. All the previous columns are presented but the data that will be inserted here are only the one that have passed referential and business logic checks.

- **ODS table**, column with table name:

  [L1].[ODS_*table name*]

  In addition to the previous table columns we have a change in the base columns. Since in this step the primary key is inserted, we are no longer allowed to have, as in the previous steps, duplicates rows, and as a consequence the columns: *JOBID_INS*, *INS_TIME*, *JOBID_UPD* and *UPD_TIME* are inserted to keep track of the updates to specific columns and when they have happened instead of just *JOBID* and *INS_TIME*.

- **KEY column** which will indicate, with a *yes*, if the source column is a primary key and an empty string if it is not.

- **NOT NULL column** which will indicate, as in the previous case, with a *yes* if the source column cannot present the value null.

- **DEFAULT VALUE column** that will contain, if present, the default value of the specified source column.

[mago].[SRC_MA_Items]		
Field	Field Format	Field Length

[L0].[STG_MA_Items]		
Field	Field Format	Field Length
JOBID	numeric	
INS_TIME	datetime	

[L0].[DLT_MA_Items]		
Field	Field Format	Field Length
JOBID	numeric	
INS_TIME	datetime	
FLG_NEG	int	

Figure 4.2: Schema describing tables in Excel metadata file

[L1].[OK_MA_Items]		
Field	Field Format	Field Length
JOBID	numeric	
INS_TIME	datetime	
FLG_NEG	int	

[L1].[ODS_MA_Items]					
Field	Field Format	Field Length	KEY	NOT NULL	DEFAULT VALUE
JOBID_L1_INS	numeric				
INS_TIME	datetime			yes	GETDATE()
FLG_NEG	int			yes	0
JOBID_L1_UPD	numeric			yes	0
UPD_TIME	datetime				GETDATE()

Figure 4.3: Schema describing tables in Excel metadata file

GROUPS.xlsx

Since all tables belonging to the same group must be in the same SSIS package, this file is also extremely important as it contains the information about the group of each table.
Additionally, in the Excel file there is presented an additional column for the control of the METADATA Excel file.
The Excel schema, shown in Figure 4.4, consist of:

- **TABLE_NAME column** where the source table name is written.
- **GROUP column** which contains the group to which the table belongs.
- **IDENTITY column** which will contain either ANAGRAFICHE or MOVIMENTI to indicate if the table in consideration is a fact table or a dimension table.

TABLE_NAME	GROUP	IDENTITY

Figure 4.4: Schema describing the Groups Excel File

TABLE_REFERENCES.xlsx

This table contains the referential relationships between tables in the data warehouse. Specifically, it lists the columns that are used to reference each table. By explicitly documenting the foreign key constraints, the file provides the necessary metadata for resolving data dependencies during the ETL process.
The principal purpose of this file is to support the Pre Load step of the automation framework. During ETL, it is common to encounter the case where a record is a fact table references a key that does not exist in the associated dimension table. Such problems may arise due to missing or delayed dimension updates, leading to referential integrity issues and potential ETL failures [16]. As a consequence, it would be wrong to directly remove the records with no matching key.
The Excel schema, represented in Figure 4.5, is composed as follows:

- **TABLE_NAME column** as in the previous Excel file, this column contains the table names.
- **GROUP column** will contain the group to which a table belongs to.
- **IDENTITY column** which will identify if the source table is part of ANAGRAFICHE or MOVIMENTI.
- **To_Table_Reference column** which contains the name of the table to which is referencing to.
- **Key_Reference column** which will contain a list of the columns that are in common between the two tables that are referencing to each other.

TABLE_NAME	IDENTITY	GROUP	REFERENCE_TO	THROUGH_FK

Figure 4.5: Schema describing the Excel file containing references between tables

4.2.2 SSIS XML Package and Templates

As figure 4.1 shows, after the ingestion of the metadata files, the generation of the package is next. This generation is allowed thanks to SSIS packages being XML documents with a rigid internal structure that make them ideal for a templated based generation.

XML is a markup language, a system for embedding tags and notations within text or data to describe its structure, which allow the representation of hierarchical data in a human readable but processable by machine format. XML format allows for flexible and self describing structures, which makes it suitable for metadata management, configuration files and communication between heterogeneous systems.
XML documents are composed of elements, attributes and textual content that are organized in a similar structure as tree. Each element is enclosed within an opening and a closing tag, with attributes written inside to provide additional information about the elements. The design, which is hierarchical, allows XML to model complex datasets, including, for example, nested relationships, optional fields and repeated structures which can define the structure of a dataset. Listing 4.1 shows an example of a simple XML code describing a table *Customers* and its columns.

```
<Table name="Customers">
    <Column name="CustomerID" type="int" />
    <Column name="FirstName" type="string" />
    <Column name="LastName" type="string" />
    <Column name="Email" type="string" />
</Table>
```

Listing 4.1: Example of a simple XML table definition

In this project, reusable XML templates were designed to automate the creation of SSIS packages in such a way to align with the standardized inte-

gration framework of the company.
These templates are used as *blueprints* for the package structure as they include placeholders for values, in this way Python scripts dynamically insert the appropriate values taken from the Excel file ingested in the first phase. Each object of a package was treated individually for simplicity and put together at the end inside a unique XML file.

We are now going to see each XML template used for the generation of all the packages.

Package generation XML Template

Listing 4.2 shows the main XML template which defines the structure of a SSIS package. This template contains the essential information required for a package to function, such as the *creation date* of the package, *creator computer name* and *creator username*.
During the package generation, it is used as last XML template since it contains the placeholders for all of the major components of a package. More specifically:

- *<DTS:PackageParameters>* indicated the start of the XML code for the package parameters
- *<DTS:Variables>* is the starting point for the XML code for the variables of the package.
- *<DTS:Executables>* indicates the start of the XML code for all the tasks inside the package, such as SQL tasks, Excecute other packages, Data Flow tasks and so on.
- *<DTS:PrecedenceConstraints>* tag is the start of the XML code that will indicate which objects will be linked at package scope.

Additionally, the package itself and almost every SSIS object requires a **DTSID**, which is effectively a **UUID** (Universally Unique Identifier), a 128-bit identifier that guarantees uniqueness across systems and environments, ensuring that no two objects are accidentally assigned the same identifier.

This design provides reusability and flexibility, allowing to generate a wide variety of different SSIS packages.

```
<?xml version="1.0"?>
<DTS:Executable xmlns:DTS="www.microsoft.com/SqlServer/Dts"
  DTS:refId="Package"
  DTS:CreationDate="{creation_info["creation_date"]}"
  DTS:CreationName="Microsoft.Package"
  DTS:CreatorComputerName="{creation_info["creator_computer_name"]}"
  DTS:CreatorName="{creation_info["creator_name"]}"
  DTS:DTSID="{{{dtsid_project}}}"
  DTS:ExecutableType="Microsoft.Package"
  DTS:LastModifiedProductVersion="16.0.5685.0"
  DTS:LocaleID="1033"
  DTS:ObjectName="{package_name}"
  DTS:PackageType="5"
  DTS:VersionBuild="{version_build}"
  DTS:VersionGUID="{{{dtsid_project}}}">
  <DTS:Property DTS:Name="PackageFormatVersion">8</DTS:Property>
  <DTS:PackageParameters>
    {package_parameters}
  </DTS:PackageParameters>

  <DTS:Variables>
    {package_variables}
  </DTS:Variables>

  <DTS:Executables>
    {package_tasks}
  </DTS:Executables>

  <DTS:PrecedenceConstraints>
    {package_links}
  </DTS:PrecedenceConstraints>

</DTS:Executable>
```

Listing 4.2: SSIS Package XML Template

Package Parameter and Variable XML

Listing 4.3 and Listing 4.4 describes the XML code to create, respectively, a package parameter and a variable. These code fragments are not standalone as they must be inserted into the placeholders defined in the template of Listing 4.2.
Specifically, package parameters are placed within the {*package_parameters*} section, while variables are inserted into the {*package_variables*} section.

It is important to notice that in SSIS there is an evident difference between variables and parameters.

- **Parameters** are used in SSIS to provide external configurability to a package. They allow values to be assigned at the moment of execution or deployment and, once execution begins, parameters are read only. Listing 4.3 shows the XML template, with placeholder for Python, to generate a single parameter in a determinated SSIS package. Several attributes are worth noting, such as:
  - *DTS:PackageParameter* marks the start of the parameter definition, which will go in the specific placeholder in the Listing 4.2.
  - *DTS:DataType* specifies the type of the parameter, such as String, Int32, or Boolean and it is dynamically inserted.
  - *DTS:DTSID* is automatically generated and guarantees uniqueness of the parameter inside the package.
  - *DTS:ObjectName* gives the parameter its logical name, which can be used to reference it within the package.
  - Inside the nested <*DTS:Property*> tag, the default value that will be used if no external value is indicated.
- **Variables** are used to store internal values that support the execution logic of a package. Variables can hold temporary results or any information that tasks in the package may need. Differently from parameters, variables are mutable, as a consequence their value can change dynamically during the execution of the package. Listing 4.4 shows us the XML template to create a variable in the package, with some important points:
  - *DTS:Variable* marks the start of the variable definition, which will go in the specific placeholder in the Listing 4.2.
  - Like parameters, variables also have a *DTS:DTSID* to uniquely identify them.
  - The *DTS:Namespace="User"* attribute indicates that this is a user-defined variable, as opposed to system variables that SSIS provides by default.
  - *DTS:ObjectName* specifies the variable's name, which will be used to reference it in tasks or expressions.
  - The <*DTS:VariableValue*> tag defines the initial value of the variable, with its own *DTS:DataType* attribute.

– Additionally, when a variable is configured as an expression, the template includes *DTS:Expression*, allowing its value to be calculated dynamically at runtime rather than being fixed.

```
<DTS:PackageParameter
DTS:CreationName="Microsoft.PackageParameter"
DTS:DataType="{data_type}"
DTS:DTSID="{{{generate_uuid4()}}}"
DTS:ObjectName="{object_name}">
    <DTS:Property
        DTS:Name="ParameterValue"
        DTS:DataType="{data_type}">{default_value}</DTS:Property>
</DTS:PackageParameter>
```

Listing 4.3: SSIS XML Package Parameter

```
<DTS:Variable
DTS:DTSID="{{{generate_uuid4()}}}"
{eval_expression if is_expression==1 else ""}
{DTS:Expression="{expression}" if is_expression==1 else ""}
DTS:IncludeInDebugDump="6789"
DTS:Namespace="User"
DTS:ObjectName="{variable_name}">
<DTS:VariableValue
    DTS:DataType="{variable_data_type}">{variable_value}</DTS:
        VariableValue>
</DTS:Variable>
```

Listing 4.4: SSIS XML Package Variable

SQL Task XML

In SSIS, an **Execute SQL Task** is used to run SQL queries against a relational database, allowing data definition, data manipulation and store procedure to be executed within a package.
There are two different approaches to define the SQL statement that will be executed. The first approach is to write directly the query within the task configuration inside the Execute SQL Task component. The second one is to pass, to the Execute SQL Task component, the SQL query as a variable, making it more flexible. The second approach is more versatile because the query is assigned at runtime, meaning that the same SQL statement can be different depending on the execution context. For example a variable can store the name of a target table and the SQL task can then execute a TRUNCATE TABLE statement on that specific table.
The listing 4.5 presents the XML template of the SQL Task used in this project.
Several attributes are particularly important:

- *<DTS:Executable>* marks the start of the task definition, which will go inside the <DTS:Executables>in the main XML code presented in Listing 4.2.
- *DTS:refId* uniquely identifies the task inside the package by combining the package name with the task name.
- *DTS:DTSID* is a globally unique identifier generated for the task.
- *<DTS:ObjectData>* is a block that holds the details of the SQL statement that will be executed. More specifically, the "SQLTask:SqlStatementSource" holds the actual SQL query that will be run.
  In the template in Listing 4.5 it is represented by the placeholder {sql_task_statement}.
- If the task is expected to return a single row, the {single_row_result_set} option can be included. This ensures that the output is handled properly when mapped to SSIS variables.
- In Listing 4.5, the {parameters} placeholder indicates where input or output parameters for the SQL statement can be defined, allowing flexible interaction between the task and other components of the package.

```
<DTS:Executable
DTS:refId="Package\\{sql_task_name}"
DTS:CreationName="Microsoft.ExecuteSQLTask"
DTS:Description="Write in the flow manager the start of the L0"
DTS:DTSID="{{{generate_uuid4()}}}"
DTS:ExecutableType="Microsoft.ExecuteSQLTask"
DTS:LocaleID="-1"
DTS:ObjectName="{object_name}"
DTS:TaskContact="{description}; Microsoft Corporation; SQL Server
    2022;  2022 Microsoft Corporation; All Rights Reserved;http://
    www.microsoft.com/sql/support/default.asp;1"
    DTS:ThreadHint="0">
    <DTS:Variables />
    <DTS:ObjectData>
    <SQLTask:SqlTaskData
        SQLTask:SqlStatementSource="{sql_task_statement}"
        {single_row_result_set if single_result_set==1 else ""}
            xmlns:SQLTask="www.microsoft.com/sqlserver/dts/tasks/
            sqltask">

        {parameters}

    </SQLTask:SqlTaskData>

    </DTS:ObjectData>
</DTS:Executable>
```

Listing 4.5: SSIS XML Package SQL Task

Object Links XML

The execution flow between tasks is controlled by *precedence constraints*, which are represented visually as arrows connecting tasks. These links determines both order of execution and conditions needed to be met to run another task. For example, a task may be executed only if the previous task succeeds, fails or completes regardless of the outcome.
Listing 4.4 shows the XML templates used in this project to connect each task:

- *<DTS:PrecedenceConstraint>* marks the start of the definition of a precedence constraint and it will go and replace, in Listing 4.2, the {package_links} placeholder.
- *DTS:refId* uniquely identifies the constraint within the package by combining the package reference with the constraint name. In this way you select the scope where the object linking is happening, for example at package or sequence container scope.
- *DTS:DTSID*, as in previous cases, provides a globally unique identifier for the constraint.
- *DTS:From* and *DTS:To* specify the link between two tasks. More specifically, DTS:From indicates the task whose completion triggers the next task while DTS:To identifies the task that will be executed if the previous task is triggered.
- In Listing 4.6, optional error handling can be specified with {error_link if is_error==1 else ""}. This allows the constraint to represent an error path, allowing the case where a target task is executed when the source task fails.

```
<DTS:PrecedenceConstraint
DTS:refId="Package{ref}.PrecedenceConstraints[{constraint_name}]"
DTS:CreationName=""
DTS:DTSID="{{{generate_uuid4()}}}"
DTS:From="Package\\{from_name}"
DTS:LogicalAnd="True"
DTS:ObjectName="{constraint_name}"
DTS:To="Package\\{to_name}"
{error_link if is_error==1 else ""} />
```

Listing 4.6: SSIS XML Package Variable

Expression Task XML

An **Expression Task** is used to assign, at runtime, a value to a specific variable or property through the use of expressions, which are formulas that can perform calculations, evaluate conditions or concatenate strings. For example, in this project the expression task is used to compute the value of the JOBID in the current job and then assign it to the variable. The formula to compute this is:

$$
\begin{aligned}
&@[User :: JOBID] = \\
&((DT\_I8)YEAR(GETDATE())) * 10000 * 1000000 \\
&+MONTH(GETDATE()) * 100000000 \\
&+DAY(GETDATE()) * 1000000 \\
&+DATEPART("HH", GETDATE()) * 10000 \\
&+DATEPART("MI", GETDATE()) * 100 \\
&+DATEPART("SS", GETDATE())
\end{aligned}
$$

Where:

- YEAR(GETDATE()) extracts the current year, converted to a 64-bit integer (in SSIS is identified by DT_I8). It is then multiplied by 10000*1000000 to shift it into the highest digits of the JOBID.
- MONTH(GETDATE()) is multiplied by 100000000, placing it immediately after the year.
- DAY(GETDATE()) is multiplied by 1000000, placing it after the month.
- DATEPART("HH", GETDATE()) provides the current hour, multiplied by 10000.
- DATEPART("MI", GETDATE()) provides the current minute, multiplied by 100.
- DATEPART("SS", GETDATE()) provides the current second, which fills the last two digits.

The resulting JOBID is a concatenation of the date and time in the format: YYYYMMDDHHMMSS. For example, considering the data 17/09/2025 at hour 09:00:00 the result will be: *20250917090000*.

Listing 4.7 represent the XML template to generate an expression task, with important attributes such as:

- *<DTS:Executable>* marks the start of the task definition, which will go inside the *<DTS:Executables>* in the main XML code presented in Listing 4.2.
- *DTS:refId* uniquely identifies the constraint within the package
- *DTS:DTSID* which provides a globally unique identifier for the object in the package.
- *<DTS:ObjectData>* tag which contain the expression to compute the JOBID.

```
<DTS:Executable
DTS:refId="Package\\{name}"
DTS:CreationName="Microsoft.ExpressionTask"
DTS:Description="Expression Task"
DTS:DTSID="{{{generate_uuid4()}}}"
DTS:ExecutableType="Microsoft.ExpressionTask"
DTS:LocaleID="-1"
DTS:ObjectName="{name}"
DTS:TaskContact="Expression Task;Microsoft Corporation; SQL Server
    2022;  2022 Microsoft Corporation; All Rights Reserved;http
    ://www.microsoft.com/sql/support/default.asp;1"
DTS:ThreadHint="0">
    <DTS:Variables />
    <DTS:ObjectData>
        <ExpressionTask
        Expression="{get_current_jobid}" />
    </DTS:ObjectData>
</DTS:Executable>
```

Listing 4.7: SSIS XML Package Expression Task

Sequence Container XML

Sequence Containers are used to group tasks into logical units. Organizing tasks within containers allow developers to manage complex workflows in a more efficient way and allow to apply constraints or transactions to multiple tasks at one.
It is important to understand that sequence containers do not perform transformations themselves, they act as structural elements that improve clarity and control within a SSIS Package.
Listing 4.8 shows the XML templates used in this project to create a sequence container. The key attributes of the XML code are:

- As in previous explained objects, <*DTS:Executable*> marks the start of the task definition, which will go inside the <*DTS:Executables*> in the Listing 4.2.
- *DTS:refId* uniquely identifies the constraint within the package
- *DTS:DTSID*, as in previous cases, provides a globally unique identifier for the constraint.
- <*DTS:Executables*> marks where all tasks that are grouped inside the container will be placed. The placeholder {container_tasks} is replaced at runtime with the actual XML definitions of other tasks that will belong to the container.
- <*DTS:PrecedenceConstraints*> defines the execution flow between the tasks inside the container. In the Listing 4.8 the placeholder {link_block} is dynamically replaced with the XML that specifies how tasks are linked.

```
<DTS:Executable
DTS:refId="Package\\{group_name}"
DTS:CreationName="STOCK:SEQUENCE"
DTS:Description="Contain all tables referring to a specific group"
DTS:DTSID="{{{generate_uuid4()}}}"
DTS:ExecutableType="STOCK:SEQUENCE"
DTS:LocaleID="-1"
DTS:ObjectName="{object_name}">
<DTS:Variables />

<DTS:Executables>
    {container_tasks}
</DTS:Executables>
```

```
<DTS:PrecedenceConstraints>
    {link_block}
</DTS:PrecedenceConstraints>

</DTS:Executable>
```

Listing 4.8: SSIS XML Package Sequence Container

Data Flow Task XML

The core component of SSIS for performing ETL operations is the **Data Flow Task**. This task is responsible for the movement and transformation of data. Inside a data flow tasks, developers must define pipelines consisting of elements called *components*, such as *Source connection*, *Destination connection* and *Derived column*.
Components work together to perform operations such as: extract and transform data, apply business logics and load data.
Listing 4.9 represents the XML template used in this project to generate data flow tasks according to needs. In the XML code elements of particular importance are:

- *<DTS:Executable>*, as in previous examples, marks the start of the task definition, which will go inside the *<DTS:Executables>* in the Listing 4.2.
- *DTS:refId* uniquely identifies the constraint within the package
- *DTS:DTSID* provides a globally unique identifier for the constraint.
- The tag <pipeline>encapsulates the data flow logic, including all components and paths for data movement.
- Inside the tag *<pipeline>* the tag *<components>* contains the definitions of all pipeline components, such as Source, Destination, Derived Column, Lookup, or Aggregate components. The {components} placeholder is dynamically replaced with the actual XML of the components for the specific data flow.
- *<paths>* tag defines the connections between components. The {links} placeholder represents the logical paths linking outputs of one component to inputs of another, effectively building the ETL pipeline.

```
<DTS:Executable
DTS:refId="Package\\{group_name}\\{table_name}\\{
    dataflow_task_name}"
DTS:CreationName="Microsoft.Pipeline"
DTS:Description="{description}"
DTS:DTSID="{{{generate_uuid4()}}}"
DTS:ExecutableType="Microsoft.Pipeline"
DTS:LocaleID="-1"
DTS:ObjectName="{dataflow_task_name}"
DTS:TaskContact="Performs high-performance data extraction,
    transformation and loading;Microsoft Corporation; Microsoft
    SQL Server; (C) Microsoft Corporation; All Rights Reserved;
    http://www.microsoft.com/sql/support/default.asp;1">
<DTS:Variables />
    <DTS:ObjectData>
        <pipeline version="1">
        <components>
            {components}
        </components>

        <paths>
            {links}
        </paths>
        </pipeline>
    </DTS:ObjectData>
</DTS:Executable>
```

Listing 4.9: SSIS XML Package Data Flow Task

4.3 Python Code Structure and Modules

To allow the creation of the automated framework generator several tools were utilized. At the core of the development of the automated framework for SSIS package generation is **Python**, a programming language that allows the flexibility and power to manipulate XML structures and easily integrate external metadata sources.
In addition to Python, as previously stated, **Excel** and **SQL Server Integration Services (SSIS)** were utilized to allow the creation of this project. This section describes the role of these supporting tools, with particular focus on the main tool: Python.

4.3.1 Base tools and Libraries

For the creation of this project only the use of **Pandas** library was necessary. Pandas is a well known Python library used for data analysis and manipulation. Originally created by Wes McKinney in 2008, it has become a cornerstone of the Python data ecosystem and is widely used in fields such as data science, machine learning and business intelligence.
The core Pandas' data structures that were utilized in this project are:

- **Series**: one dimensional labeled array that is capable of holding data of any type.
- **DataFrames**: two dimensional and tabular data structure with labeled axes, similar to a spreadsheet.

Pandas was utilized to facilitate the manipulation of metadata stored in the Excel files. Dataframes provided advantages as it allowed the metadata to be managed in a tabular format, similar to the Excel spreadsheet. This allowed for efficient filtering, grouping and other operations when generating SSIS packages.
In addition to this the **uuid**, **platform** and **datetime** libraries were utilized.

- **uuid**: as previously stated, some SSIS objects need an uuid to be generated. This library allow to access to a specific function that allow to generate an uuid.
- **platform**: is a library that allows us to get information about the user, such as computer name and creator name. This information are needed in the creation of a SSIS package.
- **datetime** this library is needed to get the time in which the SSIS package is created.

4.3.2 L0 Package Generation

After the ingestion of data from the Excel files, the following step, illustrated in Figure 4.1, is the generation of SSIS packages responsible for populating the tables in the staging layer of the company's data integration framework. Utilizing information stored in the METADATA.xlsx and GROUPS.xlsx files, the algorithm generated SSIS packages with the naming convention 001_L0_IDENTITY_GroupName, where IDENTITY and GroupName are both dynamically changed. Each package corresponds to a specific group of source tables and they ensure that data from heterogeneous systems are systematically ingested into the staging area.
The pseudo code for this generation process is presented in Algorithm 1.

Algorithm 1: Pseudo code for automated L0 package generation

```
Input: group_name, tables_list
Output: L0 SSIS package (XML)
Generate package parameters and base variables;
Generate start/end procedures for group_name;
foreach table in tables_list do
    Generate task to retrieve last read date;
    Generate STG dataflow task (copy source → STG);
    Generate DLT dataflow task (minus-based comparison);
    Generate table manager SQL task;
    Build execution links: last_date_read → STG → DLT →
     table_manager;
    Create a table container including tasks and links;
    Append table container to group container;
    Build table-specific variables (columns, minus query, table name);
    Append variables to package variables;
end
Generate group-level variable (group name);
Build package-level links: start_procedure → group_container →
 end_procedures;
Assemble package with: parameters, variables, tasks, links;
Save final package as L0_IDENTITY_group_name.dtsx;
```

More specifically:

1. The basic variables of the package are generated, such as the variable holding the group name and the variables holding the status value for the procedures, as explained it chapter 4.

2. The **Start Procedure**, **End Procedure OK** and **End Procedure**

Error SQL Tasks are generated. These objects will be unique per group, which means unique inside the package. Their use is to write in the FLOW MANAGER the start of the ETL process, end of the process if it is successful and end of the process if it is unsuccessful. The status variables created are used in this step.

3. A **Last Date Read** SQL task is generated for each table in the group. This SQL task retrieves the JOBID of the last ETL process that has been successfully completed and stores it in a package variable.

4. For each table in the group, two Data Flow Tasks are generated. The first Data Flow Task transfers data from the source systems into the corresponding STG table. This step is straightforward, as the data is simply copied from the source into the staging layer without transformations.
   The second task is more complex and is responsible for identifying changes in the data and propagating them into the DLT table. This is achieved by performing set operations between two different snapshots of the staging table: one corresponding to the most recent load and one to the previous load. The SQL query used for this comparison is shown in Listing 4.10. Two subqueries are used:

   - The first identifies deleted records, i.e., rows present in the previous load but not in the latest one. These records are marked with the flag FLG_NEG = 1.
   - The second identifies newly inserted records, i.e., rows present in the latest load but not in the previous one. These are flagged with FLG_NEG = 0.

   An EXCEPT operator, which returns rows present in one dataset but not in the other, is utilized in both subqueries.

   The two results are combined with a UNION ALL and a unified dataset of changes is obtained, which is then written into the DLT table.

```
SELECT {joined_columns}, FLG_NEG
FROM (
    SELECT 1 AS FLG_NEG, {joined_MINUS_NEG_cols}
    FROM (
        SELECT {joined_stg2_cols}
        FROM [L0].STG_{table_name} stg2
        WHERE stg2.JOBID = '" + (DT_WSTR, 20)@[User::
            LAST_DATE_READ] + "'

        EXCEPT

        SELECT {joined_stg1_cols}
        FROM [L0].STG_{table_name} stg1
        WHERE stg1.JOBID = '" + (DT_WSTR, 20)@[$Package::JOBID
            ] + "'
    ) MINUS_NEG

    UNION ALL

    SELECT 0 AS FLG_NEG, {joined_MINUS_POS_cols}
    FROM (
        SELECT {joined_stg2_cols}
        FROM [L0].STG_{table_name} stg2
        WHERE stg2.JOBID = '" + (DT_WSTR, 20)@[$Package::JOBID
            ] + "'

        EXCEPT

        SELECT {joined_stg1_cols}
        FROM [L0].STG_{table_name} stg1
        WHERE stg1.JOBID = '" + (DT_WSTR, 20)@[User::
            LAST_DATE_READ] + "'
    ) MINUS_POS
) UNION_1;
```

Listing 4.10: Minus-based comparison query for STG tables

5. For each table, an Execute SQL Task **Table Manager** is created. This task executes a stored procedure which write inside the TABLE_MANAGER the number of records that are copied from the source systems to the STG table.

6. Next, the links between all these previously generated tasks are built and inserted in a table dedicated sequence container. Additionally, variables

specific to the table are created and inserted at a package level scope.

7. Once all the XML for the single tables are generated, the links at the package scope are created and all the different elements are put together, inside the XML seen in Listing 4.2, to then generate the final XML and saved in a specific output folder.

Figure 4.6 present the visual result of the Algorithm 1, i.e. how a SSIS Package looks like, after being generated using the Python script, inside the tool SSIS.

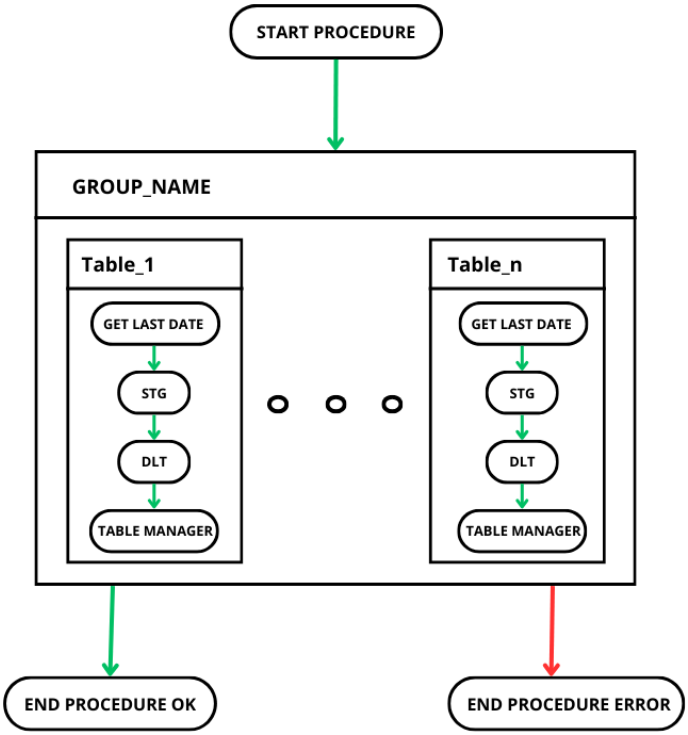

Figure 4.6: L0 SSIS Package Design

4.3.3 PRE LOAD Package Generation

Recalling the Figure 4.1, the third step is the generation of the PRE LOAD Package, which addresses the issue where records in a fact table have no referential match in a dimension table is generated, for each group. This situation can occur when transactional data references dimension attributes that are missing or not yet available in the system.
More specifically, this problem is solved by *preloading* these unmatched records into the dimension's ODS table. To achieve this, the algorithm extracts the distinct primary keys from the DLT table using the most recent JOBID. These keys are first inserted into a dedicated PRE LOAD table of the dimension in question. Since the related descriptive attributes are unavailable, only the primary key values are inserted, while all non-key attributes are initialized with default values according to their data type.
Algorithm 2 shows the steps to generate the package:

Algorithm 2: Pseudo code for PRE_LOAD SSIS Package Generation

```
Input: Grouped tables (MOVIMENTI with related
       ANAGRAFICHE), output folder
Output: PRE_LOAD SSIS packages
foreach group in grouped_tables do
    Extract group name;
    foreach table in group do
        Extract MOVIMENTI table name;
        Get related ANAGRAFICHE tables;
        foreach anagrafiche table do
            Generate referential check SQL query and add to package
             variables;
            Generate Execute SQL Task to truncate PRE_LOAD table;
            Generate Dataflow Task for PRE_LOAD population ( DLT
             → PRE_LOAD);
            Generate MERGE query;
            Generate Execute SQL Task to merge data into ODS table;
            Build execution links: PRE_LOAD Dataflow Task →
             Execute SQL Task Merge;
            Build small container for this ANAGRAFICHE table;
            Append small container to container task lists;
        end
    end
    Build medium container combining truncate tasks and small
     containers;
    Build execution links: truncate PRE_LOAD → medium container;
    Build big container combining all medium containers;
    Generate group-level variable;
    Generate last date read dlt variable;
    Assemble package with: parameters, variables, tasks, links;
    Generate final SSIS package XML;
    Save final package as L1_PRE_LOAD_group_name.dtsx;
end
```

More specifically, the algorithm to generate such package is divided as follows:

1. From the TABLE_REFERENCES.xlsx file the group name is extracted along with the division of the MOVIMENTI and ANAGRAFICHE tables.

2. A referential check query is generated in order to retrieve the distinct foreign key values that will later be used in the MERGE operation. As the query in Listing 4.11 shows, it applies a ranking function to identify the most recent records from the DLT table, based on the primary key columns, ordered by `JOBID DESC` and `FLG_NEG ASC`. Only the top ranked (rn=1) records are then selected, as it guarantees that are the most recent one. The query is stored in a variable, and its placeholders are dynamically replaced during execution by Python. @[User::LAST_DATE_READ_DLT_group] is a package variable responsible for retrieving the most recent JOBID in the DLT table.

```
WITH TopRanked AS (
    SELECT {joined_columns}, JOBID, INS_TIME, FLG_NEG,
           ROW_NUMBER() OVER(
               PARTITION BY {joined_pk_columns}
               ORDER BY JOBID DESC, FLG_NEG ASC
           ) AS rn
    FROM [L0].DLT_{table_name}
    WHERE JOBID >= @[User::LAST_DATE_READ_DLT_{group}]
)
SELECT DISTINCT({TR_joined_foreign_keys})
FROM TopRanked TR
WHERE TR.rn = 1;
```

Listing 4.11: Top-ranked query for DLT tables

3. An Execute SQL Task is then created to truncate the PRE_LOAD table. This ensures that the table is cleared before each iteration of the framework.

4. A Data Flow Task to populate the PRE LOAD table is then created. Using the SQL query previously generated, stored in a variable, the PRE LOAD table is populated with the distinct foreign key values obtained.

5. A second Execute SQL Task is created to execute the MERGE query shown in Listing 4.12, which inserts records into the ODS table only when no matching record already exists. As a consequence, the primary key values of the dimension tables are kept, while all non key columns are initialized with their respective default values.

```
MERGE INTO [L1].[ODS_{anag_table_name}] AS target
USING [L1].[PRE_LOAD_{anag_table_name}] AS source
    {fk_control_part_expression}
WHEN NOT MATCHED THEN
```

```
INSERT ({joined_columns}, JOBID_l1_ins, INS_TIME, FLG_NEG
    , jobid_l1_upd, upd_time)
VALUES ({joined_values_cols_list}, source.[JOBID],
    GETDATE(), source.FLG_NEG, source.[JOBID], GETDATE())
    ;
```

Listing 4.12: Merge query for loading PRE_LOAD into ODS

6. Once the tasks are defined, the execution links are created and the tasks are grouped into containers for each dimension table. These are then combined at higher levels into medium and big containers, which ensures clarity in the execution flow.

7. Lastly, all tasks, containers, variables, and parameters are put together into a complete package. The resulting SSIS package is then translated into the XML structure shown in Listing 4.2 and saved in the designated output directory.

Figure 4.7 shows the visual result of the Algorithm 2: how a SSIS Package looks like, after being generated using the Python script, inside the tool SSIS.

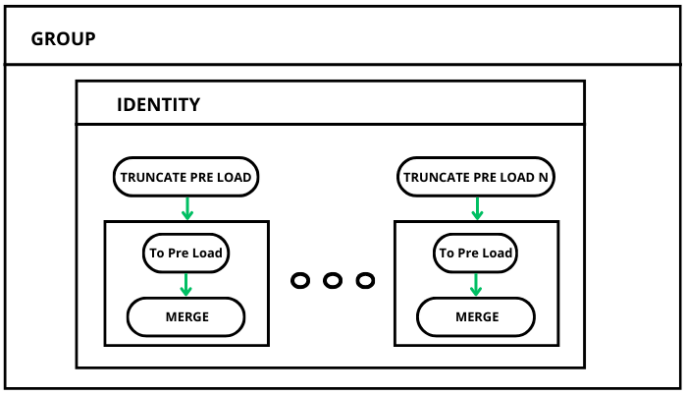

Figure 4.7: PRE LOAD SSIS Package Design

4.3.4 L1 Package Generation

Last step, as Figure 4.1 shows, is to generate the SSIS packages responsible for the L1 layer.
Unlike the complete L1 level, used across the company, the automation developed in this project was limited until the creation of the ODS tables as going beyond this point would have been inefficient given specific business requirements.
Algorithm 3 presents the algorithm used to generate the SSIS packages for this step.

Algorithm 3: Pseudo code for automated L1 package generation

```
Input: group_name, tables_list
Output: L1 SSIS package (XML)
Generate package parameters and base variables;
Generate start/end procedures for group_name;
Generate Last Date Read Dlt SQL query;;
foreach table in tables_list do
    Generate table specific OK SQL query;
    Generate Truncate_OK Execute SQL task to truncate OK table;
    Generate OK Dataflow Task (move data DLT → OK);
    Generate MERGE_ODS Sql query and Execute SQL Task (merge
     OK → ODS);
    Build execution links: Truncate_OK → OK Dataflow Task →
     MERGE_ODS;
    Create a table container including tasks and links;
    Append table container to group container;
    Build table-specific variables;
    Append variables to package variables;
end
Generate group-level variable (group name);
Build package-level links: start_procedure → group_container →
 end_procedures;
Assemble package with: parameters, variables, tasks, links;
Save final package as L1_IDENTITY_group_name.dtsx;
```

More specifically:

1. Package parameters and basic variables are initialized. Among these, STATUS variables are defined to record the outcome of the L1 execution in the FLOW_MANAGER table.

2. Same as in the L0 step, the Start Procedure, End Procedure OK, and

End Procedure ERROR are created for each group to manage the execution flow and ensure error handling by writing in the FLOW_MANAGER the result.

3. The query **Last Date Read DLT** is generated. This query is stored inside a variable and its purpose is to retrieve the most recent JOBID from the DLT table.

4. For each table, the query handling the movement and checks of data from the DLT table to the OK table is created. The query, shown in Listing 4.13, retrieves only the most recent records through the use of the Last Date Read DLT variable (@[User::LAST_DATE_READ_DLT] in the Listing 4.13) from the DLT table using a ranking function. The ranking function orders records by the primary key columns, sorts them by JOBID in descending order and FLG_NEG in ascending order, this ensures that the row assigned rank 1 corresponds to the most recent valid record.

```
SELECT {joined_columns}, JOBID,
       COALESCE(FLG_NEG, 0) AS FLG_NEG,
       COALESCE(INS_TIME, 'GETDATE()') AS INS_TIME
FROM (
    SELECT {joined_columns}, JOBID, INS_TIME, FLG_NEG,
          ROW_NUMBER() OVER(
               PARTITION BY {joined_pk_columns}
               ORDER BY JOBID DESC, FLG_NEG ASC
          ) AS rn
    FROM [LO].DLT_{table_name}
    WHERE JOBID > " + @[User::LAST_DATE_READ_DLT] + "
) RankedRows
WHERE RankedRows.rn = 1;
```

Listing 4.13: Query to retrieve ranked rows from DLT table

5. Before loading new data, an Execute SQL Task to truncate the OK table is generated. This guarantees that the OK table is empty at the start of each iteration of the process.

6. A Data Flow Task is then generated to move the data, using the filtering query previously generated, from the DLT to the OK table.

7. The Execute SQL Task containing a MERGE SQL query, represented in Listing 4.14 is created. This query is responsible for synchronizing the ODS table with the data in the OK table: existing records are updated, and new records are inserted.

```
MERGE INTO [L1].[ODS_{table_name}] AS target
USING [L1].[OK_{table_name}] AS source
    {pk_control_part_expression}

WHEN MATCHED THEN
    UPDATE SET
        {joined_target_col_eq_source_col},
        target.JOBID_l1_upd = source.JOBID,
        target.FLG_NEG = source.FLG_NEG,
        target.UPD_TIME = GETDATE()

WHEN NOT MATCHED THEN
    INSERT ({joined_columns}, JOBID_l1_ins, INS_TIME, FLG_NEG
        , upd_time, jobid_l1_upd)
    VALUES ({joined_source_cols}, source.[JOBID], GETDATE(),
        source.[FLG_NEG], GETDATE(), source.[JOBID]);
```

Listing 4.14: MERGE query for loading data into ODS table

8. Once the SQL and Data Flow Tasks are generated, execution links are established between them (Truncate OK → Data Flow Task → MERGE ODS). These are grouped into a dedicated table container that encapsulates all steps for a single table.

9. Finally, the containers for each table are appended to the group container, package-level links are created (Start Procedure → Group Container → End Procedures), and all components are assembled into the final SSIS package. The result is stored as an XML file named 010_L1_IDENTITY_GroupName.dtsx.

Figure 4.8 presents the visual result of the Algorithm 3, i.e. how a SSIS Package looks like, after being generated using the Python script, inside the tool SSIS.

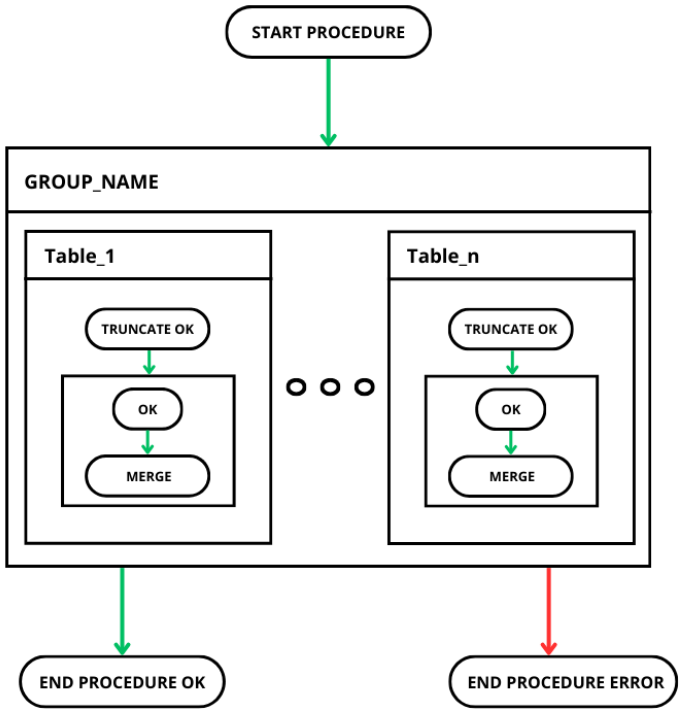

Figure 4.8: L1 SSIS Package Design

Chapter 5

Experimental Evaluation

In this chapter the results of the execution of generated packages is going to be shown. In the first part the set up of the example will be introduced, along with how the Excel files are completed and how the data looks in source tables. After, the creation of the package will be performed and then the results of the execution of those packages will be provided, along with an explanation of the results.

5.1 Validation Setup

5.1.1 Source Data

For the example provided, the source data is composed of 6 tables:

MA_Customer: stores information about the customers who make purchases. It is composed of the following columns:

- *CustomerID*: is an integer which uniquely identify the customer. It is the primary key.
- *CustomerName*: is a varchar which contains name and surname of the customer
- *Region*: is the region where the customer is from.
- *Segment*: the segment to which the customer is part of.

MA_Date: serves as a date dimension, providing temporal details such as the calendar date, weekday, month, and year.

- *DateID*: is an integer which uniquely identify the date. It is the primary key.
- *Date*: is a datetime which contains the date.
- *Weekday*: it is a varchar which indicates the day of the week.
- *Month*: it is a varchar which indicates the month of the year.
- *Year*: it is an integer which indicates the year.

MA_Inventory: records product stock levels for each store on specific dates. It links together stores, products, and dates, providing insights into how inventory changes over time and across locations

- *InventoryID*: is an integer which identify the inventory. It is part of the composed primary key.
- *StoreID*: is an integer which identify the store. It is part of the composed primary key.
- *DateID*: is an integer which identify the date. It is part of the composed primary key.
- *ProductID*: is an integer which identify the product. It is part of the composed primary key.
- *StockLevel*: it is an integer which represent the quantity in stock.

MA_Product: contains details about the products being sold. It is used to analyze sales and inventory by product type

- *ProductID*: is an integer which uniquely identify the product. It is the primary key.
- *ProductName*: is a nvarchar which contains the name of the product.
- *Category*: it is a varchar which indicates the category the product belongs to.
- *Brand*: it is a nvarchar which indicates the brand of the product.

MA_Sales: represents individual sales transactions.

- *SaleID*: is an integer which identify the sale id. It is part of the composed primary key.

- *CustomerID*: is an integer which identify the customer who purchased. It is part of the composed primary key.
- *ProductID*: is an integer which identify the date of the purchase. It is part of the composed primary key.
- *SaleDate*: is an integer which identify the purchase date. It is part of the composed primary key.
- *Amount*: it is an integer which represent the amount of the purchase.

MA_Stores: describes the stores where inventory is kept and sales occur.

- *StoreID*: is an integer which uniquely identify the store. It is the primary key.
- *StoreName*: is a nvarchar which contains the name of the store.
- *Location*: it is a varchar which indicates the location the store.
- *Manager*: it is a varchar which indicates the name of the manager of the store.

It is important to clarify how they are related to one another, the data model follows a star schema design, where **MA_Sales** and **MA_Inventory** act as fact tables, while the remaining tables serve as dimension tables.

5.1.2 Excel data

As explained in the previous chapters, it is extremely important to fill correctly the Excel files in question.

Figure 5.1 and 5.2 shows the METADATA Excel sheet for the table: MA_Customer.

[mago].[SRC_MA_Customer]			[L0].[STG_MA_Customer]			[L0].[DLT_MA_Customer]		
Field	Field Format	Field Length	Field	Field Format	Field Length	Field	Field Format	Field Length
CustomerID	int		CustomerID	int		CustomerID	int	
CustomerName	varchar	50	CustomerName	varchar	50	CustomerNa	varchar	50
Region	varchar	30	Region	varchar	30	Region	varchar	30
Segment	varchar	20	Segment	varchar	20	Segment	varchar	20
			JOBID	numeric		JOBID	numeric	
			INS_TIME	datetime		INS_TIME	datetime	
						FLG_NEG	int	

Figure 5.1: MA_Customer Source, STG and DLT tables.

[L1].[OK_MA_Customer]			[L1].[ODS_MA_Customer]					
Field	Field Format	Field Length	Field	Field Format	Field Length	KEY	NOT NULL	DEFAULT VALUE
CustomerID	int		CustomerID	int		yes		
CustomerName	varchar	50	CustomerName	varchar	50			
Region	varchar	30	Region	varchar	30			
Segment	varchar	20	Segment	varchar	20			
JOBID	numeric		JOBID_L1_INS	numeric				
INS_TIME	datetime		INS_TIME	datetime			yes	GETDATE()
FLG_NEG	int		FLG_NEG	int			yes	0
			JOBID_L1_UPD	numeric			yes	0
			UPD_TIME	datetime				GETDATE()

Figure 5.2: MA_Customer OK, ODS tables and information.

Figure 5.3 and 5.4 shows the METADATA Excel sheet for the table: MA_Date.

[mago].[SRC_MA_Date]			[L0].[STG_MA_Date]			[L0].[DLT_MA_Date]		
Field	Field Format	Field Length	Field	Field Format	Field Length	Field	Field Format	Field Length
DateID	int		DateID	int		DateID	int	
Date	datetime		Date	datetime		Date	datetime	
Weekday	varchar	20	Weekday	varchar	20	Weekday	varchar	20
Month	varchar	20	Month	varchar	20	Month	varchar	20
Year	int		Year	int		Year	int	
			JOBID	numeric		JOBID	numeric	
			INS_TIME	datetime		INS_TIME	datetime	
						FLG_NEG	int	

Figure 5.3: MA_Date Source, STG and DLT tables.

[L1].[OK_MA_Date]			[L1].[ODS_MA_Date]					
Field	Field Format	Field Length	Field	Field Format	Field Length	KEY	NOT NULL	DEFAULT VALUE
DateID	int		DateID	int		yes		
Date	datetime		Date	datetime				
Weekday	varchar	20	Weekday	varchar	20			
Month	varchar	20	Month	varchar	20			
Year	int		Year	int				
JOBID	numeric		JOBID_L1_INS	numeric				
INS_TIME	datetime		INS_TIME	datetime			yes	GETDATE()
FLG_NEG	int		FLG_NEG	int			yes	0
			JOBID_L1_UPD	numeric			yes	0
			UPD_TIME	datetime				GETDATE()

Figure 5.4: MA_Date OK, ODS tables and information.

Figure 5.5 and 5.6 shows the METADATA Excel sheet for the table:

MA_Product.

[mago].[SRC_MA_Product]				[L0].[STG_MA_Product]			[L0].[DLT_MA_Product]		
Field	Field Format	Field Length	Field	Field Format	Field Length	Field	Field Format	Field Length
ProductID	int		ProductID	int		ProductID	int	
ProductName	nvarchar	50	ProductName	nvarchar	50	ProductName	nvarchar	50
Category	varchar	50	Category	varchar	50	Category	varchar	50
Brand	nvarchar	50	Brand	nvarchar	50	Brand	nvarchar	50
			JOBID	numeric		JOBID	numeric	
			INS_TIME	datetime		INS_TIME	datetime	
						FLG_NEG	int	

Figure 5.5: MA_Product Source, STG and DLT tables.

[L1].[OK_MA_Product]			[L1].[ODS_MA_Product]					
Field	Field Format	Field Length	Field	Field Format	Field Length	KEY	NOT NULL	DEFAULT VALUE
ProductID	int		ProductID	int		yes		
ProductName	nvarchar	50	ProductName	nvarchar	50			
Category	varchar	50	Category	varchar	50			
Brand	nvarchar	50	Brand	nvarchar	50			
JOBID	numeric		JOBID_L1_INS	numeric				
INS_TIME	datetime		INS_TIME	datetime			yes	GETDATE()
FLG_NEG	int		FLG_NEG	int			yes	0
			JOBID_L1_UPD	numeric			yes	0
			UPD_TIME	datetime				GETDATE()

Figure 5.6: MA_Product OK, ODS tables and information.

Figure 5.7 and 5.8 shows the METADATA Excel sheet for the table: MA_Store.

[mago].[SRC_MA_Store]			[L0].[STG_MA_Store]			[L0].[DLT_MA_Store]		
Field	Field Format	Field Length	Field	Field Format	Field Length	Field	Field Format	Field Length
StoreID	int		StoreID	int		StoreID	int	
StoreName	nvarchar	50	StoreName	nvarchar	50	StoreName	nvarchar	50
Location	varchar	50	Location	varchar	50	Location	varchar	50
Manager	varchar	50	Manager	varchar	50	Manager	varchar	50
			JOBID	numeric		JOBID	numeric	
			INS_TIME	datetime		INS_TIME	datetime	
						FLG_NEG	int	

Figure 5.7: MA_Store Source, STG and DLT tables.

[L1].[OK_MA_Store]			[L1].[ODS_MA_Store]					
Field	**Field Format**	**Field Length**	**Field**	**Field Format**	**Field Length**	**KEY**	**NOT NULL**	**DEFAULT VALUE**
StoreID	int		StoreID	int		yes		
StoreName	nvarchar	50	StoreName	nvarchar	50			
Location	varchar	50	Location	varchar	50			
Manager	varchar	50	Manager	varchar	50			
JOBID	numeric		JOBID_L1_INS	numeric				
INS_TIME	datetime		INS_TIME	datetime			yes	GETDATE()
FLG_NEG	int		FLG_NEG	int			yes	0
			JOBID_L1_UPD	numeric			yes	0
			UPD_TIME	datetime				GETDATE()

Figure 5.8: MA_Store OK, ODS tables and information.

Figure 5.9 and 5.10 shows the METADATA Excel sheet for the table: MA_Inventory.

[mago].[SRC_MA_Inventory]			[L0].[STG_MA_Inventory]			[L0].[DLT_MA_Inventory]		
Field	**Field Format**	**Field Length**	**Field**	**Field Format**	**Field Length**	**Field**	**Field Format**	**Field Length**
InventoryID	int		InventoryID	int		InventoryID	int	
StoreID	int		StoreID	int		StoreID	int	
DateID	int		DateID	int		DateID	int	
ProductID	int		ProductID	int		ProductID	int	
StockLevel	int		StockLevel	int		StockLevel	int	
			JOBID	numeric		JOBID	numeric	
			INS_TIME	datetime		INS_TIME	datetime	
						FLG_NEG	int	

Figure 5.9: MA_Inventory Source, STG and DLT tables.

[L1].[OK_MA_Inventory]			[L1].[ODS_MA_Inventory]					
Field	**Field Format**	**Field Length**	**Field**	**Field Format**	**Field Length**	**KEY**	**NOT NULL**	**DEFAULT VALUE**
InventoryID	int		InventoryID	int		yes		
StoreID	int		StoreID	int		yes		
DateID	int		DateID	int		yes		
ProductID	int		ProductID	int		yes		
StockLevel	int		StockLevel	int				
JOBID	numeric		JOBID_L1_INS	numeric				
INS_TIME	datetime		INS_TIME	datetime			yes	GETDATE()
FLG_NEG	int		FLG_NEG	int			yes	0
			JOBID_L1_UPD	numeric			yes	0
			UPD_TIME	datetime				GETDATE()

Figure 5.10: MA_Inventory OK, ODS tables and information.

Figure 5.11 and 5.12 shows the METADATA Excel sheet for the table: MA_Sales.

[mago].[SRC_MA_Sales]			[L0].[STG_MA_Sales]			[L0].[DLT_MA_Sales]		
Field	**Field Format**	**Field Length**	**Field**	**Field Format**	**Field Length**	**Field**	**Field Format**	**Field Length**
SaleID	int		SaleID	int		SaleID	int	
CustomerID	int		CustomerID	int		CustomerID	int	
ProductID	int		ProductID	int		ProductID	int	
SaleDate	datetime		SaleDate	datetime		SaleDate	datetime	
Amount	float		Amount	float		Amount	float	
			JOBID	numeric		JOBID	numeric	
			INS_TIME	datetime		INS_TIME	datetime	
						FLG_NEG	int	

Figure 5.11: MA_Sales Source, STG and DLT tables.

[L1].[OK_MA_Sales]			[L1].[ODS_MA_Sales]					
Field	**Field Format**	**Field Length**	**Field**	**Field Format**	**Field Length**	**KEY**	**NOT NULL**	**DEFAULT VALUE**
SaleID	int		SaleID	int		yes		
CustomerID	int		CustomerID	int		yes		
ProductID	int		ProductID	int		yes		
SaleDate	datetime		SaleDate	datetime				
Amount	float		Amount	float				
JOBID	numeric		JOBID_L1_INS	numeric				
INS_TIME	datetime		INS_TIME	datetime			yes	GETDATE()
FLG_NEG	int		FLG_NEG	int			yes	0
			JOBID_L1_UPD	numeric			yes	0
			UPD_TIME	datetime				GETDATE()

Figure 5.12: MA_Sales OK, ODS tables and information.

Additionally, Figures 5.13 and 5.14 show the GROUPS Excel file and the TABLE_REFERENCES Excel file completed.

TABLE_NAME	GROUP	IDENTITY
MA_Customer	CustomerDomain	ANAGRAFICHE
MA_Product	CustomerDomain	ANAGRAFICHE
MA_Store	StoreOperations	ANAGRAFICHE
MA_Date	StoreOperations	ANAGRAFICHE
MA_Sales	SalesTransactions	MOVIMENTI
MA_Inventory	InvetoryTracking	MOVIMENTI

Figure 5.13: Excel file containing information about table's group.

TABLE_NAME	GROUP	IDENTITY	To_Table_Reference	Key_Reference
MA_Sales	SalesTransactions	MOVIMENTI	MA_Customer	CustomerID
MA_Sales	SalesTransactions	MOVIMENTI	MA_Product	ProductID
MA_Inventory	InventoryTracking	MOVIMENTI	MA_Store	StoreID
MA_Inventory	InventoryTracking	MOVIMENTI	MA_Date	DateID
MA_Inventory	InventoryTracking	MOVIMENTI	MA_Product	ProductID

Figure 5.14: Excel file containing information about relations between tables.

5.1.3 SSIS Packages

The step that follows is the SSIS packages generation. Before the packages representing the ETL processes, three packages are created to simplify the workflow of the project.
The first one is the **MAIN** package, represented in Figure 5.15, which is the package that will be executed to execute all the other packages as it is designed to execute two other packages and compute the JOBID to pass it to the child packages.

The second one is the **MAIN_L0**, represented in Figure 5.16, executes the packages responsible for the staging area, the L0 level.

The last one is the **MAIN_L1**, shown in Figure 5.17, executes the packages responsible for the population of the OK and ODS tables, along with the execution of the package responsible for the PRE LOAD method.

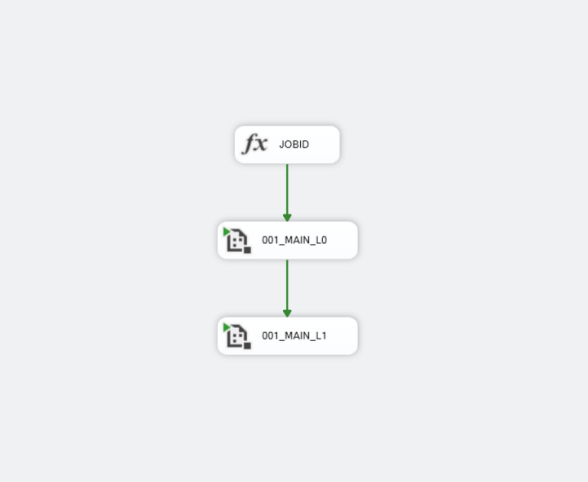

Figure 5.15: Main Package.

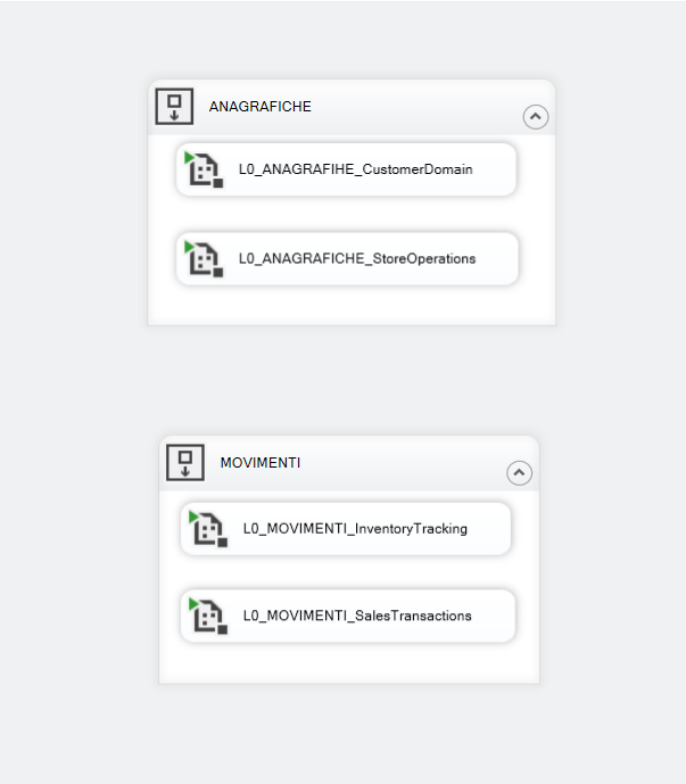

Figure 5.16: MAIN_L0 package.

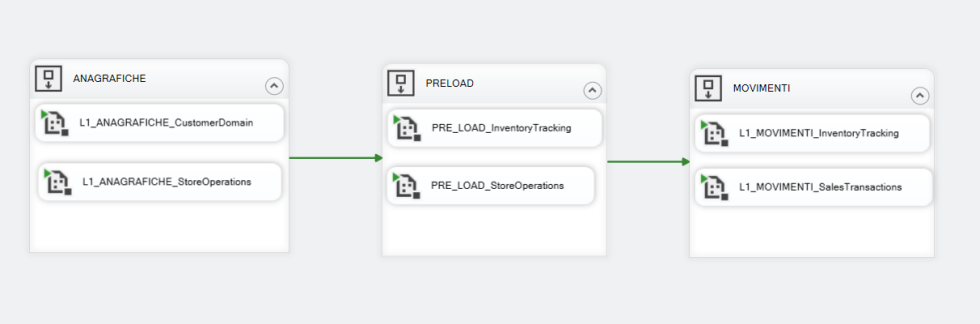

Figure 5.17: MAIN_L1 package.

For the sake of simplicity and avoiding repetitions, as all packages have the same structure, only the SSIS package of the CustomerDomain group, is going to be shown.

L0 level - Staging Area Package

Figure 5.20 represent the visual interface of the SSIS package *001_ANAGRAFICHE_CustomerDomain*. Starting from the top and going to the bottom:

1. **START PROCEDURE**: is the *Execute Sql Task* which writes into the FLOW_MANAGER the start of the procedure, keeping track of the process.

2. **Group Sequence container**: the bigger sequence container Customer-Domain is responsible for grouping the other two sequence container, each one relative to the table belonging to the group.

3. **Table Sequence Container**: groups the tasks responsible for the execution of the L0 level for each table.

4. **LAST DATE READ**: is a *Execute Sql Task* which read the last JOBID from the FLOW_MANAGER and assign it to a package variable.

5. **STG**: is a *Data Flow Task* responsible for the copy of the data from the source, Figure 5.18 shows how it is composed. *SOURCE* is a connection task, responsible for connecting to the source, *Colonna derivata* is a task which will add to the data just copied the JOBID and the INS_TIME columns while *STG* is a connection task to the STG table where the data is going to be stored.

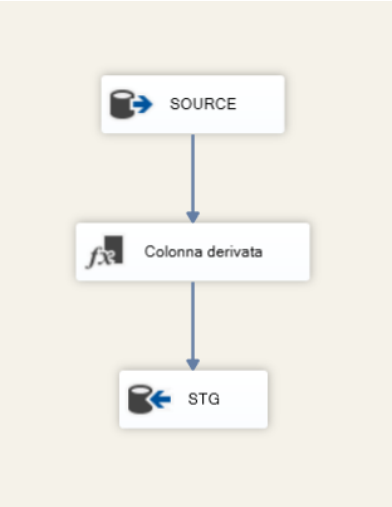

Figure 5.18: How the Data Flow Task is made in the STG level.

6. **DLT**: is a *Data Flow Task* which performs the set operations, discussed in the previous chapters, to identify the new, updated and deleted data from the STG table and move it to the DLT table. Figure 5.19 shows how the task is composed: *MINUS* which is a connection task to the STG table where the data is read with the SQL query shows in Listing 4.10, *Colonna derivata* that is task to add the JOBID, INS_TIME and the FLG_NEG columns and *DLT*, the connection to the DLT table.

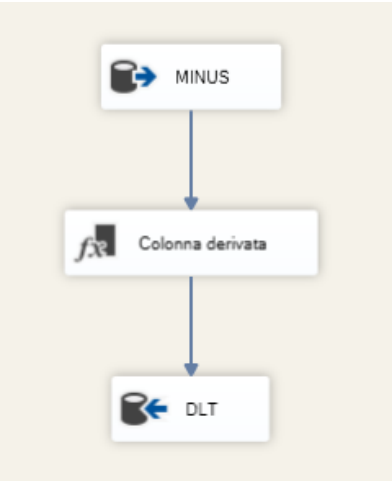

Figure 5.19: How the Data Flow Task is made in the DLT level.

7. **TABLE MANAGER**: is a *Execute Sql Task* responsible to track how many rows are copied from the sources.

8. **END PROCEDURE OK**: this *Execute Sql Task* updates the record inserted in the FLOW_MANAGER adding the fact that the process has ended successfully.

9. **END PROCEDURE ERROR**: this *Execute Sql Task* updates the record inserted in the FLOW_MANAGER adding the fact that the process has ended unsuccessfully.

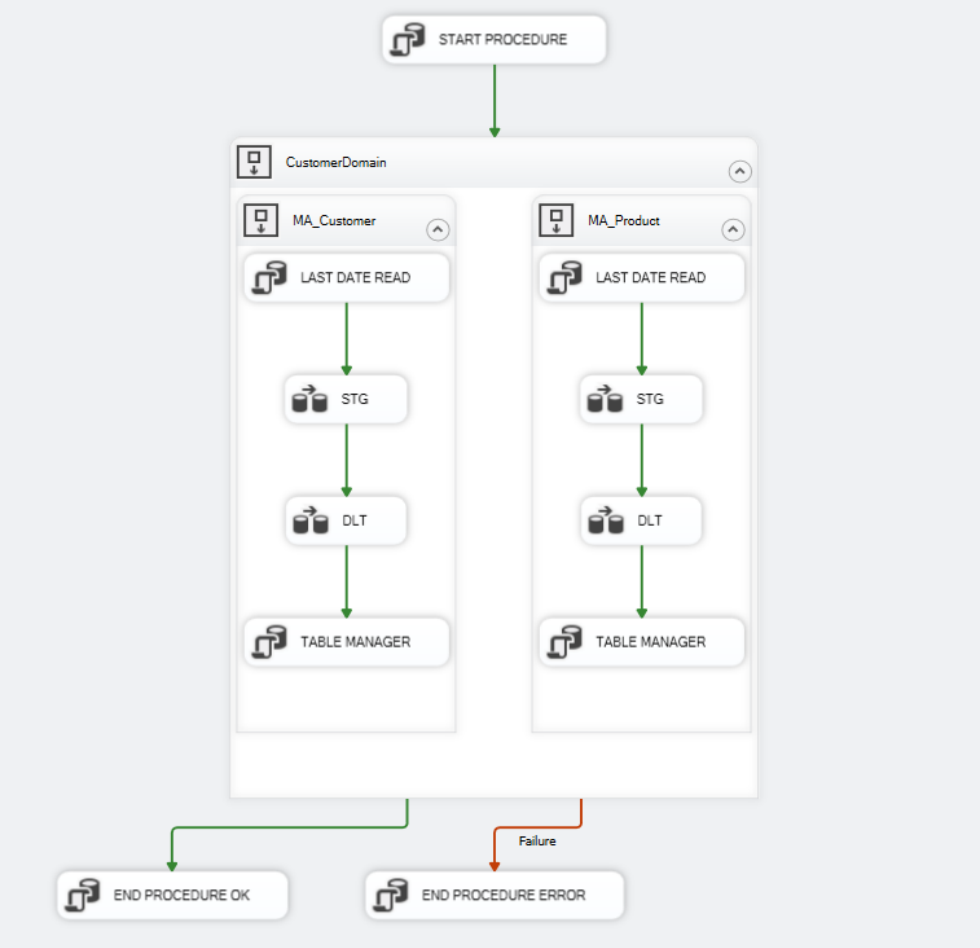

Figure 5.20: Package responsible for the execution of the L0 level.

L1 level - staging area Package

Figure 5.22 represent the next step, the package *001_L1_ANAGRAFICHE_CustomerDomain*, responsible for the population of the OK and ODS tables in the L1 level.
From the top to the bottom of the package we have:

- **START PROCEDURE**: is a *Execute Sql Task* which writes into the FLOW_MANAGER the start of the procedure, the difference with respect to the previous package is that the NUM_LEVEL column is set to 1.
- **Group Sequence container**: the bigger sequence container *CustomerDomain* is responsible for grouping the other sequence container, each one relative to the operations performed on a table belonging to the group.
- **Table Sequence Container**: groups the tasks responsible for the execution of the operations of the L1 level for each table.
- **Truncate OK**: is a *Execute Sql Task* which truncates the OK table.
- **OK**: is a *Data Flow Task* responsible for collecting the latest data from the DLT and copy it to the OK table. Figure 5.21 shows the tasks inside the Data Flow Task: a connection to the DLT table where data is gathered with the SQL query shown in Listing 4.13 and a connection to the OK table to move the data to the OK table.

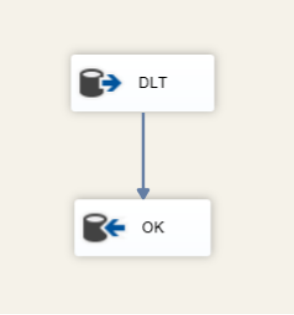

Figure 5.21: OK Data Flow Task.

- **ODS**: is an *Execute Sql Task* responsible for the MERGE SQL query, shown in Listing 4.14.

- **END PROCEDURE OK**: this *Execute Sql Task* updates the record inserted in the FLOW_MANAGER adding the fact that the process has ended successfully.
- **END PROCEDURE ERROR**: this *Execute Sql Task* updates the record inserted in the FLOW_MANAGER adding the fact that the process has ended unsuccessfully.

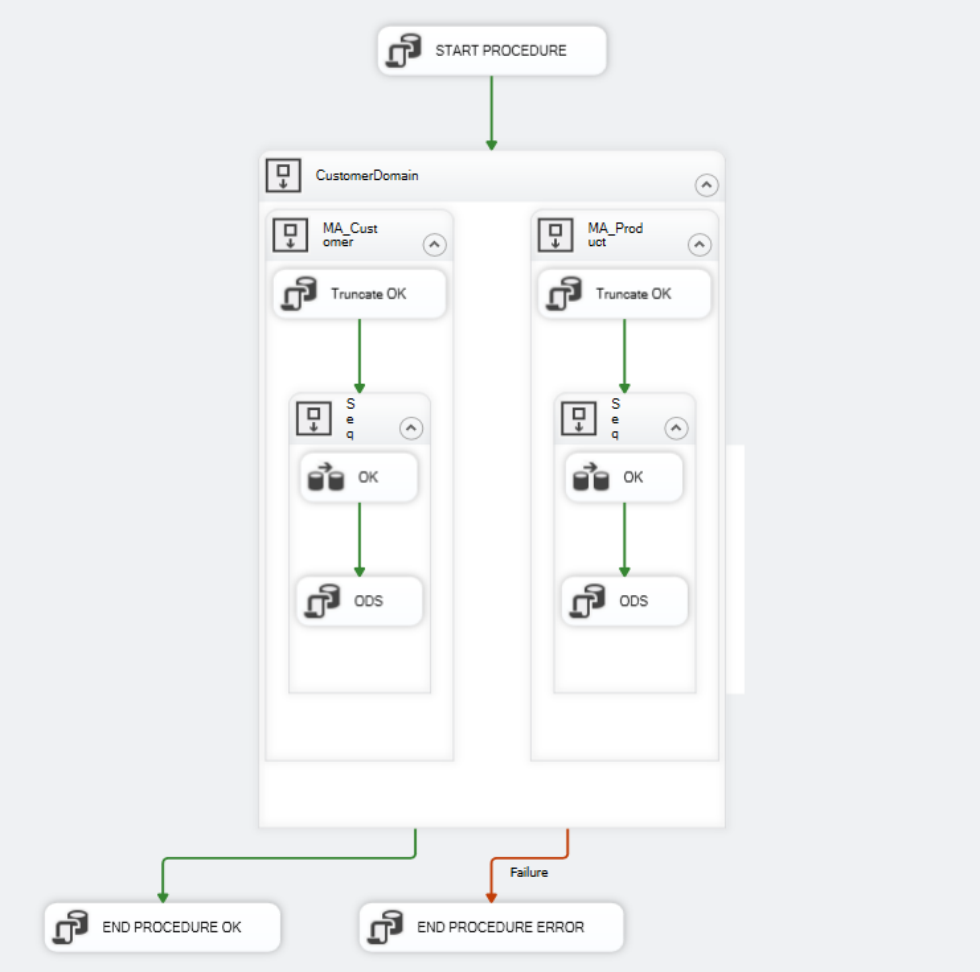

Figure 5.22: Package responsible for the execution of the L1 level.

Pre Load Package

Figure 5.23 represents the graphical interface of the SSIS package *001_PRE_LOAD_SalesTransactions* for the PRE LOAD step, explained in the previous chapter. As in the previous case, only the package related to the SalesTransactions group is going to be shows for the sake of simplicity, as the structure of the package is the same for different groups.
From the top to the bottom the tasks are:

- **Group Sequence container**: this sequence container groups the sequence containers regarding each table belonging to the group.
- **Table Sequence container**: this sequence container groups the tasks of a single table.
- **Dimension Table Sequence container**: this sequence container is created based on the relationships between the fact table and its dimension table. The number of sequence containers generated depends on how many dimensions are linked to the fact table.
- **Truncate PRE_LOAD**: a *Execute Sql Tasks* which executes the TRUNCATE query to remove all data from the dimension PRE LOAD table. In this case the MA_Sales has two linked tables, as a consequence two Execute Sql Tasks.
- **To_Table_PRE_LOAD**: is a *Data Flow Task* responsible for populating the PRE LOAD table with the necessary data.
- **PRE_LOAD_ODS_MERGE**: is a *Execute SQL Task* which performs a MERGE SQL query, shown in Listing 4.12, to insert the data in the ODS table.

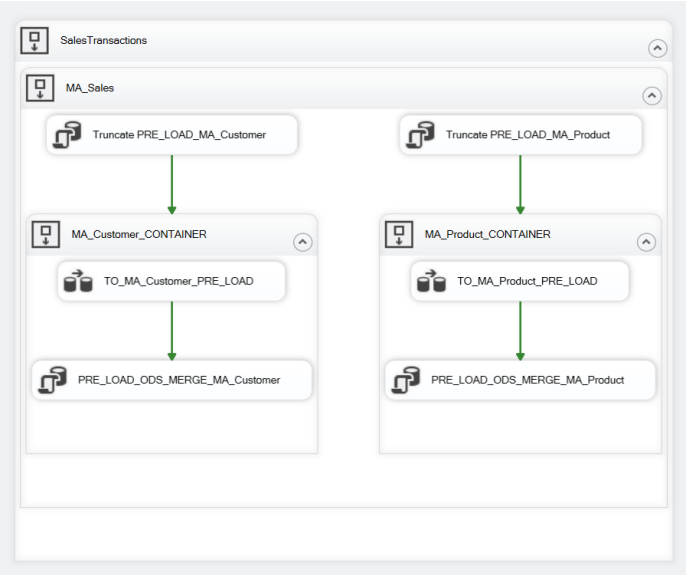

Figure 5.23: Package responsible for the pre load of the group SalesTransactions.

5.2 Results

Results of the execution of the packages are going to be presented in three scenarios. The first scenario one represent the initial data extraction from the source system. The second scenario shows how the program behaves when new and updated data, after the first upload, is introduced. Lastly, the third scenario addresses the case in which a record in a fact table has no corresponding entry in the related dimension table.

First scenario: initial data extraction

Since only the data from the tables belonging to the CustomerDomain group are going to be considered, we are narrowing the results to only the table shown in Figure 5.24 for sake of simplicity.

	CustomerID	CustomerName	Region	Segment
1	1	Alice Johnson	North	Retail
2	2	Bob Smith	South	Wholesale
3	3	Charlie Brown	East	Retail

Figure 5.24: MA_Customer source data

Following the workflow of the package responsible for the L0 level, represented in the Figure 5.20, in the FLOW_MANAGER is added a row representing the current status of the job, Figure 5.25 show the table at this point, with $status = 1$ as the process is running.

	IDENTITY	NUM_LEVEL	GRP_NAME	JOBID	STATUS	START_DATE	END_DATE
1	ANAGRAFICHE	0	CustomerDomain	20250917093000	1	2025-09-17 09:30:00	1900-01-01 00:00:00

Figure 5.25: FLOW_MANAGER table after first insert.

Subsequently, the data are copied from the source and the STG table is populated, as Figure 5.26 shows. Next, set operations are performed, and the resulting data are copied from the STG table to the DLT table, Figure 5.27 represents the results.

	CustomerID	CustomerName	Region	Segment	JOBID	INS_TIME
1	1	Alice Johnson	North	Retail	20250917093000	2025-09-17 09:30:00.000
2	2	Bob Smith	South	Wholesale	20250917093000	2025-09-17 09:30:00.000
3	3	Charlie Brown	East	Retail	20250917093000	2025-09-17 09:30:00.000

Figure 5.26: STG table populated

	CustomerID	CustomerName	Region	Segment	JOBID	INS_TIME	FLG_NEG
1	1	Alice Johnson	North	Retail	20250917093000	2025-09-17 09:30:00.000	0
2	2	Bob Smith	South	Wholesale	20250917093000	2025-09-17 09:30:00.000	0
3	3	Charlie Brown	East	Retail	20250917093000	2025-09-17 09:30:00.000	0

Figure 5.27: DLT table populated

Following the population of the STG and DLT tables, the TABLE_MANAGER is populated, as Figure 5.28 shows.

	IDENTITY	GRP_NAME	TABLE_NAME	JOBID	NUM_ROWS	INS_TIME
1	ANAGRAFICHE	CustomerDomain	MA_Customer	20250917093000	3	2025-09-17 09:30:00.000

Figure 5.28: Table_Manager table showing the number of rows copied to the STG table.

Lastly, since the procedure has ended successfully, the FLOW_MANAGER is updated and the status is updated to 3, for success, and the end_date is updated as well, as Figure 5.29 shows.

	IDENTITY	NUM_LEVEL	GRP_NAME	JOBID	STATUS	START_DATE	END_DATE
1	ANAGRAFICHE	0	CustomerDomain	20250917093000	3	2025-09-17 09:30:00	2025-09-17 09:30:00

Figure 5.29: FLOW_MANAGER after level L0 package has successfully executed.

Next, the package responsible for the L1 step is executed.
The FLOW_MANAGER is again updated with a new record representing the status of the process of the level L1, as Figure 5.30 shows.

	IDENTITY	NUM_LEVEL	GRP_NAME	JOBID	STATUS	START_DATE	END_DATE
1	ANAGRAFICHE	0	CustomerDomain	20250917093000	3	2025-09-17 09:30:00	2025-09-17 09:30:00
2	ANAGRAFICHE	1	CustomerDomain	20250917093000	1	2025-09-17 09:30:00	1900-01-01 00:00:00

Figure 5.30: FLOW_MANAGER table as soon at the package responsible for the level L1 is executed.

Next, the OK table is truncated and then populated with the new controlled data coming from the DLT table of the level L0, as shown in figure 5.31.

	CustomerID	CustomerName	Region	Segment	JOBID	INS_TIME	FLG_NEG
1	1	Alice Johnson	North	Retail	20250917093000	2025-09-17 09:30:00.000	0
2	2	Bob Smith	South	Wholesale	20250917093000	2025-09-17 09:30:00.000	0
3	3	Charlie Brown	East	Retail	20250917093000	2025-09-17 09:30:00.000	0

Figure 5.31: OK table after the population with the SQL query.

Following the population of the OK table, the MERGE SQL query is performed to populate the ODS table, with results shown in Figure 5.32.

	CustomerID	CustomerName	Region	Segment	JOBID_L1_INS	INS_TIME	FLG_NEG	JOBID_L1_UPD	UPD_TIME
1	1	Alice Johnson	North	Retail	20250917093000	2025-09-17 09:30:00.000	0	20250917093000	2025-09-17 09:30:00.000
2	2	Bob Smith	South	Wholesale	20250917093000	2025-09-17 09:30:00.000	0	20250917093000	2025-09-17 09:30:00.000
3	3	Charlie Brown	East	Retail	20250917093000	2025-09-17 09:30:00.000	0	20250917093000	2025-09-17 09:30:00.000

Figure 5.32: ODS table after the population with the MERGE query

The process ends, successfully, with the updated of the FLOW_MANAGER table, shown in Figure 5.33, as in the previous level.

	IDENTITY	NUM_LEVEL	GRP_NAME	JOBID	STATUS	START_DATE	END_DATE
1	ANAGRAFICHE	0	CustomerDomain	20250917093000	3	2025-09-17 09:30:00	2025-09-17 09:30:00
2	ANAGRAFICHE	1	CustomerDomain	20250917093000	3	2025-09-17 09:30:00	2025-09-17 09:30:00

Figure 5.33: FLOW_MANAGER after level L1 package has successfully executed.

Second scenario - Updating Data

Considering now an updated version of the MA_Customer, shown in Figure 5.24, with the Region of the customer with $CustomerID = 3$ updated to 'North'.
The procedure is the same as in the previous scenario. First, the FLOW_MANAGER table is updated with the new entry representing the actual process, Figure 5.34 shows the result.

	IDENTITY	NUM_LEVEL	GRP_NAME	JOBID	STATUS	START_DATE	END_DATE
1	ANAGRAFICHE	0	CustomerDomain	20250917093000	3	2025-09-17 09:30:00	2025-09-17 09:30:00
2	ANAGRAFICHE	1	CustomerDomain	20250917093000	3	2025-09-17 09:30:00	2025-09-17 09:30:00
3	ANAGRAFICHE	0	CustomerDomain	20250917100000	1	2025-09-17 10:00:00	1900-01-01 00:00:00

Figure 5.34: Flow manager table after the execution of the package.

The whole source data is again copied into the STG table, this time with updated JOBID and INS_TIME, as Figure 5.35 shows.

	CustomerID	CustomerName	Region	Segment	JOBID	INS_TIME
1	1	Alice Johnson	North	Retail	20250917093000	2025-09-17 09:30:00.000
2	2	Bob Smith	South	Wholesale	20250917093000	2025-09-17 09:30:00.000
3	3	Charlie Brown	East	Retail	20250917093000	2025-09-17 09:30:00.000
4	1	Alice Johnson	North	Retail	20250917100000	2025-09-17 10:00:00.000
5	2	Bob Smith	South	Wholesale	20250917100000	2025-09-17 10:00:00.000
6	3	Charlie Brown	Nord	Retail	20250917100000	2025-09-17 10:00:00.000

Figure 5.35: STG table

The main difference, with respect to the previous scenario, happens during the population of the DLT table. The *Minus* Sql Query, shown in Listing 4.10, is executed and the result is shown in Figure 5.36. The old row is inserted with $FLG\_NEG = 1$ while the new one with $FLG\_NEG = 0$

	CustomerID	CustomerName	Region	Segment	JOBID	INS_TIME	FLG_NEG
1	1	Alice Johnson	North	Retail	20250917093000	2025-09-17 09:30:00.000	0
2	2	Bob Smith	South	Wholesale	20250917093000	2025-09-17 09:30:00.000	0
3	3	Charlie Brown	East	Retail	20250917093000	2025-09-17 09:30:00.000	0
4	3	Charlie Brown	East	Retail	20250917100000	2025-09-17 10:00:00.000	1
5	3	Charlie Brown	Nord	Retail	20250917100000	2025-09-17 10:00:00.000	0

Figure 5.36: DLT table

Again, as in the previous scenario, in the TABLE_MANAGER will be inserted the number of row copied in the STG table.
The process finishes with the update of the FLOW_MANAGER table to note the success of the procedure, as Figure 5.37 shows.

	IDENTITY	NUM_LEVEL	GRP_NAME	JOBID	STATUS	START_DATE	END_DATE
1	ANAGRAFICHE	0	CustomerDomain	20250917093000	3	2025-09-17 09:30:00	2025-09-17 09:30:00
2	ANAGRAFICHE	1	CustomerDomain	20250917093000	3	2025-09-17 09:30:00	2025-09-17 09:30:00
3	ANAGRAFICHE	0	CustomerDomain	20250917100000	3	2025-09-17 10:00:00	1900-01-01 00:00:00

Figure 5.37: FLOW_MANAGER table after the package has completed successfully.

Following the end of the package responsible for the L0 level, the package for the L1 level is now executed. As in the previous case, the FLOW_MANAGER is updated to indicate the execution of the procedure, shown in Figure 5.38.

	IDENTITY	NUM_LEVEL	GRP_NAME	JOBID	STATUS	START_DATE	END_DATE
1	ANAGRAFICHE	0	CustomerDomain	20250917093000	3	2025-09-17 09:30:00	2025-09-17 09:30:00
2	ANAGRAFICHE	1	CustomerDomain	20250917093000	3	2025-09-17 09:30:00	2025-09-17 09:30:00
3	ANAGRAFICHE	0	CustomerDomain	20250917100000	3	2025-09-17 10:00:00	1900-01-01 00:00:00
4	ANAGRAFICHE	1	CustomerDomain	20250917100000	1	2025-09-17 10:00:00	1900-01-01 00:00:00

Figure 5.38: FLOW_MANAGER table after the execution of the package responsible for the L1 level.

Next, the OK table is truncated and repopulated, with results shown in Figure 5.39. Note that only the record with FLG_NEG=0 is passed, this is due to how the SQL query, shown in Listing 4.13, works.

	CustomerID	CustomerName	Region	Segment	JOBID	INS_TIME	FLG_NEG
1	3	Charlie Brown	Nord	Retail	20250917100000	2025-09-17 10:00:00.000	0

Figure 5.39: OK table

Since in the ODS table is already presented the record not updated, the MERGE Query will update the existing one, Figure 5.40 shows the result.

	CustomerID	CustomerName	Region	Segment	JOBID_L1_INS	INS_TIME	FLG_NEG	JOBID_L1_UPD	UPD_TIME
1	1	Alice Johnson	North	Retail	20250917093000	2025-09-17 09:30:00.000	0	20250917093000	2025-09-17 09:30:00.000
2	2	Bob Smith	South	Wholesale	20250917093000	2025-09-17 09:30:00.000	0	20250917093000	2025-09-17 09:30:00.000
3	3	Charlie Brown	Nord	Retail	20250917093000	2025-09-17 09:30:00.000	0	20250917100000	2025-09-17 10:00:00.000

Figure 5.40: ODS table updated

Lastly, Figure 5.41 shows the result on the TABLE_MANAGER of the successful execution of the package.

	IDENTITY	NUM_LEVEL	GRP_NAME	JOBID	STATUS	START_DATE	END_DATE
1	ANAGRAFICHE	0	CustomerDomain	20250917093000	3	2025-09-17 09:30:00	2025-09-17 09:30:00
2	ANAGRAFICHE	1	CustomerDomain	20250917093000	3	2025-09-17 09:30:00	2025-09-17 09:30:00
3	ANAGRAFICHE	0	CustomerDomain	20250917100000	3	2025-09-17 10:00:00	1900-01-01 00:00:00
4	ANAGRAFICHE	1	CustomerDomain	20250917100000	3	2025-09-17 10:00:00	1900-01-01 00:00:00

Figure 5.41: FLOW_MANAGER table after the successful execution of the L1 level package.

Third Scenario - Pre Load

For this scenario, the table MA_Sales from *SalesTransactions* group, shown in Figure 5.42, it is also considered.
The behaviour of the program when records do not have a referential check is going to be shown: in this case *CustomerID=5* and *CustomerID=7*, from

table represented in Figure 5.42, do not exist in the MA_Customer source table, shown in Figure 5.24.

	SaleID	CustomerID	ProductID	SaleDate	Amount
1	1	1	1	2025-10-01 00:00:00.000	1500
2	2	2	2	2025-10-02 00:00:00.000	800
3	3	3	3	2025-10-03 00:00:00.000	300
4	4	5	4	2025-11-03 00:10:00.000	750
5	5	7	5	2025-12-03 00:12:00.000	900

Figure 5.42: MA_Sales source table

The procedure until the PRE_LOAD package is the same as in the previous scenarios, the difference is that the PRE_LOAD package sends data to the ODS of the MA_Customer table.

Considering the PRE_LOAD package represented in figure 4.7, the PRE_LOAD tables are first truncated. Next, the data is taken from the DLT table to the PRE_LOAD table with the SQL query shown in Listing 4.12.
Figure 5.43 show the result in the PRE_LOAD table, which still includes all records from the fact table, both those with corresponding entries in the related dimension table and those without.

	CustomerID	CustomerName	Region	Segment	JOBID_L1_INS	INS_TIME	FLG_NEG	JOBID_L1_UPD	UPD_TIME
1	1				20250917100000	2025-09-17 10:00:00.000	0	20250917100000	2025-09-17 10:00:00.000
2	2				20250917100000	2025-09-17 10:00:00.000	0	20250917100000	2025-09-17 10:00:00.000
3	3				20250917100000	2025-09-17 10:00:00.000	0	20250917100000	2025-09-17 10:00:00.000
4	5				20250917100000	2025-09-17 10:00:00.000	0	20250917100000	2025-09-17 10:00:00.000
5	7				20250917100000	2025-09-17 10:00:00.000	0	20250917100000	2025-09-17 10:00:00.000

Figure 5.43: PRE_LOAD table

The last step of the PRE_LOAD package is to merge the obtained data with the ODS table of the dimension in consideration, MA_Customer in this case. The SQL MERGE query, shown in Listing 4.12 is now executed and figure 5.44 show the result, in the ODS table. It is noticeable that records without a referential match have their non key attributes set to default values, this serves as a flag to easily identify them and facilitate discussion on how the issue, in future, should be addressed.

	CustomerID	CustomerName	Region	Segment	JOBID_L1_INS	INS_TIME	FLG_NEG	JOBID_L1_UPD	UPD_TIME
1	1	Alice Johnson	North	Retail	20250917093000	2025-09-17 09:30:00.000	0	20250917093000	2025-09-17 09:30:00.000
2	2	Bob Smith	South	Wholesale	20250917093000	2025-09-17 09:30:00.000	0	20250917093000	2025-09-17 09:30:00.000
3	3	Charlie Brown	Nord	Retail	20250917093000	2025-09-17 09:30:00.000	0	20250917100000	2025-09-17 10:00:00.000
4	5				20250917100000	2025-09-17 10:00:00.000	0	20250917100000	2025-09-17 10:00:00.000
5	7				20250917100000	2025-09-17 10:00:00.000	0	20250917100000	2025-09-17 10:00:00.000

Figure 5.44: ODS table after pre load step has successfully completed.

Chapter 6

Conclusion and future directions

The goal of this thesis was to design and implement a metadata driven automation framework for SSIS package generation. By using metadata files and Python scripts, the program automatically generates SSIS packages following the data integration framework designed by the company.
This approach drastically reduces the time needed by developers to just create the packages and set up the parameters. For instance, a project with multiple tables per group could take up to two working days to be manually created while, once spent the time to create the metadata files, the process creation is reduced to just a few hours.

The framework was validated through experiments and results confirmed that the generated packages perform as expected. The PRE LOAD step proved effective in ensuring referential consistency between fact and dimension tables by inserting placeholder records flagged for further review.

Overall, this work demonstrates that metadata-driven automation can make ETL development more scalable, maintainable, and aligned with the evolving needs of data integration projects.

6.1 Future Directions

Although the proposed solution successfully achieves its objectives, there are several directions that could lead to future improvements, such as:

Generative AI for the metadata files generation, as instead of manually creat-

ing the metadata Excel files, generative AI, as large language models, could help in their creation. Recent research shows how generative AI can streamline pipeline and transformation tasks in data engineering [12].

ELT and cloud architectures adaptation, as the current solution is designed specifically for SSIS and on premise ETL. Extending the solution to support ELT processes would improve the program's future applicability as a shift from ETL to ELT is widely discussed in modern data integration literature [13].

User friendly graphical interface, as creating a graphical interface or a web frontend for managing metadata and executing the package generation without direct Excel file edits would make the tool more accessible to less technical users and reduce the potential for errors.

Implementing *Change Data Capture* method discussed in Chapter 4, where, differently than the *MINUS* solution that was implemented, the direct copy of all the data will no longer be performed and the STG table would be removed, keeping only the DLT table in the L0 level.

In the end, this thesis work approaches one of the most significant challenges in data engineering: the automation of ETL pipeline creation in response to specific user requests and gives a strong foundation with also the possibility for future improvements.

Bibliography

[1] 1KeyData.com. Data warehouse definition – what is a data warehouse. Online article, 2025. "A data warehouse is a subject-oriented, integrated, time-variant and non-volatile collection of data ...".

[2] Amazon Web Services. What is a data mart?, 2025. Accessed: 2025-10-06.

[3] Michael Armbrust, Tathagata Das, Shixiong Zhu, Ali Ghodsi, Reynold Xin, and Matei Zaharia. Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. https://databricks.com/wp-content/uploads/2021/05/Lakehouse-Whitepaper.pdf, 2021. Accessed: 2025-06-25.

[4] J. Power D.˙Data warehouse. DSSResources Glossary, 2021.

[5] James Dixon. Pentaho, hadoop, and data lakes. https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/, 2010. Accessed: 2025-06-25.

[6] Folium.ai. Cloud data warehousing- pros and cons, 2023. Accessed: 2025-10-06.

[7] Matteo Golfarelli and Stefano Rizzi. *Data Warehouse Design: Modern Principles and Methodologies*. McGraw-Hill Education, New York, 2009.

[8] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data integration: The teenage years. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 9–16. VLDB Endowment, 2006.

[9] IBM. What is a data mart?, 2025. Accessed: 2025-10-06.

[10] Mastech Infotrellis. Cloud vs. on-prem: Choosing the right data warehouse, 2023. Accessed: 2025-10-06.

[11] William H. Inmon. *Building the Data Warehouse*. Wiley, Indianapolis, IN, 4th edition, 2005.

[12] Krishna Kanagarla. Data engineering with generative ai: Automating pipelines and transformations using llms. *SSRN*, 2025. Describes usage of generative AI in automating data pipelines.

[13] Srinivasa Rao Karanam. Etl vs elt: A comparative analysis for modern data integration. *International Journal of Core Engineering Management*, 2023. Comparative study of ETL and ELT paradigms.

[14] Ranjith Katragadda, Sreenivas Sremath Tirumala, and David Nandigam. Etl tools for data warehousing: An empirical study of open source talend studio versus microsoft ssis. In *Proceedings of the 2nd World Congress on Computer Applications and Information Systems (WCCAIS 2015)*, Hammamet, Tunisia, 2015. N&N Global Technology.

[15] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. Wiley, New York, 1996.

[16] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. Wiley, Indianapolis, IN, 3rd edition, 2013.

[17] Dan Linstedt and Michael Olschimke. *Building a Scalable Data Warehouse with Data Vault 2.0*. Morgan Kaufmann, Waltham, MA, 2015.

[18] Microsoft. Sql server integration services, jan 2025. Ultimo aggiornamento: 2 gennaio 2025.

[19] Microsoft Learn. Oltp vs. olap: What's the difference? https://learn.microsoft.com/en-us/sql/ssas/olap/olap-vs-oltp, 2023. Accessed: 2025-06-25.

[20] Panoply. Data mart vs data warehouse: The difference with examples, 2024. Accessed: 2025-10-06.

[21] Raghu Ramakrishnan, Johannes Gehrke, and Venkatesh Ganti. *Database Management Systems*. McGraw-Hill, New York, NY, 4th edition, 2019.

[22] R. Sawadogo et al. Data lakes for big data analytics: Challenges and opportunities. *Journal of Big Data*, 7:1–20, 2020.

[23] D. Seenivasan. Etl vs elt: Choosing the right approach for your data warehouse. *International Journal for Research Trends and Innovation*, pages 110–122, 2022.

[24] TechTarget. On-premises vs. cloud data warehouses: Pros and cons, 2024. Accessed: 2025-10-06.

[25] Panos Vassiliadis. A survey of extract–transform–load technology. *International Journal of Data Warehousing and Mining*, 5(3):1–27, 2009.