Politecnico di Torino

Politecnico di Torino

Master's Degree in Ingegneria Informatica
(Computer Engineering)

A.a. 2025/2026

Graduation Session March 2026

Modernizing Enterprise Data Integration

Building an ETL Framework on the Boomi Ecosystem Enhanced by Generative AI

Supervisor:
Prof. Tania Cerquitelli

Candidate:
Gaia De Falco

Company Supervisor:
Dr. Daniela De Angelis

Abstract

The thesis project, developed during my internship at **Mediamente Consulting**, was conceived in response to a client's need to modernize and enhance its data integration processes. The specific requirement was the adoption of a technology offering native cloud scalability, a low-code/no-code development approach for accelerated time-to-market, and parallel processing capabilities to manage increasing data volumes efficiently.

To address this, the main objective was the design and development of a robust **ETL** framework built on **Boomi**, a market-leading cloud-native platform. The resulting architecture was engineered to be modular, highly parameterizable, and inherently scalable, ensuring fluid integration across diverse systems by efficiently managing the complex stages of data extraction, transformation, and loading.

The project's focus was twofold: an intensive exploration of Boomi's advanced features and their practical application to the proprietary corporate framework. A key priority was the transition of legacy workflows into a Boomi-native environment, which required a re-interpretation of the existing Oracle-based logic to leverage the platform's architecture fully.

This process was essential to transform rigid procedures into a standardized data flow that ensures enterprise-wide reliability and governance. Beyond the technical implementation, this experience provided extensive hands-on experience with Boomi's orchestration tools, demonstrating how a cloud-native approach enables a more comprehensive management of information. This transition not only optimizes current processes but also establishes a flexible environment capable of integrating broader data perspectives and diverse future sources.

The project also explored advanced optimizations to further refine the integration lifecycle.
First, the framework's scalability was assessed in relation to **Boomi DataHub**, identifying it as a strategic evolution for centralized master data management (MDM) and cross-platform governance. Separately, experimental tests were conducted using **Boomi AgentStudio** to develop autonomous agents capable of automating the synthesis of technical profiles from natural language inputs, allowing for their direct deployment onto the process canvas.

Ultimately, this framework provides a solid and reusable foundation for future enhancements to the client's integration landscape, contributing to a more efficient, auditable, and controlled management of data flows while consolidating the expertise in modern middleware technologies.

Table of Contents

List of Figures

List of Tables

Glossary

AI/ML	Artificial Intelligence/Machine Learning
APAC	Asia-Pacific
API	Application Programming Interface
BA	Business Analytics
BI	Business Intelligence
CDC	Change Data Capture
CPM	Corporate Performance Management
CRM	Customer Relationship Management
CRUD	Create, Read, Update, Delete
CSV	Comma Separated Values
CTE	Common Table Expression
CUI	Conversational User Interface
DBA	Database Administration
DCL	Data Control Language
DDL	Data Definition Language
DGM	Deep Generative Model
DL	Deep Learning
DML	Data Manipulation Language
DPP	Dynamic Process Property
DQL	Data Query Language
DQS	Data Quality Steps
DWH	Data Warehouse
ELT	Extract, Load and Transform
EMEA	Europe, Middle East, and Africa
ERP	Enterprise Resource Planning

ETL	Extract, Transform and Load
FK	Foreign Key
GenAI	Generative Artificial Intelligence
IDE	Integrated Development Environment
iPaaS	Integration Platform as a Service
JDBC	Java DataBase Connectivity
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LLM	Large Language Model
MDM	Master Data Management
ML	Machine Learning
NoSQL	Not only SQL
ODI	Oracle Data Integrator
ODS	Operational Data Store
OJDBC	Oracle JDBC Driver
OLTP	Online Transaction Processing
PII	Personally Identifiable Information
PK	Primary Key
SaaS	Software as a Service
SCD	Slowly Changing Dimensions
SK	Surrogate Key
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCO	Total Cost of Ownership
TLS	Transport Layer Security
UDF	User Defined Function
URL	Uniform Resource Locator
XML	EXtensible Markup Language

Chapter 1

Introduction

In recent years, enterprise operations have experienced a major digital transformation that has reshaped how internal processes are designed, managed, and optimized. The rapid evolution of information technologies, together with the widespread use of cloud services and the growing interconnection of systems, has turned information into a strategic asset and a crucial driver of competitiveness. As a result, modern organizations now operate within technological ecosystems that are increasingly complex and often composed of diverse and heterogeneous platforms.

It is within this context, and in response to a client's need to reduce operational costs, improve efficiency, and significantly lower data latency across their enterprise, that this thesis project took shape. Its main objective was to bring process design under a single platform capable of enabling seamless communication among heterogeneous systems and of supporting the development of data integration pipelines with relatively low effort. The work focused on translating a common integration standard, originally designed for internal use at Mediamente Consulting, into the Boomi platform in a way that could be easily replicated and reused across different business units and operational areas of the client's organization.

Consequently, the project's goal became the creation of a robust and scalable framework intended to serve as a reference model for future integration initiatives. This will allow the organization to address new challenges quickly and remain aligned with the rapid pace of technological change.

1.1 The Company

The company where I carried out my thesis project is **Mediamente Consulting S.r.l.**[1], a firm specializing in Advanced Business Analytics and Data Management, and an integral part of the Data Science Business Unit of **Var Group** (SeSa S.p.A. group) [2]. This Business Unit functions as a strategic hub, consisting of over 100 specialists, dedicated to guiding enterprises in their transformation into *"truly Data-Driven Companies"*. Its comprehensive offering spans Data Management, Business Intelligence, Advanced Analytics, and Applied Artificial Intelligence, primarily targeting the Retail, Manufacturing, Pharmaceutical, and Financial Services sectors.

Mediamente Consulting structures its operations into various units, each dedicated

to developing advanced and tailored technological solutions, namely:

1. **Database Administration (DBA)**: responsible for managing and securing the company's databases. Their role is to ensure that data is well-organized, accessible, protected, and always available

2. **Data Integration**: responsible for acquiring, cleaning, and harmonizing data from multiple sources. Their work involves extracting information from different systems, transforming it to comply with corporate rules and client-specific requirements, and loading it into a centralized data warehouse

3. **Data Visualization**: specialized in designing and optimizing dashboards and reports, primarily with Power BI. Their goal is to allow clients to explore data intuitively and make well-informed decisions based on it

4. **Corporate Performance Management (CPM)**: responsible for planning, budgeting, forecasting, and monitoring business performance, helping the company to manage resources effectively and achieve strategic objectives.

Specifically, the unit directly involved in this thesis project is **Data Integration**. Its core mandate is critical to the client's data journey: to analyze data sourced from heterogeneous systems and harmonize it into a common profile (ensuring Data Governance), thoroughly understanding the potential of modern integration platforms (such as low-code iPaaS), and strategically advising clients on the most efficient and high-performing software solutions aligned with their specific business objectives.

1.2 Purpose of the project

Data has taken on an increasingly central strategic role in everyday business management. Organizations that adopt a data-driven approach are able to convert their analysis into a competitive value, predicting customer needs and market trends, enhancing service quality, identifying inefficiencies, and optimizing resource management.

However, the exponential increase in the volume of data generated daily makes it necessary to use tools that are capable of collecting, processing, and harmonizing this information not only effectively but also in an automated manner.

One of the primary setbacks to the full valorization of corporate data lies in its heterogeneity; information is frequently scattered across disparate systems, organized in various formats (e.g., CSV, JSON, XML files, relational or NoSQL databases, and external APIs), and is characterized by variable quality levels.

The absence of a unified informational flow not only slows down operational processes but also prevents the achievement of a complete and coherent view of corporate information.
Without effective integration, data becomes challenging to compare, aggregate, and analyze, severely limiting the potential for extracting useful and business-critical knowledge.

In this context, ETL processes play a fundamental role, enabling data extraction from various sources, its transformation according to common rules, and its loading

into centralized or distributed target systems. Technological evolution has made these processes increasingly automated, modular, and scalability-oriented.

In recent years, **iPaaS** solutions have emerged, offering cloud-based tools for the centralized management of integration flows [3]. These platforms allow for the more efficient design, execution, and monitoring of ETL processes, simultaneously reducing infrastructure costs and simplifying maintenance.
Among these solutions is **Boomi**[4], the platform deployed in the project presented in this thesis, specialized in Integration Platform as a Service (iPaaS), API Management, Master Data Management, and Data Preparation.

The presence of a single platform capable of orchestrating an entire ETL pipeline in a modular, low-code/no-code manner, and with low latency, significantly enhances the work performed by analysts and software designers [5]. It allows connecting to a wide variety of information sources, and, more importantly, it does not limit the creation of pipelines to personnel proficient in specific programming languages. Rather, it increases the design scalability, the modularization of work, and the pool of competent individuals who can contribute to it, directly translating into improved operational efficiency for the client.

The ultimate objective of this project is to deliver a comprehensive analysis of the Boomi Platform, with the goal of understanding its core concepts, architectural principles, and operational capabilities. In addition, the project involves the translation and adaptation of an existing ETL workflow into the Boomi environment, as well as an extended exploration of advanced platform features, including AI Agents and DataHub. The study includes an in-depth evaluation of Boomi's integration, transformation, and orchestration capabilities, supported by a structured testing process to assess the platform's suitability for the client's requirements.

1.3 Scope and Limitations

While the core strength of the Boomi platform lies in its employment of functional blocks (**Shapes/Steps**) and the low-code/no-code approach for constructing ETL pipelines, this also constitutes an inherent limitation.

The platform is designed primarily as an *Application Integration Tool*, offering high flexibility in the selection of connectors for heterogeneous ecosystems. However, unlike dedicated ELT engines, which are natively optimized for high-volume data transformations, Boomi's record-centric architecture can introduce a significant overhead.

While the platform does not strictly prohibit complex logic, the implementation of intricate algorithmic transformations often faces the constraints of the JVM heap memory and the latency inherent in shape-to-shape data routing. In a standard configuration, the constant parsing and re-serialization of documents across multiple Map stages can lead to non-linear performance degradation as data volumes scale.

Furthermore, although the platform is readily accessible to users with limited programming expertise, the design and implementation of highly complex, customized,

or high-performance workflows requires specialized technical knowledge to leverage the platform's advanced features and best practices fully.

Lastly, while Boomi supports relational databases relatively easily, the platform is not optimized to match the native processing performance of a dedicated database or data warehousing engine when handling large volumes of complex tabular data.

1.4 Future Developments

Considering the rapid and continuous transformation of digital technologies, the developed framework offers several directions for strategic evolution.

A future development might involve the integration of **Machine Learning** algorithms directly within the ETL pipeline, for instance, AI models that can be leveraged to automate data quality checks, perform predictive transformations, or optimize process performance.

In this regard, preliminary tests were conducted during this project to evaluate the automation of specific workflow segments through **AI Agents**. A primary area of focus was the autonomous generation of **Profiles**, which serve as the fundamental structural engine of any Boomi integration flow. While the platform provides a native Oracle connector capable of automatic profile creation via "Import Operations," this functionality is often limited to standard relational schemas. In scenarios where a native connector is unavailable or where data is sourced from unstructured or non-standardized APIs, AI Agents can be employed to parse source metadata and automatically build the required Boomi profiles [6]. This approach significantly reduces the manual effort of mapping complex JSON or XML structures and ensures that the integration logic remains agile and adaptable to changing source specifications.
Furthermore, even when standard imports are possible, they are often insufficient when specific data types or custom formats are required during the transformation pipeline. A technical requirement might necessitate converting numeric types or enforcing strict hierarchical constraints that differ from the source schema. Automating this process ensures that profiles are not only synchronized with the database but also pre-configured with precise technical specifications, reducing manual overrides and the risk of data type mismatches.

During the development of this project, preliminary explorations of **DataHub**'s capabilities were already conducted, particularly leveraging its environment to execute stages of the `OK` Stage (validation/target); a strategically significant evolution would be, instead, the deployment of the entirety of the ETL pipeline on Boomi DataHub [7].

Integrating DataHub represents a natural next step, as it transforms the existing ETL architecture into a centralized Master Data Management (MDM) solution, offering a specialized environment for managing, validating, and synchronizing master entities across the enterprise, embedding governance directly into the integration lifecycle. By migrating logic into DataHub, the client gains a unified system for defining *"Golden Records"*, enabling full end-to-end traceability and consistent data propagation across the entire information ecosystem.

1.5 Thesis Structure

The project structure is outlined through several consecutive phases, each aimed at achieving a specific goal in the development process of the ETL pipeline.
Initially, the project provides an in-depth overview of the company's existing ETL workflow and underlying framework, followed by a detailed study of the Boomi Platform, focusing on its core features, components, and extended capabilities.
The workflow is then systematically decomposed into well-defined stages and implemented within the Boomi environment to reflect the company's operational processes. Finally, potential optimizations and future development opportunities are identified and discussed to enhance performance, scalability, and long-term maintainability.

In the following chapters, the areas that will be addressed are:

- State of the art
- The Boomi Platform
- Technical specification of Project Components
- ETL Workflow Design and Implementation
- Enhanced ETL Workflow with Error Handling and Alerts
- Optimized ETL Workflow with Data Lookup
- Boomi DataHub
- AI Agents and AgentStudio
- Conclusions

Chapter 2

State of the Art

Mediamente Consulting employs a proprietary and well-established ETL framework across its client base. The primary objective of this standardized methodology is to ensure a unified project direction, significantly optimizing development and long-term maintenance across diverse data solutions; furthermore, it guides developers in creating solutions that align with the company guidelines while still being tailored for each client's needs. In particular, this framework leverages the core ETL principles of connecting and transforming data from various sources, ensuring compliance, security, and reliability, while managing high volumes of data.

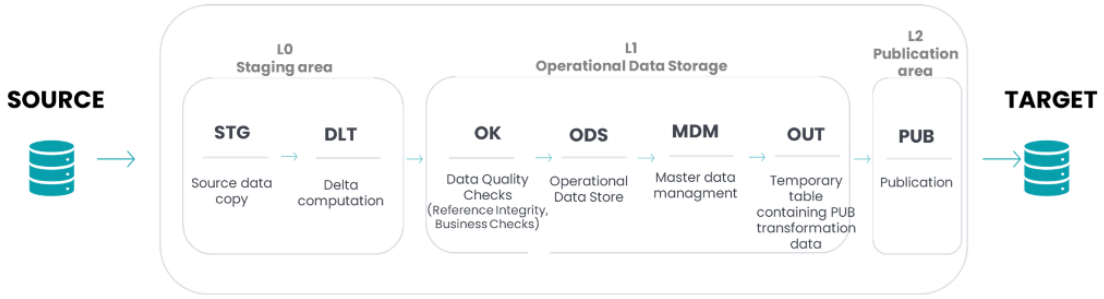


Figure 2.1: Mediamente Consulting's proprietary ETL framework

This internal framework has proven to be highly effective and flexible, notably characterized by its:

- **Usability**: simple implementation and management by technical teams
- **Accessibility**: ease of comprehension for new developers and during maintenance operations
- **Scalability**: capacity to adapt to evolving data volumes and client requirements

It was against this background that a specific client request emerged: the need to migrate their existing data integration and ETL flow to the Boomi Integration platform.

The reasons behind this request were primarily strategic, aiming at overcoming the limitations of the client's previous setup; specifically, they sought a solution that would deliver improvements in three key areas:

1. **Cost Reduction:** the transition to a modern, cloud-native platformlike Boomi, was expected to provide a more cost-effective and scalable pricing model compared to the existing infrastructure. The migration aimed to lower the total cost of ownership (TCO) while providing the flexibility to handle fluctuating data volumes

2. **Operational Efficiency:** the project sought to enhance the overall efficiency of data-flow management by simplifying complex integration tasks and automating manual processes. This consequently increased the reliability and maintainability of data pipelines, ensuring a more robust framework for long-term operations

3. **Latency Reduction:** the architecture focuses on substantially decreasing data latency to meet the sector's growing demand for real-time data availability, reduced time-to-market, and faster decision-making

Given the proven reliability of Mediamente Consulting's proprietary framework, the project aimed to leverage the advanced capabilities and cloud-native functionalities offered by Boomi. Consequently, the project was defined as the faithful yet optimized re-implementation of the company's ETL framework on the new platform. This strategic approach preserves the tested logical architecture and established operational advantages while significantly enhancing them through Boomi's superior connectivity and scalability features.

Specifically, the architecture of the existing framework is structured around three general phases, each further segmented into sub-phases that delineate the process from data extraction to final loading and merging.

2.1 Data Warehouse

The framework designed by the company is predominantly focused on managing relational data (structured in tables), and it leverages consolidated, enterprise-level data integration systems; specifically, the framework is engineered to interact with and mirror the architecture of traditional Data Warehouse (DWH) systems.
In this context, a detailed analysis of the DWH architecture is essential for understanding the integration logic itself.

The design of the framework inherently supports the two primary modeling schemas used to structure the Data Warehouse:

1. Star Schema: the most widespread Data Warehouse model; it consists of a single, central Fact Table surrounded by multiple, directly linked Dimension Tables. It simplifies query design and significantly enhances query performance due to fewer complex joins being required to retrieve data

2. Snowflake Schema: an extension of the Star Schema where the dimension tables are normalized, meaning that they are further broken down into separate subtables to eliminate data redundancy. It reduces storage space, but it increases

model complexity and degrades query performance due to the higher number of join operations required to traverse the dimensional hierarchy

In line with the foundational principles of dimensional modeling by Ralph Kimball and Margy Ross in "*The Data Warehouse Toolkit*"[8], and following the architectural concepts of Data Warehousing as outlined by Filippo La Noce and Luigi D'Ercole in "*Data Warehousing: dal dato all'informazione*"[9], the framework is explicitly tailored to support systems designed for advanced business analysis.

This approach guarantees that the integrated data is organized around four critical factors:

- Usability and Intuitiveness: presenting data in a simplified structure makes it immediately comprehensible and navigable by business users
- Query Performance: optimizing data access for Business Intelligence (BI) tools and complex analytical queries by minimizing table joins and maximizing lookup speed
- Consistency: enforcing the use of conformed dimensions across multiple fact tables and business processes, ensuring that metrics are calculated consistently throughout the enterprise
- Flexibility and Extensibility: allowing the model to be easily adapted to accommodate new business requirements, data sources, and organizational changes without requiring a complete redesign

The adoption of a framework thus engineered ensures that Data Integration is not merely a data movement exercise but a fundamental step towards effective Data Governance and the extraction of strategic analytical value.

2.2 L0 - Staging Area

The Staging Area constitutes the initial phase, or layer, of the ETL process within the DWH architecture, responsible for receiving the extracted data from various operational source systems (OLTP), such as management software, flat files, APIs, or relational databases.

Data resides in its raw, unprocessed format, retaining the structure and data types of the source system; no transformation, cleaning, or validation logic is applied in this phase.

The data extraction can follow two possible strategies, depending on latency requirements, data volume, and the impact on the source systems:

- Full Extraction: entails the transfer of all records from the source system to the target, without prior evaluation of modifications since the last extraction. While this method ensures data completeness and simplicity, it often leads to operational inefficiencies when dealing with large datasets, as it requires considerable time and system resources
- Initial Load: a complete initial load of the data into the DWH, followed by periodic extractions that consider only the modified or newly added records. There are several techniques to achieve this strategy [8]:

- **CDC** (change data capture): an active table monitoring to take into account data changes automatically through a measure of the last data insertion time. It is best suited for large tables since it avoids the consecutive downloads of unnecessary data, although it does not allow to keep track of physical cancellations
- **MINUS**: a set-difference operation obtained by comparing the current state of the source database with the last known copy in the DWH. It is best suited for keeping track of physical cancellations, but it is inefficient for large volumes of data

A key methodological principle of the project is the decoupling of the **STG** (Staging) and **DLT** (Delta) steps. This separation was adopted on the account that the source system lacked a native timestamp or technical key to identify and extract incremental changes directly. This approach allows the STG layer to isolate the extraction process from resource-intensive transformation logic, thereby minimizing the impact on production systems, while the DLT stage focuses on processing only the necessary incremental changes; it significantly reduces processing time and I/O load compared to repeated full loads.

It is important to emphasise that, following Mediamente Consulting's best practices, in the event that a source field for incremental extraction is made available, it is recommended to only adopt the DLT Stage.

2.2.1 STG

Maintaining a faithful 1:1 replica of the extracted data within this layer, regardless of the subsequent extraction strategy, is crucial as it provides a reliable rollback and audit point for future analysis or corrections without needing to repeat the extraction process.

To ensure proper data control, technical metadata is added to every record in this stage:

- `JOBID`: execution timestamp, represents the time at which the execution takes place
- `INS_TIME`: insertion time, represents the time at which the data is extracted for the first time

These metadata are propagated and referenced throughout the entire ETL process to facilitate delta identification and manage the DLT step. This ensures that the downstream transformation logic can accurately distinguish between newly processed records and those previously loaded, which is vital for maintaining DWH integrity and efficiency.

2.2.2 DLT

It is critical for identifying differences between the current extraction and the previous state, thus enabling the precise management of data changes across the pipeline. This stage is implemented through distinct strategies, depending on the available metadata on the source system:

- **CDC (Change Data Capture):** In this approach, Delta identification relies on specific source fields that flag new or modified records. Depending on the source system, this is typically handled in two ways:
  - **Timestamp-based:** for instance, using a field like `UPD_TIME` (date and time of the last modification). Any record with an `UPD_TIME` greater than the maximum timestamp of the previous load is extracted
  - **Status-based:** using a status field (e.g., pro-status) that tracks the record's lifecycle through specific codes: *null* for new or updated records, **P** during processing and transmission, **S** for successfully received data, and **E** in case of transmission errors

  It remains a known limitation of these CDC methods that, without explicit source fields, no automated mechanism exists to reliably track physical record deletions

- **MINUS:** this technique involves a row-by-row comparison between the current load and the previous load to determine the exact delta. A new technical metadata field, `FLG_NEG`, is added to manage the status of the record, as it indicates the type of operation:
  - **0 (Insert):** a new record
  - **1 (Delete/Cancellation):** a physical deletion from the source

  The update case is handled differently; the previous version of the record is marked as a cancellation (`FLG_NEG=1`), immediately followed by the insertion of the modified record (`FLG_NEG=0`).

2.3 L1 - Operational Data Storage

The Operational Data Storage is a crucial step in the ETL process, dedicated to preparing and transforming data in a coherent and consolidated format for subsequent analytical use.

In this phase, the raw data extracted from the STG and DLT layers undergoes a series of operations to achieve the integrity required by the target analytical systems; in particular, cleaning, correction, and error handling processes are applied.

This phase is further divided into four other phases to optimize execution, maintainability, and fault tolerance, increasing debugging capabilities and flow management.

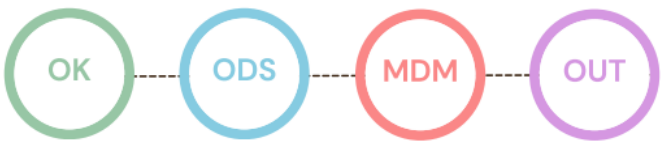


Figure 2.2: Four consecutive steps that constitute the L1 stage

Namely, these consecutive steps are:

- OK: guarantees referential integrity, applies data validation rules

- ODS (Operational Data Store): consolidates data into a common format, ensuring consistency
- MDM (Master Data Management): enables data enrichment by incorporating standardized master data from internal and/or external sources
- OUT: holds the latest, fully processed records and manages the final data structure, before publication to the analytical targets

2.3.1 OK Step

This transformation step is fundamental to ensuring the quality and referential integrity of the data prior to final consolidation. At the core of this phase lies the **truncate-insert strategy**: rather than adopting an incremental update logic, the `OK` table is systematically cleared and repopulated during every execution. This architectural choice is critical, as it ensures the table consistently reflects the most current and verified state of the records, effectively eliminating the risk of data obsolescence or residual inconsistencies from previous runs.
The process is structured into two distinct operational phases, governed by this refresh logic:

- **Data Validation**: the verification of referential integrity between Foreign Keys (FKs) and Primary Keys (PKs). This step enforces strict technical constraints, including data types, format compliance, business and not-null rules, ensuring that only schema-perfect data survives the insertion
- **Error Handling**: a self-healing mechanism that isolates non-compliant records into a dedicated quarantine area (the `E$` table). This separation not only shields the target system from corruption but is also fundamental for providing actionable feedback to the client. This allows for the subsequent remediation and reprocessing of records once anomalies are resolved, ensuring zero data loss across the pipeline

A critical requirement for the success of this phase is ensuring absolute record uniqueness (deduplication) before the final insert takes place. This is achieved by applying the `ROW_NUMBER()` window function, computed over the record's PKs and technical metadata. This ensures that only a single, authoritative version of each record is passed forward; in instances where multiple records share the same key, custom business rules are invoked, such as preserving the most recently modified entry or the one with the highest data density, to determine the definitive version to be retained.

2.3.2 ODS Step

Following the initial validation phase, the `ODS` stage is dedicated to current state management, ensuring the layer consistently maintains the most accurate and up-to-date version of every record. To achieve data consistency, this layer employs a `MERGE` statement to efficiently synchronize incremental changes through a primary key-based comparison: a match triggers an `UPDATE` to refresh existing entities with validated values, while a mismatch results in a new `INSERT`.

Beyond synchronization, the ODS serves as the definitive gateway for data harmonization, ensuring all information adheres to unified business standards before exposure to downstream consumers. To support this governance, the architecture enriches each record with a specialized set of technical metadata designed for long-term lineage and auditing:

- **Persistence**: the original execution metadata is captured to preserve the record's origin; specifically, JOBID is mapped to JOBID_INS, while INS_TIME is persisted without modification. These values remain immutable throughout the record's lifecycle, ensuring a reliable historical audit trail
- **Modification Tracking**: temporal updates are managed via UPD_TIME and JOBID_UPD, which reflect the most recent execution. While INS_TIME preserves the immutable moment of first ingestion, UPD_TIME provides real-time visibility into the latest state of the data

This dual-tracking mechanism allows administrators to reconstruct the evolution of any record, enabling rapid troubleshooting and corporate compliance. By persisting rigorously validated information, the ODS functions as a *"Single Source of Truth"*, a high-quality asset that guarantees reliability for strategic reporting and long-term historical analysis.

2.3.3 MDM Step

The MDM Step is the subsequent step in the transformation pipeline, primarily focused on enhancing the analytical value of the data through enrichment, which integrates current data with supplementary data originated from internal and external data sources, applying various merging strategies and business rules.

The MDM Step is often the point where the historical traceability of the data entity is finalized; this is implemented using Slowly Changing Dimensions (SCD) methodology to ensure the chronological tracking of changes over time, namely, maintaining a record of the state of an entity at any given point in time [8].

A central architectural responsibility of the MDM is the assignment and management of Surrogate Keys (SK). These are unique, internal identifiers generated specifically to represent entities within the DWH, replacing the original natural keys. Typically, they are *"integers either assigned in sequence or generated by a robust hashing algorithm that guarantees uniqueness"*[8].

SKs are inserted to achieve several strategic architectural objectives:

- Data Independence: they remain invariable even if the entity's natural key changes in the source system, thus preventing cascading updates in the DWH
- Guaranteed Uniqueness: they ensure a unique identifier is always available, particularly when natural keys may be composite, non-unique, or poorly managed in the source systems
- Simplified Data Integration: they facilitate data integration by avoiding key conflicts when merging data from different source systems

- Performance Improvement: SKs are more efficient than natural keys (e.g., long strings or composite keys) in terms of database storage and, critically, for optimizing join operations within the model

2.3.4 OUT Step

The OUT Step represents the final orchestration phase, consisting of creating a staging or temporary table specifically designed to mirror the target schema, typically a dimensional model, deployed in the target system. This layer acts as a structural buffer, ensuring coherence and data uniformity before the definitive loading occurs.

This temporary structure isolates the final Delta, which contains only the latest set of validated records and modifications destined for publication. By decoupling the transformation logic from the final ingestion, the OUT Step serves as a critical checkpoint to verify that all data types, granularity levels, and structural relationships adhere to the target dimensional design; this is an essential audit point, employed to prevent loading failures and to ensure that the data is "business-ready".

2.4 L2 - Publication Area

This final stage involves the delivery of the fully transformed data into the Publication Area.

The data, now structured according to the dimensional model defined in the preceding (OUT) layer, is loaded into the target environment, specifically designed to maximize analytical performance and user accessibility.

This layer contains aggregated and structurally optimized data (typically in the form of fact and dimension tables) engineered to facilitate extremely fast querying and deep-dive analytical tasks; data granularity is set to the level required by the end-users, balancing detail against query speed.

The end goal is to provide a clean, structured, and unified DWH that acts as the single source of truth for all business intelligence activities. By presenting data in a format optimized for analysis, this layer directly supports the strategic decision-making processes across the enterprise, ensuring high-performance report generation and providing a stable, consistent view of historical and current trends, essential for reliable performance measurement and forecasting.

2.5 Case Study

To clarify the implementation of these steps, the following scenario, involving processing inventory data from a corporate warehouse, is considered as a practical use case.

The core objective of the entire ETL process is to ingest this raw source information and provide a clean, validated, and up-to-date dataset to a third-party Data Visualization or Business Intelligence (BI) tool, ensuring that only validated data is used for reporting and decision-making.

For this scenario, the chosen technique for accurately computing the difference (delta) between successive ETL runs is the **DLT MINUS** strategy.

In particular, the following table [2.3] illustrates the source data ingested into the ETL workflow, which includes the product code, the provider, the ordered quantity, a description, and the month of the order. While the schema contains three foreign keys, `CODE`, `PROVIDER_CODE`, and `MONTH_CODE`, for the sake of simplicity, this analysis will focus exclusively on the `CODE` attribute and its corresponding master data table

CODE	DESCRIPTION	QUANTITY	PROVIDER_CODE	MONTH_CODE
cod_1	desc_1	0	pcode_1	mcode_1
cod_2	desc_2	1	pcode_2	mcode_2
cod_3	desc_3	2	pcode_3	mcode_3
cod_4	desc_4	3	pcode_4	mcode_4
cod_5	desc_5	-4	pcode_5	mcode_5

Figure 2.3: Source table

JOBID_INS	JOBID_UPD	CODE	INS_TIME	UPD_TIME	FLG_NEG	DESCRIPTION
20251010110000	20251011110000	cod_1	10/10/2025 11:00	11/10/2025 11:00	0	desc_1
20251011110000	20251013110000	cod_2	11/10/2025 11:00	13/10/2025 11:00	0	desc_2
20251011110000	20251013110000	cod_3	11/10/2025 11:00	13/10/2025 11:00	0	desc_3
20251013110000	20251013110000	cod_4	13/10/2025 11:00	13/10/2025 11:00	0	desc_4

Figure 2.4: Consolidated master data table

2.5.1 First Run

The first step is the copy of the raw source data into the STG table; as part of the protocol, the records are augmented with the technical fields `JOBID` and `INS_TIME`.

CODE	DESCRIPTION	QUANTITY	PROVIDER_CODE	MONTH_CODE	JOBID	INS_TIME
cod_1	desc_1	0	pcode_1	mcode_1	20251207110000	7/12/25 11.00
cod_2	desc_2	1	pcode_2	mcode_2	20251207110000	7/12/25 11.00
cod_3	desc_3	2	pcode_3	mcode_3	20251207110000	7/12/25 11.00
cod_4	desc_4	3	pcode_4	mcode_4	20251207110000	7/12/25 11.00
cod_5	desc_5	-4	pcode_5	mcode_5	20251207110000	7/12/25 11.00

Figure 2.5: `STG` table at initial run

Then, a `MINUS` operation is applied to isolate new entries with respect to the previous run. With the assumption that this is the initial run and that therefore the *"Yesterday"* table is empty, all records are new and therefore inserted into the `DLT` table, with the addition of the third technical field `FLG_NEG=0`.

CODE	DESCRIPTION	QUANTITY	PROVIDER_CODE	MONTH_CODE	JOBID	INS_TIME	FLG_NEG
cod_1	desc_1	0	pcode_1	mcode_1	20251207110000	7/12/25 11.00	0
cod_2	desc_2	1	pcode_2	mcode_2	20251207110000	7/12/25 11.00	0
cod_3	desc_3	2	pcode_3	mcode_3	20251207110000	7/12/25 11.00	0
cod_4	desc_4	3	pcode_4	mcode_4	20251207110000	7/12/25 11.00	0
cod_5	desc_5	-4	pcode_5	mcode_5	20251207110000	7/12/25 11.00	0

Figure 2.6: `DLT` table

The next step is the data quality assessment and validation of all pending records before they are added to the OK table. This phase is organized as a sequential, two-step validation process:

- **Referential Integrity Check**: a lookup operation performed against the relevant master data table to verify the validity of all foreign keys (FKs)
- **Business Rules Compliance Check**: a validation step ensuring conformity with the business logic to ensure the correct data types and values for all fields, as well as checking coherence against the defined intervals

In this instance, both the Referential Integrity and Business Rules checks for the last record in the DLT table fail, as it references a product with a non-existing code (`cod_5`) and contains an invalid quantity value (`-4`), as only positive quantities are considered valid. Consequently, this record is flagged as invalid and is deferred to the `E$` table, ensuring that only consistent and compliant data are propagated downstream.

CODE	DESCRIPTION	QUANTITY	PROVIDER_CODE	MONTH_CODE	JOBID	INS_TIME	FLG_NEG
cod_1	desc_1	0	pcode_1	mcode_1	20251207110000	7/12/25 11.00	0
cod_2	desc_2	1	pcode_2	mcode_2	20251207110000	7/12/25 11.00	0
cod_3	desc_3	2	pcode_3	mcode_3	20251207110000	7/12/25 11.00	0
cod_4	desc_4	3	pcode_4	mcode_4	20251207110000	7/12/25 11.00	0
cod_5	desc_5	-4	pcode_5	mcode_5	20251207110000	7/12/25 11.00	0

Figure 2.7: Errors invalidating the record

A `ROW_NUMBER()` operation is then applied to the `DLT` to guarantee that only the most recent and unique records are selected for insertion into the `OK` table. This step prevents duplicate or outdated records from being processed, maintaining the integrity and accuracy of the data pipeline.

ROW_NUMBER()	CODE	DESCRIPTION	QUANTITY	PROVIDER_CODE	MONTH_CODE	JOBID	INS_TIME	FLG_NEG
1	cod_1	desc_1	0	pcode_1	mcode_1	20251207110000	7/12/25 11.00	0
1	cod_2	desc_2	1	pcode_2	mcode_2	20251207110000	7/12/25 11.00	0
1	cod_3	desc_3	2	pcode_3	mcode_3	20251207110000	7/12/25 11.00	0
1	cod_4	desc_4	3	pcode_4	mcode_4	20251207110000	7/12/25 11.00	0

Figure 2.8: `ROW_NUMBER()` computation

Finally, both the `OK` table and the `E$` table are updated with the new records. According to the implemented logic, records in the `E$` table that failed validation or were otherwise deferred will either be re-processed in a subsequent ETL run (through a specific **Recycle Error** procedure) or sent to the client for further examination and resolution. This ensures a robust mechanism for handling exceptions while preserving the overall consistency and reliability of the ETL workflow.

Once the `OK` table is consolidated, the final step of the ETL process is the promotion of the processed data to the `ODS` through a loading operation, executed via a `MERGE` statement, which ensures that all records are unique, fully validated,

CODE	DESCRIPTION	QUANTITY	PROVIDER_CODE	MONTH_CODE	JOBID	INS_TIME	FLG_NEG
cod_1	desc_1	0	pcode_1	mcode_1	20251207110000	7/12/25 11.00	0
cod_2	desc_2	1	pcode_2	mcode_2	20251207110000	7/12/25 11.00	0
cod_3	desc_3	2	pcode_3	mcode_3	20251207110000	7/12/25 11.00	0
cod_4	desc_4	3	pcode_4	mcode_4	20251207110000	7/12/25 11.00	0

Figure 2.9: Final `OK` table after the first run

CODE	DESCRIPTION	QUANTITY	PROVIDER_CODE	MONTH_CODE	JOBID	INS_TIME	FLG_NEG
cod_5	desc_5	-4	pcode_5	mcode_5	20251207110000	7/12/25 11.00	0

Figure 2.10: Final `E$` table after the first run

and secured according to established data governance policies. Additionally, two technical metadata fields are appended and maintained: the last update timestamp (`UPD_TIME`) and the last updating job identifier (`JOBID_UPD`), fields which are consistently updated during each ETL run only if changes (updates) occur in the original record. This mechanism ensures that the data in the ODS layer remains the definitive, unique, and up-to-date view of the source system's current state, facilitating audits and supporting downstream processes that rely on the most current and accurate dataset.

CODE	DESCRIPTION	QUANTITY	PROVIDER_CODE	MONTH_CODE	JOBID	INS_TIME	FLG_NEG	UPD_TIME	JOBID_UPD
cod_1	desc_1	0	pcode_1	mcode_1	20251207110000	7/12/25 11.00	0	7/12/25 11.00	20251207110000
cod_2	desc_2	1	pcode_2	mcode_2	20251207110000	7/12/25 11.00	0	7/12/25 11.00	20251207110000
cod_3	desc_3	2	pcode_3	mcode_3	20251207110000	7/12/25 11.00	0	7/12/25 11.00	20251207110000
cod_4	desc_4	3	pcode_4	mcode_4	20251207110000	7/12/25 11.00	0	7/12/25 11.00	20251207110000

Figure 2.11: Final `ODS` table with consolidated and latest data

2.5.2 Second Run

The second execution (Second Run) introduces complexity by processing changes in the source data; specifically, there is a modification in a product's description, while other product records are physically removed from the source table, as detailed in Figure [2.12].

CODE	DESCRIPTION	QUANTITY	PROVIDER_CO	MONTH
cod_1	desc_1	0	pcode_1	mcode_1
cod_2	desc_2	1	pcode_2	mcode_2
cod_3	~~desc_3~~ desc_6	2	pcode_3	mcode_3
~~cod_4~~	~~desc_4~~	~~3~~	~~pcode_4~~	~~mcode_4~~
~~cod_5~~	~~desc_5~~	~~4~~	~~pcode_5~~	~~mcode_5~~

Figure 2.12: Source table at the second run

The `STG` and `DLT` tables maintain a record of all runs for debugging, auditing, and historical analysis purposes, as well as to correctly identify new records with respect to the previous runs.

In this instance, the cancellation of the records with `cod_4` and `cod_5` is logically reflected in the `DLT` table by appending the records and setting the `FLG_NEG=1`. Similarly, the update of the record with `cod_3` is modeled as the cancellation of the previous record, followed by the insertion of the second version.

Since both modifications satisfy the business rules and the integrity constraints and are therefore passed to the `OK` table, they are consequently promoted to the `ODS` table, as shown in Figures [2.16, 2.17]. The cancellation of the record with `cod_5`, instead, is still an error as the referential integrity is not yet satisfied.

CODE	DESCRIPTION	QUANTITY	PROVIDER_CODE	MONTH_CODE	JOBID	INS_TIME
cod_1	desc_1	0	pcode_1	mcode_1	20251207110000	7/12/25 11.00
cod_2	desc_2	1	pcode_2	mcode_2	20251207110000	7/12/25 11.00
cod_3	desc_3	2	pcode_3	mcode_3	20251207110000	7/12/25 11.00
cod_4	desc_4	3	pcode_4	mcode_4	20251207110000	7/12/25 11.00
cod_5	desc_5	-4	pcode_5	mcode_5	20251207110000	7/12/25 11.00
cod_1	desc_1	0	pcode_1	mcode_1	20251208110000	8/12/25 11.00
cod_2	desc_2	1	pcode_2	mcode_2	20251208110000	8/12/25 11.00
cod_3	desc_6	2	pcode_3	mcode_3	20251208110000	8/12/25 11.00

Figure 2.13: `STG` after the second run, keeps track of the previous records

CODE	DESCRIPTION	QUANTITY	PROVIDER_CODE	MONTH_CODE	JOBID	INS_TIME	FLG_NEG
cod_1	desc_1	0	pcode_1	mcode_1	20251207110000	7/12/25 11.00	0
cod_2	desc_2	1	pcode_2	mcode_2	20251207110000	7/12/25 11.00	0
cod_3	desc_3	2	pcode_3	mcode_3	20251207110000	7/12/25 11.00	0
cod_4	desc_4	3	pcode_4	mcode_4	20251207110000	7/12/25 11.00	0
cod_5	desc_5	-4	pcode_5	mcode_5	20251207110000	7/12/25 11.00	0
cod_4	desc_4	3	pcode_4	mcode_4	20251208110000	8/12/25 11.00	1
cod_3	desc_3	2	pcode_3	mcode_3	20251208110000	8/12/25 11.00	1
cod_3	desc_6	2	pcode_3	mcode_3	20251208110000	8/12/25 11.00	0

Figure 2.14: `DLT` table after the second run

CODE	DESCRIPTION	QUANTITY	PROVIDER_CODE	MONTH_CODE	JOBID	INS_TIME	FLG_NEG	ROW_NUMBER()
cod_1	desc_1	0	pcode_1	mcode_1	20251207110000	7/12/25 11.00	0	
cod_2	desc_2	1	pcode_2	mcode_2	20251207110000	7/12/25 11.00	0	
cod_3	desc_3	2	pcode_3	mcode_3	20251207110000	7/12/25 11.00	0	
cod_4	desc_4	3	pcode_4	mcode_4	20251207110000	7/12/25 11.00	0	
cod_5	desc_5	-4	pcode_5	mcode_5	20251207110000	7/12/25 11.00	0	
cod_4	desc_4	3	pcode_4	mcode_4	20251208110000	8/12/25 11.00	1	1
cod_3	desc_3	2	pcode_3	mcode_3	20251208110000	8/12/25 11.00	1	1
cod_3	desc_6	2	pcode_3	mcode_3	20251208110000	8/12/25 11.00	0	0

Figure 2.15: `DLT` table after the `ROW_NUMBER()` operation is applied

CODE	DESCRIPTION	QUANTITY	PROVIDER_CODE	MONTH_CODE	JOBID	INS_TIME	FLG_NEG
cod_3	desc_6	2	pcode_3	mcode_3	20251208110000	8/12/25 11.00	0
cod_4	desc_4	3	pcode_4	mcode_4	20251208110000	8/12/25 11.00	1

Figure 2.16: `OK` table after the second run

CODE	DESCRIPTION	QUANTITY	PROVIDER_CODE	MONTH_CODE	JOBID	INS_TIME	FLG_NEG	UPD_TIME	JOBID_UPD
cod_1	desc_1	0	pcode_1	mcode_1	20251207110000	7/12/25 11.00	0	7/12/25 11.00	20251207110000
cod_2	desc_2	1	pcode_2	mcode_2	20251207110000	7/12/25 11.00	0	7/12/25 11.00	20251207110000
cod_3	desc_6	2	pcode_3	mcode_3	20251207110000	7/12/25 11.00	0	8/12/25 11.00	20251208110000
cod_4	desc_4	3	pcode_4	mcode_4	20251207110000	7/12/25 11.00	1	8/12/25 11.00	20251208110000

Figure 2.17: `ODS` table after the second run

It is important to notice the structural difference between the persistence layers: while the DLT stage is designed to maintain duplicate records, preserving one instance for each ETL run to maintain a full history of changes, the ODS must strictly enforce uniqueness based on the logical Primary Key.

To bridge this structural gap, a deduplication strategy is implemented during the OK phase using the `ROW_NUMBER()` analytic function. This mechanism ensures that only the most relevant version of each record reaches the `ODS`, resolving potential collisions caused by overlapping delta windows or multiple updates within a single batch. This function is computed as follows:

```
SELECT * FROM (
    SELECT CODE, PROVIDER_CODE, MONTH_CODE, QUANTITY, DESCRIPTION,
   JOBID, INS_TIME, FLG_NEG
    ROW_NUMBER() OVER (PARTITION BY CODE, PROVIDER_CODE, MONTH_CODE
    ORDER BY JOBID DESC, FLG_NEG ASC, INS_TIME DESC) AS ROW_NUM
    FROM DLT_ORDERS
)
WHERE ROW_NUM = 1;
```

In this implementation, the logic follows these criteria:

- **Partition**: the data is grouped by its logical Primary Key (in this case, `CODE`, `PROVIDER_CODE`, and `MONTH_CODE`). This ensures that the ranking is local to each specific entity
- **Ordering**: within each group, records are sorted, ensuring that only the most recent version of the record (the "latest truth") receives the rank of 1
- **Filtering**: during the insertion into the `OK` table, only records where `ROW_NUMBER=1` are selected

As shown in Figures [2.8, 2.15], the `ROW_NUMBER()` assigns 1 to the record with the most recent `INS_TIME` and the highest `FLG_NEG`. This mechanism effectively resolves collisions before the `MERGE` operation into the `ODS`, preventing primary key violations and ensuring that the final dataset reflects the latest available state of the source system.

Chapter 3

The Boomi Platform

"Boomi is a pioneer in the Integration Platform as a Service (iPaaS) space" [5], providing a cloud-native, multi-tenant environment designed to build integration processes, manage APIs, and automate workflows. By leveraging a no-code/low-code approach, Boomi makes process design intuitive and immediate, supporting users through extensive documentation and a developers' community.

This architecture is capable of handling diverse integration needs, ranging from cloud-to-cloud and SaaS-to-SaaS scenarios to cloud-to-on-premise data exchanges.

3.1 Background

At this stage, it is helpful to outline the key features of iPaaS and SaaS solutions, their similarities, and differences.

SaaS or Software as a Service, refers to application software hosted in the cloud and delivered over the Internet to compatible devices. SaaS providers operate and manage both the application and its underlying infrastructure, freeing organizations from installation, maintenance, and upgrades. Instead of purchasing and installing software locally, users create an account, subscribe to the service, and begin working immediately. This delivery model offers rapid deployment, predictable costs, and high accessibility, making it one of the most widely adopted paradigms in contemporary enterprise environments [10].

On the other hand, **iPaaS** refers to a suite of cloud-based tools designed to integrate data and applications distributed across heterogeneous systems. It allows organizations to build and deploy integration flows that connect systems hosted in public and private clouds, as well as it allows to connect cloud environments and on-premise data centers. The rise of iPaaS is largely tied to the rapid expansion of SaaS applications, as they are a compelling choice for addressing specific business or administrative needs; however, this ease of use often leads individual departments to independently acquire SaaS tools, resulting in a fragmented and increasingly complex ecosystem of cloud-based applications. iPaaS platforms address this issue by providing a centralized layer for data flows orchestration and ensuring consistent communication among disparate systems.[3].

3.2 Main Functionalities and Components

The core strength of the Boomi platform lies in its set of user-friendly components that accelerate the development lifecycle. Its architecture emphasizes reusability while allowing ease of adoption and configuration, enabling both technical and semi-technical users to design sophisticated integration flows with minimal effort.

One of Boomi's defining attributes is its drag-and-drop configuration environment, which abstracts much of the complexity that is traditionally associated with integration development. Instead of writing large amounts of code, developers can focus on defining business logic, data transformations, and error-handling strategies, thus reducing both development time and the risk of implementation errors.

A key enabler of this rapid development model is Boomi's extensive library of pre-built connectors. These connectors provide ready-made interfaces to a wide range of systems (e.g., SaaS applications, on-premises software, relational databases, file systems) and automatically handle authentication, data retrieval, and protocol management. Data can be ingested in standard formats such as JSON, XML, or CSV, ensuring compatibility across heterogeneous environments.

The **Process Canvas** serves as the main visual workspace for constructing integration, based on a rich set of configurable **Process Shapes/Steps**, such as Start, Stop, Map, Decision, Branch, and Try/Catch. They define the logic and data flow of the process, each one encapsulating a specific behavior, enabling precise control over how data is retrieved, transformed, validated, routed, and ultimately delivered to its destination.

Another core component that contributes to Boomi's development efficiency is its **Profile** system, which defines the structure of the information handled within an integration process, specifying fields, data types, nested hierarchies, constraints, and validation rules. Boomi supports profiles for multiple formats, including JSON, XML, CSV, Flat Files, and Database schemas; once a profile is defined, it ensures consistent interpretation of incoming and outgoing data (regardless of their source), simplifies mapping operations, and reduces the possibility of structural inconsistencies across different integration steps.

Profiles are tightly integrated with the Map component, enabling developers to visually align source and destination fields or apply advanced scripting when needed. To accelerate this stage, Boomi leverages **Generative AI** and crowdsourced intelligence through tools like Boomi GPT and Boomi Suggest; the latter, specifically, recommends probable mappings based on Machine Learning models trained on thousands of anonymized integrations.

To further boost productivity beyond the mapping phase, the platform offers the **Process Library**, providing pre-built, reusable templates, and the *"Quick Start Wizard"*, which automates integration setup through just a few simple inputs. These capabilities, combined with AI-driven prompts, allow developers to transition rapidly from initial design to a fully functional workflow.

3.2.1 Runtime Environment Architecture

The Boomi Runtime Environment Architecture, known as **AtomSphere**, defines where and how integration processes run [11]. It is composed of Boomi Atoms, Molecules, and Atom Clouds, execution engines that allow flexible deployment across a variety of infrastructures.

A Runtime Cloud introduces forked execution, ensuring each integration process runs in a separate Java Virtual Machine (JVM). This isolation, coupled with a High Security Policy, enhances stability and data separation among tenants.

Boomi offers two primary categories of runtime cloud environments:

- Boomi-Hosted Runtime Clouds: fully managed and maintained by Boomi across geographically distributed regions (Americas, EMEA, APAC)
- Private Runtime Clouds: installed on the client's infrastructure, they are chosen when strict security or regulatory constraints apply, or when their workloads demand high processing capacities. Private Clouds also support cluster sharing across multiple customer accounts and provide full control over hardware resources, networking, and public endpoints.

3.3 Integration Lifecycle

The platform supports a streamlined lifecycle from development to deployment.

- **Build and Test**: developers build the process using the Process Canvas, then they test it directly within the Build page, eliminating the need for immediate deployment to a staging or production environment
- **Deployment**: once testing is complete, the integration process is packaged as a component and deployed to the chosen runtime environment (Atom, Cluster, or Cloud). The deployed component contains all necessary elements, including connectors, transformation rules, decision handling, and the core processing logic, to run the process end-to-end
- **Execution and Monitoring**: the deployed process is executed by the runtime engine. The platform provides robust tools for monitoring the execution, such as logs, performance metrics, and error handling

3.3.1 Deployment

Particular attention must be given to the deployment stage, as it represents the most critical phase in the integration lifecycle, transitioning the workflow from the development environment into a state ready for operational execution.

Although testing merely validates the correctness and coherence of the data flow in a controlled setting, deployment marks its promotion into a live, production-ready environment. It is the essential prerequisite for executing the ETL process against actual source systems and enabling reliable end-to-end data movement; moreover, once deployed, the process can be scheduled for recurring and fully automated executions (e.g., daily, hourly, or event-triggered), thus supporting consistent and predictable data-synchronization cycles.

Boomi provides a dedicated interface for managing the deployment lifecycle through the creation of **Packaged Components**, which encapsulate all necessary connectors, transformation maps, configuration parameters, and business logic. into versioned deployment units. Once compiled, these packages are formally assigned to the selected runtime environment (Atom, Molecule, or Cloud), while a separate **Deploy** section enables authorized specialists to manage version history, monitor execution logs, and perform operational actions such as pausing or rolling back processes. This structured model effectively ensures controlled release management, maintaining the consistency and traceability of all runtime operations within a unified administrative framework.

3.4 Advantages and Limitations

Boomi presents several strengths that make it a compelling solution in the integration landscape. Its intuitive low-code, drag-and-drop development environment enables rapid prototyping and significantly reduces the need for extensive programming knowledge, allowing developers to focus on business logic rather than the underlying technical complexity, thus providing a deep acceleration in development, directly addressing the client's need for improved efficiency by shortening the time-to-market for new integrations. Additionally, the extensive library of pre-built connectors abstracts authentication and connection management complexities, enabling the development team to concentrate primarily on defining business transformation logic.

The configuration of the platform is highly flexible and promotes reusability through shared components, templates, and an extensive Process Library, supporting consistent development practices across projects, ensuring that processes are streamlined and efficient. Boomi also provides a broad ecosystem of connectors for SaaS applications, on-premise systems, databases, file repositories, and APIs, effectively abstracting protocol handling and data access concerns. The deployment of a local Boomi AtomSphere runtime demonstrated the platform's ability to support seamless hybrid integration, ensuring secure and low-latency processing for sensitive data stored in on-premises Oracle Databases while maintaining cloud-native scalability and centralized governance for monitoring and auditing.

Boomi's mapping capabilities and data profiling system offer robust support for data transformation and validation, further enhanced by features such as Boomi Suggest and Boomi GPT, which accelerate development through automated recommendations and AI-assisted configuration. Combined with scalable runtime engines, a cloud-native execution model, and comprehensive monitoring tools, the platform enables the creation of efficient, maintainable, and easily deployable integration workflows. Moreover, advanced features such as DataHub, Boomi Data Integration (formerly Rivery) [12], and AI Agents [6] extend the platform beyond traditional integration use cases, enabling the design of more complex workflows that leverage recent advancements in areas such as machine learning, artificial intelligence, and master data management.

Despite these advantages, practical implementation revealed several technical constraints, particularly in relation to database optimization and complex interactions. Boomi's support for SQL is relatively limited, as some connectors do not allow advanced queries, and operations that would typically be handled efficiently through a single database statement often need to be decomposed into multiple process steps within the integration flow. Consequently, logic that would normally execute within the database must be re-implemented using Boomi shapes, increasing process complexity and potentially impacting performance.

Parameter handling can also be inconsistent across connectors, complicating query execution and introducing additional sources of error. More broadly, the platform is not optimized for heavy ETL workloads or high-volume, database-centric batch operations, as large datasets may lead to performance degradation or require partitioning into smaller chunks, thereby increasing operational overhead. Although the visual interface improves accessibility, complex processes with deeply nested components can still become challenging to debug. Finally, like many iPaaS solutions, Boomi relies on a usage-based cost structure and proprietary mechanisms, which may increase long-term dependency and complicate future migration efforts.

Ultimately, the project highlighted a critical trade-off: the rapid development enabled by the low-code environment comes at the expense of the advanced SQL optimization and execution efficiency typically available in traditional, highly customized database-centric solutions.

3.5 Mediamente Consulting Case Study

During my internship at Mediamente Consulting, a client proposed the migration of an existing ETL process onto the Boomi platform, as mentioned beforehand, to minimize operational costs and improve efficiency by leveraging its cloud-native capabilities. Beyond cost reduction, the project sought to introduce technological innovation by modernizing the integration approach, thereby increasing scalability and ensuring greater flexibility in adapting workflows to evolving business requirements. To successfully address this request while establishing a robust foundational blueprint for future Boomi-based projects, this thesis involved a systematic exploration of the platform's core functionalities and advanced features.

This transition from strategy to execution began with the critical challenge of establishing a stable and secure bridge between the cloud-based control plane and the client's on-premises Oracle Database infrastructure. Consequently, while we leveraged the native **Oracle Database Connector** [13], the setup required the deployment of a Boomi AtomSphere runtime engine, specifically an Atom installed within the client's network perimeter, to facilitate a secure, low-latency connection directly to the Oracle instances. This hybrid configuration effectively demonstrated the platform's capacity for secure integration, allowing cloud-managed processes to communicate with local data without exposing sensitive information to the public Internet.

Following the establishment of this connection, the core implementation phase started

under a database-centric approach. Although the objective was to transform legacy ETL logic into a modern integration flow, the Oracle environment remains the baseline for all data operations; therefore, Boomi acts as the high-performance orchestration engine while each stage concludes with a physical commit to the Oracle Database. In this framework, the tables `STG`, `DLT`, `OK`, and `E$`, which replicate the proprietary ETL flow detailed in the previous chapter, are not internal Boomi structures but objects explicitly defined within the database schema. This hybrid approach ensures that while Boomi orchestrates the logic, the database provides durable storage and auditability for every transformation step.

Chapter 4

Technical Specifications of Project Components

The project's initial phase was conceived as a thorough technological evaluation; its principal objective extends beyond a technical migration, emphasizing an analytical exploration of the Boomi platform's inherent architectural capabilities. This process involved evaluating an extensive array of components to determine which could most effectively translate the established logic of Mediamente Consulting's framework into a low-code environment.
Consequently, the process evolved into a meticulous architectural alignment, bridging the gap between the business requirements and Boomi's specific functional paradigms to ensure an optimal architectural fit.

In order to ensure a thorough understanding of the implementation phase, it is necessary to present the key Boomi components [11] and explain their roles within the ETL process.

4.1 Connection to the Database

The client's core operational data is stored in an **Oracle** environment using a relational tabular structure. Establishing a robust connection to this database was a critical prerequisite for the extraction phase, ensuring the integration platform could access the required datasets while adhering to strict security protocols; this included managing sensitive credentials and defining granular access permissions to maintain data integrity.

To align with the client's on-premise operational model, the integration architecture utilizes a local **Atom** (the platform's lightweight runtime engine), deployed directly within the internal network, serving as a secure gateway between the Boomi cloud and the on-premise data environment. By enabling processing behind the firewall, this setup guarantees that data extraction occurs within a protected environment, thereby maintaining a high security standard and avoiding the necessity of opening database ports to external traffic.

4.1.1 Oracle Database Connector

While the Boomi platform offers several alternatives, such as the Database (Legacy) Connector, which is employed for specialized tasks, the **Oracle Database Connector** was selected as the primary interface for this project. This component enables bidirectional data exchange by providing secure and efficient access to the Oracle environment [13]; it leverages the Oracle JDBC (OJDBC) Driver to communicate with the database, allowing the execution of fundamental CRUD (Create, Read, Update, Delete) operations.

In the Boomi ecosystem, the configuration of this component is decoupled into two distinct entities to maximize flexibility and security:

- **Connection:** represents the physical and logical link to the target Oracle instance, encapsulating the connection URL (specifying the hostname and port), the JDBC Driver class name, and the encrypted authentication credentials. By isolating these parameters, Boomi allows for environment-specific configurations without altering the underlying integration logic

- **Operation:** defines the function that will be executed once the session is established. It maps standard SQL commands to Boomi-specific actions, such as `GET` (for `SELECT` statements) or `SEND` (for `INSERT`, `UPDATE`, and `UPSERT`). Furthermore, the Operation is responsible for generating the **Database Profile** (leveraging the *"Import Operation"* option natively supported by this connector), which acts as the metadata schema for both request and response, dictating how the data is structured within the Boomi pipeline

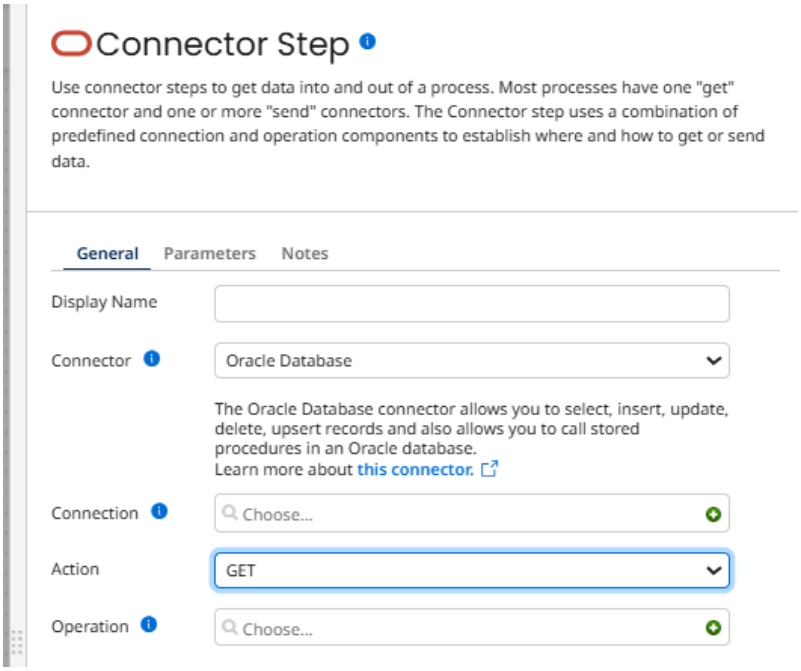


Figure 4.1: General configuration of the Oracle Database Connector

Both the Connection and Operation are treated as modular, standalone assets, and this modularity ensures that a single validated Connection can be referenced by

multiple Operations across different integration processes, maintaining consistency and reducing overhead throughout the entire ETL framework.

4.2 Boomi Properties

Particular interest was placed on **Dynamic Process Properties** (DPPs), as they proved instrumental in orchestrating the ETL logic and global flow management. Unlike document-level variables, DPPs establish properties at the process level, ensuring that assigned values persist and remain accessible to any shape or subprocess throughout the entire execution lifecycle.

4.2.1 Set Properties

The Set Properties shape plays a fundamental role in maintaining process coherence and traceability by defining and assigning values to the DPPs that drive the framework's logic.

In the context of this project, DPPs enabled the systematic assignment of both the `LastSuccessfulExecutionDate` (representing the `JOBID` of the previous run) and the current `JOBID` (the current execution timestamp). By persisting these values at the process level, the workflow guarantees coherence across all subsequent stages of `STG`, `OK`, and `ODS`, as well as during the differential comparison logic implemented in the `DLT MINUS` operation. This ensures that every component within the pipeline relies on the same execution metadata, eliminating inconsistencies and preventing temporal misalignment in data extraction or validation.

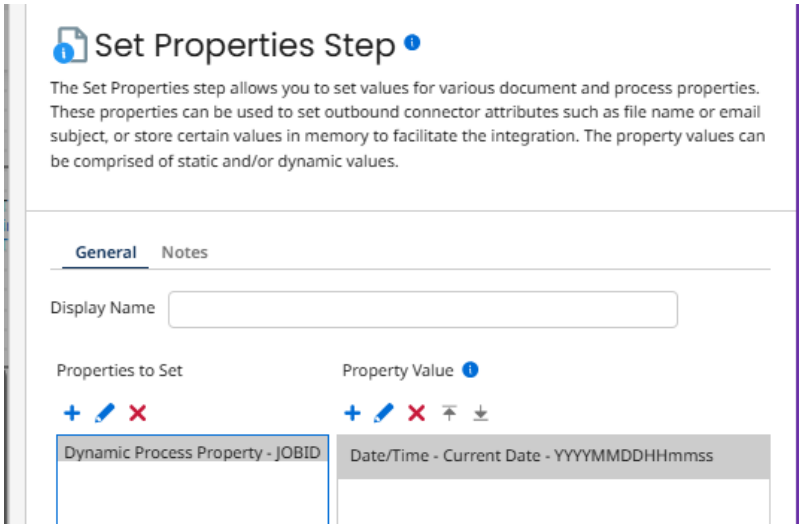


Figure 4.2: General configuration of the Set Properties Step, with `JOBID` set as the current execution date

While Boomi provides a diverse ecosystem of property types to manage data and metadata, DPPs possess unique characteristics that make them optimal for complex, end-to-end ETL governance.
To better understand their necessity, it is useful to compare them with other standard Boomi properties:

- **Process Properties:** defined at design time, these properties are primarily used for static configuration values; they are read-only and therefore cannot be updated at runtime [14]

- **Document Properties:** attached to individual documents flowing through the process; their values may differ on a per-document basis and are therefore unsuitable for managing global metadata [15]

In contrast, Dynamic Process Properties serve as a mutable, process-wide storage mechanism, featuring [16]:

- Process-Level Scope and Persistence: they are defined at the start of an execution and persist for the entire duration of a process run. Once set, they remain available and stable until the execution terminates, ensuring consistent access to shared variables across the entire workflow, including nested subprocesses

- Mutability and Dynamic State Management: unlike static Process Properties, DPPs can be updated at runtime, allowing them to support dynamic state management, such as storing iteration counters, tracking conditional flags, or updating metadata

- Cross-Component Accessibility: they can be retrieved or updated by any Process Shape without requiring specialized configuration

- Non-Document-Specific Behavior: they remain constant for the entire process run, regardless of the number or content of the documents being processed. This characteristic makes them the designated type for storing global metadata crucial for governance, such as timestamps, execution counters, and run identifiers

Leveraging these properties, the ETL framework achieves the necessary stability to orchestrate global actions and maintain a consistent data lineage across all stages of the integration pipeline.

4.3 Structuring the Project Flow

The ETL framework adopts a modular architectural pattern to prioritize scalability and compliance while ensuring high standards of readability and robust error management. Rather than developing a single, monolithic process that would be difficult to maintain and troubleshoot, each functional stage of the ETL pipeline was encapsulated into independent, specialized processes, and subsequently orchestrated and invoked through the Process Call shape, creating a hierarchical *"Parent-Child"* relationship between the various stages of the workflow.

4.3.1 Process Call Step

The Process Call is a fundamental shape in Boomi's orchestration logic, enabling the decoupling of complex integration flows into manageable units and ensuring a clean separation of concerns by delegating tasks from the main execution thread to dedicated subprocesses.

This modular design significantly enhances error containment, as any exception is limited to the specific subprocess where it occurs, preventing failure propagation across the entire pipeline while allowing for stage-specific handling strategies. Furthermore, the adoption of Process Calls facilitates granular logging and traceability; Boomi's Process Reporting provides a modular view of execution metadata, enabling developers to independently audit application responses for each subprocess.

Finally, by configuring these subprocesses for synchronous execution, the parent process maintains strict control over the execution order, ensuring that each ETL stage proceeds only upon the successful termination of the previous one.

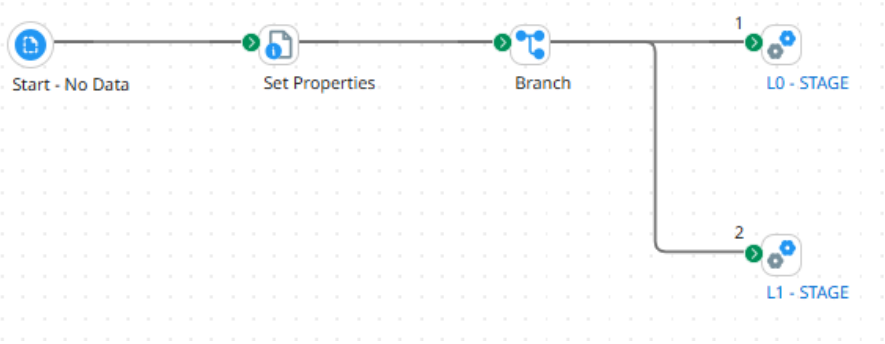


Figure 4.3: Practical application of the Process Call hierarchy within the ETL pipeline

4.3.2 Decision Step

The Decision Step provides a fundamental mechanism for runtime conditional branching, as it evaluates a specified field, property, or variable against a defined value, or another property, effectively implementing an *"if-then-else"* logic flow. Based on a Boolean result (True/False), documents are routed down two distinct paths.

The platform supports a wide array of comparison operators, including:

- String Comparisons: Equal To, Not Equal To, Matches Regular Expression, Matches Wildcards
- Numeric Comparisons: Greater Than, Less Than, Greater Than or Equal To, Less Than or Equal To

4.3.3 Business Rules

The Business Rules Step is designed for multi-layered data validation, facilitating the execution of complex logic by systematically evaluating multiple rules against the input data. These rules can be grouped using logical operators such as `AND` or `OR`, enabling the framework to perform sophisticated integrity checks before data is committed to downstream systems. A key strength of this component is its ability to integrate various input functions, including string manipulations, numeric calculations, or SQL Lookups, directly within the validation criteria. This shape is particularly effective for identifying "rejected" documents that do not meet the required business criteria without halting the entire process.

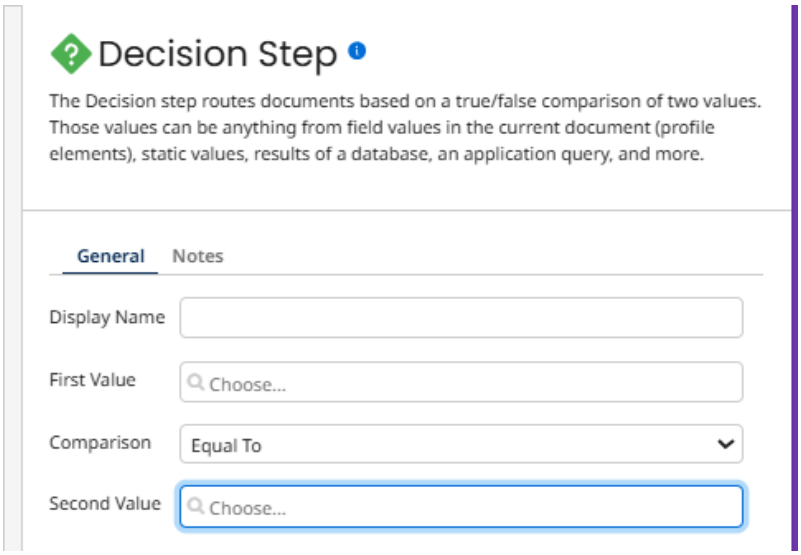


Figure 4.4: General configuration of the Decision Step

In the context of this project, the Business Rules Step serves as a critical gateway for enforcing **Validation Logic** and **Referential Integrity**. By incorporating SQL Lookups as part of the rule evaluation, the framework can dynamically verify if foreign key identifiers exist within the corresponding master tables in the Oracle Database, ensuring that only records with valid logical links are promoted to the `OK` stage. If the SQL query returns a *null* or zero count, the document is automatically routed to the rejection path, where it can be logged in the `E$` error tables for further audit, thus maintaining the overall relational consistency of the ETL pipeline.

However, the high degree of granularity and the ability to nest multiple functions introduce a significant operational drawback: the configuration phase is notoriously time-consuming. Each rule and function must be mapped manually, and the lack of a bulk-editing interface for complex logic can lead to a slower development lifecycle when compared to script-based alternatives.

4.4 Data Manipulation and Transformation

Once extracted, the data undergoes a series of transformations to ensure structural and semantic alignment with the target architecture. This phase is not merely a format conversion, acting as a data enhancement layer; it can integrate multi-stage refinements where datasets are cross-referenced to enrich the record's context, and conditional logic is applied to resolve inconsistencies, ensuring high-fidelity information flow.

4.4.1 Map Step

The Map Step is the most critical component within the Boomi platform, serving as the engine for data transformation. It facilitates the translation of data from a Source Profile (input) to a Target Profile (output) through a visual, drag-and-drop interface.

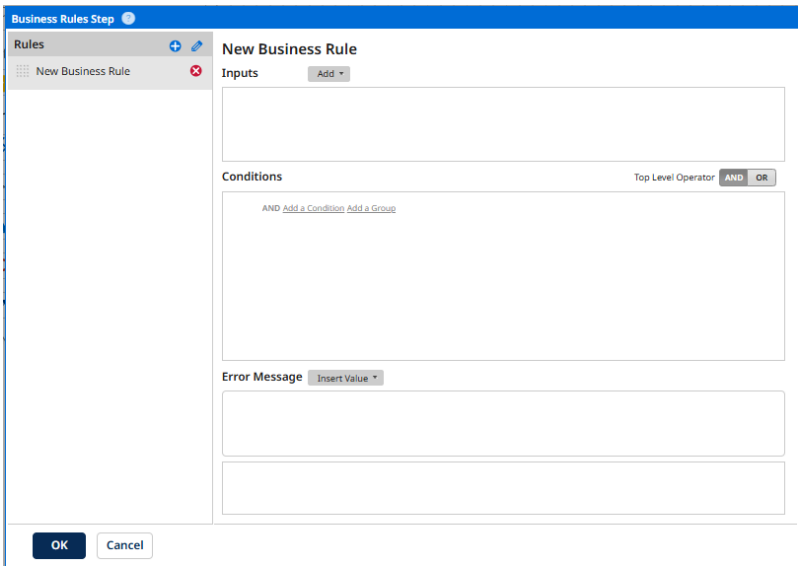


Figure 4.5: General configuration of the Business Rules Step

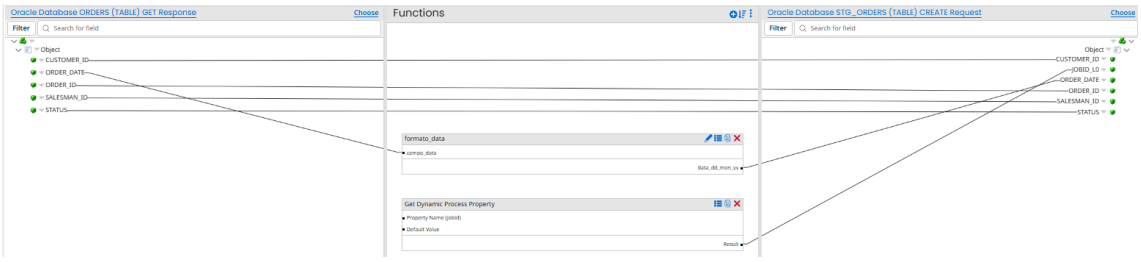


Figure 4.6: Practical implementation of the `STG` transformation logic within the Map Step of the ETL framework

Beyond simple field-to-field linking, the Map Step allows for the integration of Map Functions, which can be used to enrich or modify data during the transformation. These functions include:

- **Standard Functions:** pre-built operations to perform standard data manipulation without custom code
  - String: concatenation, substring, trim, replace, length, character conversion
  - Numeric: addition, subtraction, multiplication, division, rounding, absolute value
  - Date/Time: format date, get current date
  - Language: character conversion
- **Lookup Functions:** allow the process to retrieve additional informations from external sources or internal memory
  - SQL Database Lookup: executes a query against a relational database to retrieve related data
  - Document Cache Lookup: highly efficient mechanism to retrieve data previously stored in memory

  - Cross-reference tables: static mapping tables used to translate values between different systems

- **Custom Scripting:** enhance flexibility to implement complex and custom-tailored logic
  - JavaScript for custom logic
  - Groovy for advanced transformations
- **Other Built-in Functions:** specialised operations that interact with the integration platform's environment
  - Connector Call: directly invokes a connection to an application to retrieve or validate data
  - Properties: get or set dynamic process properties, document properties, process properties
- **User Defined Functions (UDF):** modular containers that allow for the encapsulation and reuse of complex mapping logic across multiple processes

While the Map Step is highly versatile, manual mapping for large enterprise profiles with hundreds of fields can be labor-intensive. To mitigate this, the platform offers **Boomi Suggest**, an AI-driven feature that employs Machine Learning to recommend mappings automatically; however, manual verification remains a best practice to ensure architectural accuracy and compliance with specific client requirements.

Furthermore, the Map Shape handles format conversions, allowing for the translation between heterogeneous formats such as JSON to CSV, or XML to Database profiles, and so on.

4.4.2 Data Process Step

The Data Process Step is used to perform document-level operations that go beyond simple field mapping, providing advanced capabilities for manipulating the content and structure of documents within a Boomi process. Common functionalities include:

- **Document Splitting and Merging:** break a single document into multiple documents based on defined criteria, or combine multiple documents into one
- **Search and Replace:** locate specific values or patterns within the document and replace them dynamically
- **Data Transformation Utilities:** apply encoding (Base64, URL), encryption/decryption, and compression/decompression
- **Advanced Processing:** execute scripts or apply custom logic for specialized transformations

This step is essential for manipulating entire documents rather than individual fields, enabling flexible and powerful data handling within Boomi integrations.

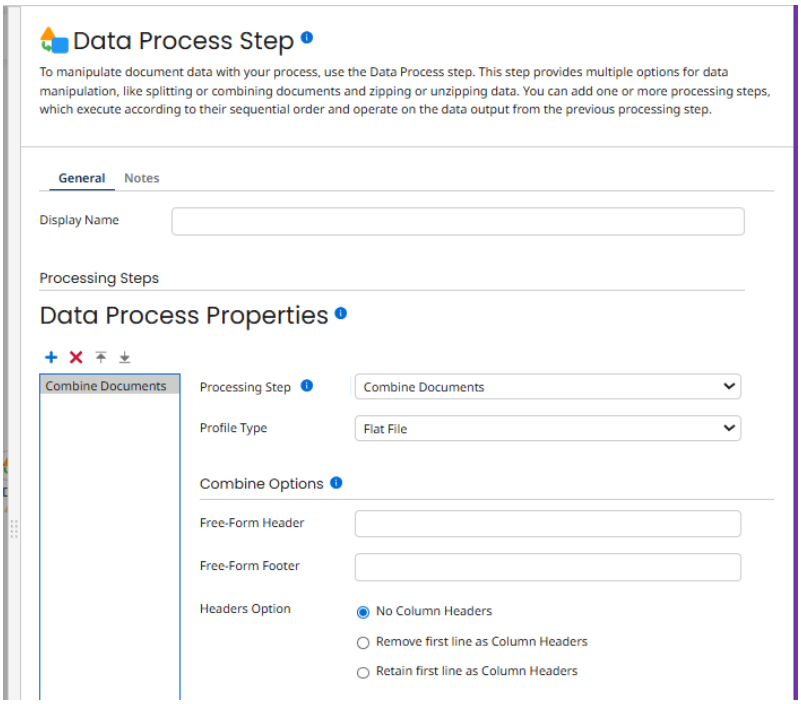


Figure 4.7: General configuration of the Data Process Step

4.4.3 Cache

The Cache component is vital for maintaining a runtime repository of data that can be referenced across different processes or branches. Since Boomi processes data in a stateless manner, meaning a document's profile is often discarded after a transformation, the cache provides a way to persist data without re-querying the source database.

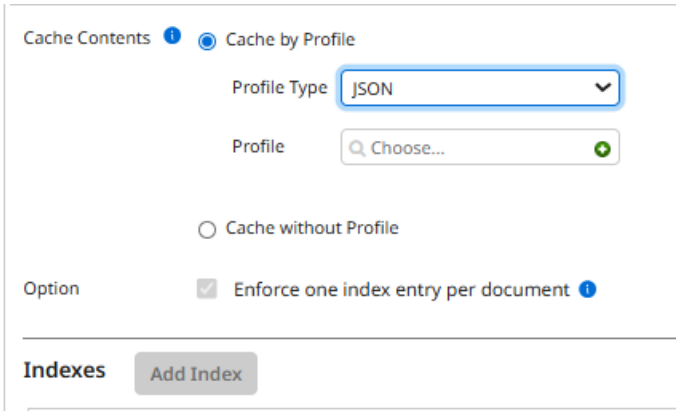


Figure 4.8: General configuration of the Cache component

Key functionalities include:

- **Lookup Operations:** enabling the retrieval of specific records using a defined key (index) from within a Map or a subsequent process
- **Performance Optimization:** reducing latency by avoiding redundant Connector calls to the on-premises Oracle Database

In the earliest stages of this project, the cache was instrumental in the implementation of the Delta Logic, enabling tracking insertions, updates, and deletions. In principle, the system must compare records from the previous run with those from the current run through a `MINUS` operation; since executing nested operations directly on the Oracle Database Connector was not feasible due to performance and architectural constraints, the operation was decoupled. The datasets were loaded into separate Document Caches, allowing the Boomi engine to perform a high-speed differential comparison (equivalent to the `MINUS` logic) in-memory, ensuring the accuracy of the incremental load.

4.4.4 Program Command

The Program Command Step is designed to execute database-specific or system-level commands as a discrete step in the integration lifecycle. This shape is particularly effective for orchestrating tasks that fall outside the scope of CRUD operations, such as database maintenance, executing complex conditional logic, or triggering external scripts.

It is important to emphasise that the Program Command shape is exclusively compatible with **Legacy Database connections**, in opposition to modern SDK-based connectors, such as the Oracle Database Connector, which rely on a structured Request/Response profile approach that abstracts the SQL layer. Whilst this is advantageous for standard operations, it is incompatible with the *"fire-and-forget"* execution model of the Program Command. Instead, by leveraging the Legacy interface, direct access to the JDBC layer can be gained, thus enabling the transmission of raw command strings directly to the database engine.

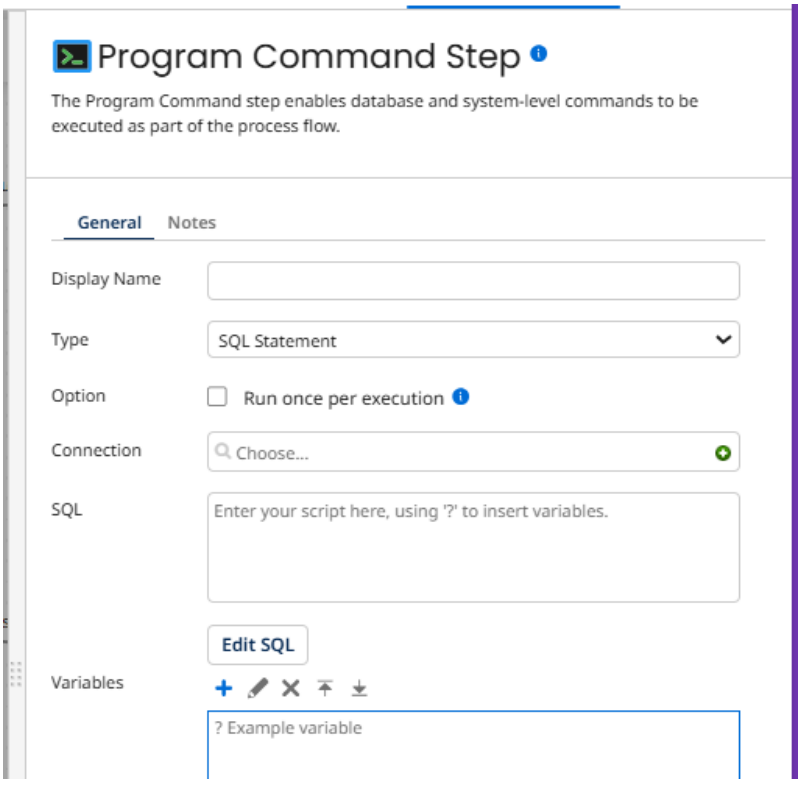


Figure 4.9: General configuration of the Program Command component

The component supports three primary execution types:

- **SQL Statements**: direct Data Manipulation Language (DML) or Data Definition Language (DDL) commands, frequently employed for complex `MERGE` (`UPSERT`) operations or bulk deletions where a standard Database connector would incur excessive performance overhead
- **Stored Procedures**: facilitates the invocation of pre-compiled logic that resides within a database engine, leveraging its computational power
- **System Commands**: execution of shell scripts, batch files, or executable binaries on the local runtime (Atom, Molecule, or Atom Cloud), commonly used for file system manipulation or triggering legacy third-party applications

In order to ensure operational flexibility, commands often require dynamic runtime values. However, it is important to acknowledge that parameters in the Program Command shape are strictly positional.
Placeholders (e.g., *"?"*) are mapped sequentially to the list of parameters defined in the component configuration; therefore, it is mandatory to ensure the exact correspondence of the Boomi parameter list to the expected sequence in the destination command. Any discrepancy in this sequence may result in a compromise to data integrity or trigger runtime exceptions due to data type mismatches.

4.5 Error Handling

To ensure the reliability of the integration flow, a comprehensive error-handling strategy is implemented across the entire pipeline, as managing exceptions in an enterprise environment is critical to prevent data loss and safeguard the stability of the overall system against localized transaction failures.

In particular, client notification mechanisms have been implemented across the entire integration flow, leveraging both native Boomi capabilities, such as connectors and process logging, as well as custom feedback channels to the Oracle system. The latter is mainly achieved through structured write-back operations to dedicated Oracle Database tables, enabling downstream systems to track processing status, errors, and business events in a consistent and auditable manner.

4.5.1 Message Step

The Message Step is a fundamental transformation component within the Boomi platform, designed for the programmatic construction of document payloads.

It relies on a token-based substitution mechanism, enabling the definition of a static template containing placeholders (e.g., $\{1\}, \{2\}, \ldots, \{n\}$) that are dynamically resolved at runtime. These parameters can be mapped from diverse data sources:

- **Profile Elements:** for extracting specific fields from XML, JSON, or Database documents currently in transit
- **Document Properties:** for capturing essential metadata and technical attributes, such as business rules error messages or connector-specific properties
- **Dynamic Process Properties:** for accessing state-variables and metadata persisted throughout the entire execution lifecycle

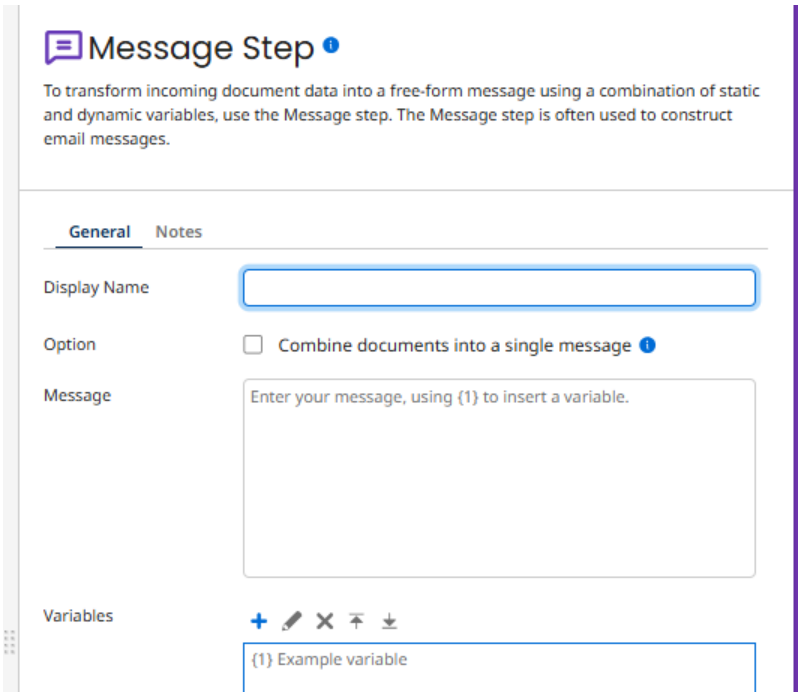


Figure 4.10: General configuration of the Message Step

- **Environment Extensions:** for retrieving context-specific values, such as environment names or endpoint URLs, ensuring the process remains portable across different stages

Within the implemented architecture, the Message Step captures the technical exception and wraps it with the specific business data that triggered the error, creating an error report that is then propagated downstream via the Mail Connector.

4.5.2 Mail Connector

The Mail Connector enables the automated exchange of information via standard protocols such as SMTP (for outbound dispatch) and POP3/IMAP (for message retrieval). When coupled with the Message Step and the Set Properties Step, this connector facilitates a dynamic alerting mechanism designed to enhance operational transparency; in particular, it leverages these components to construct email bodies or attachments programmatically, embedding specific data payloads or technical metadata associated with a failure event. This allows the system to capture the exact context that triggered an exception and propagate it downstream to the client, building a granular feedback loop that ensures the clients receive immediate insights into data discrepancies or process interruptions.

To ensure the successful execution of this notification service, several architectural prerequisites must be satisfied:

- Network Connectivity and Security: the Boomi Atom must have unrestricted outbound access to the mail server's IP and port (e.g., 25, 465, or 587); furthermore, the implementation of encryption protocols such as TLS/SSL is mandatory to ensure the integrity and confidentiality of the transmitted data

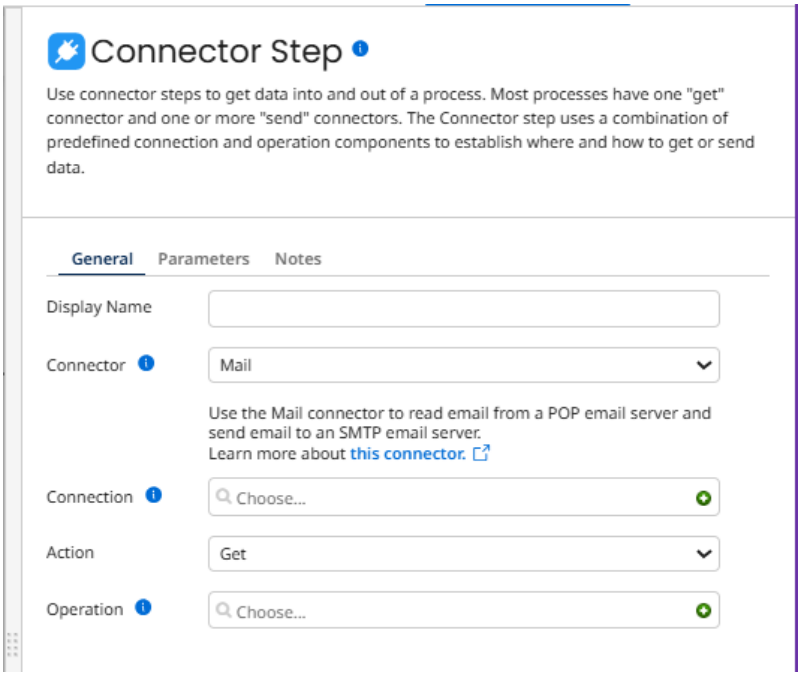


Figure 4.11: General configuration of the Mail Connector

- Authentication and Relay Permissions: valid service account credentials must be configured within the Mail Connection component
- Configuration Profile: correct definition of the mail server host, port, and protocol within the connection settings to align with the enterprise IT infrastructure.

Chapter 5

ETL Workflow Orchestration

In consideration of the sensitive and proprietary nature of the client's operational data, a rigorous anonymisation and data protection strategy has been adopted for this thesis project; the implementations described in the following sections do not rely on actual client data, but are based on a synthetic test dataset.
This controlled environment enabled the validation of functional logic, stress-testing of error-handling mechanisms, and assessment of the ETL workflow performance, without exposing real business intelligence or Personally Identifiable Information (PII). Accordingly, the results, logs, and profiles presented represent functional simulations intended to demonstrate the robustness and reliability of the Boomi implementation, rather than an exact representation of client data.

In alignment with the architectural framework detailed in Chapter 2, the project is structured into three distinct functional layers. This modular organization, independent of the specific data values processed, ensures a rigorous separation of concerns and enables granular governance throughout the data lifecycle:

- **L0:** focused on raw data extraction, initial ingestion from the source Oracle environment, and Delta computation
- **L1:** dedicated to data cleansing, schema alignment, and the application of Business Rules
- **L2:** responsible for the final promotion of validated records

Figure [5.1] provides a detailed overview of the relational schema and the synthetic dataset employed to validate the integration framework throughout this thesis project.

Table	Primary Key (PK)	Columns	Foreign Key (FK)
CONTACTS	contact_id	contact_id, first_name, last_name, email, phone, customer_id	customer_id
COUNTRIES	country_id	country_id, country_name, region_id	region_id
CUSTOMERS	customer_id	customer_id, name, address, website, credit_limit	
EMPLOYEES	employee_id	employee_id, first_name, last_name, email, phone, hire_date, manager_id, job_title	manager_id
INVENTORIES	product_id, warehouse_id	product_id, warehouse_id, quantity	product_id, warehouse_id
LOCATIONS	location_id	location_id, address, postal_code, city, state, country_id	country_id
ORDER_ITEMS	order_id, item_id	order_id, item_id, product_id, quantity, unit_price	order_id, item_id, product_id
ORDERS	order_id	order_id, customer_id, status, salesman_id, order_date	customer_id, salesman_id
PRODCUTS	product_id	product_id, product_name, description, standard_cost, list_price, category_id	category_id
PRODUCT_CATEGORIES	category_id	category_id, category_name	
REGIONS	region_id	region_id, region_name	
WAREHOUSES	warehouse_id	warehouse_id, warehouse_name, location_id	location_id

Figure 5.1: Database Schema

The following sections provide a detailed technical walkthrough of each layer, illustrating the Boomi processes and components employed to achieve the project's objectives.

5.1 L0 - Staging Area

The L0 layer serves as the primary ingestion gateway for the ETL framework, designed to extract data from the source Oracle environment and prepare it for downstream processing while preserving the original data structure.

This layer is functionally partitioned into two distinct stages: **STG (Staging)** and **DLT (Delta)**. A modular architecture, leveraging nested subprocesses via Process Calls, was implemented to decouple these stages, thereby enhancing structural stability and simplifying long-term maintenance.

It is important to emphasize that the choice of an **STG + DLT** approach, rather than a native Change Data Capture (CDC) mechanism [9], was dictated by specific source-system constraints. Although enterprise best practices generally employ CDC to optimize data transfer, the source environment in this use case lacks the necessary technical metadata (such as reliable last-update timestamps or incremental identifiers) required for automated delta extraction. Consequently, the STG + DLT pattern was adopted as a robust alternative to simulate delta detection within the integration platform, ensuring data consistency despite the absence of source-side tracking fields.

5.1.1 STG

The STG (Staging) Step is responsible for the physical extraction of records from the source database and their ingestion into the staging area. Using the **Oracle Database Connector** with the local Atom runtime, the process executes optimized SQL queries to retrieve the required datasets from the client's on-premise environment.

The data retrieval phase utilizes a *"Standard Get"* operation, a choice dictated by its efficiency in executing direct, single-entity queries against the relational schema. A key advantage of this approach is the use of Boomi's **Import Operation** wizard. As shown in Figures [5.3, 5.4], this feature allows the user to directly select a valid

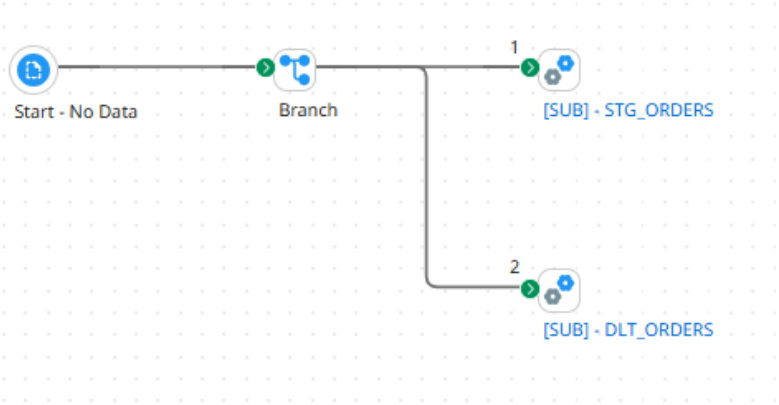


Figure 5.2: Implementation of the L0 layer in the Boomi Process Canvas

Connection component, which triggers a real-time metadata inquiry against the target database. This enables the automatic retrieval of table structures and data types, eliminating the need for manual schema definition and reducing configuration errors.

By automatically inferring the target table from the database schema, the wizard generates both the Request and Response Profiles, eliminating the need for manual XML or JSON definitions. This automated mapping of database metadata significantly accelerates the design phase, minimizing human error and ensuring that Boomi data types remain perfectly synchronized with the underlying Oracle architecture.

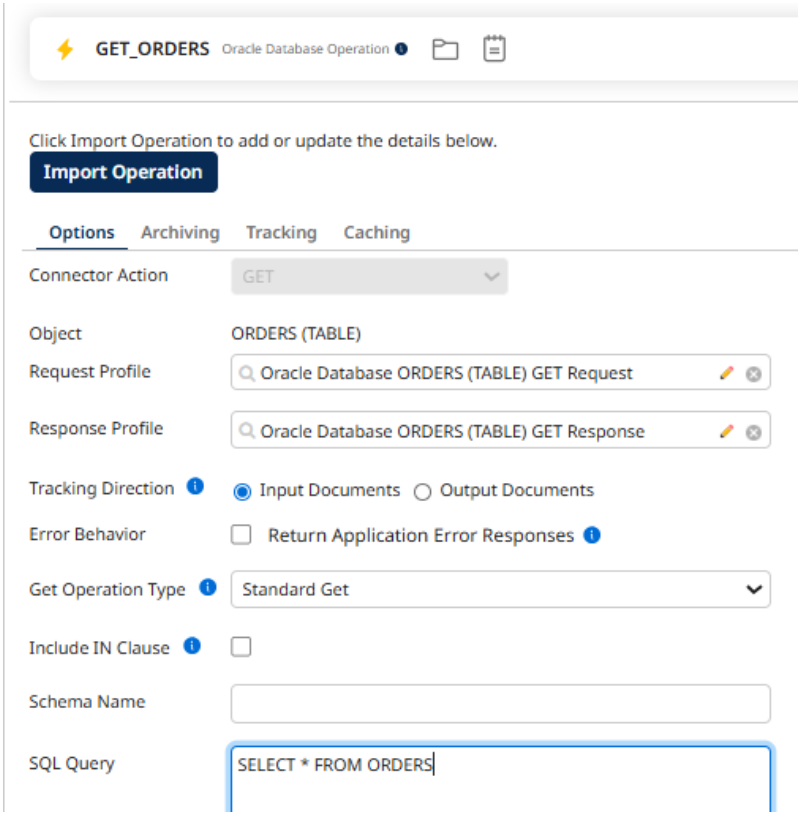


Figure 5.3: Oracle Database Connector configuration to acquire data from the source tables and inject it into the Boomi Process

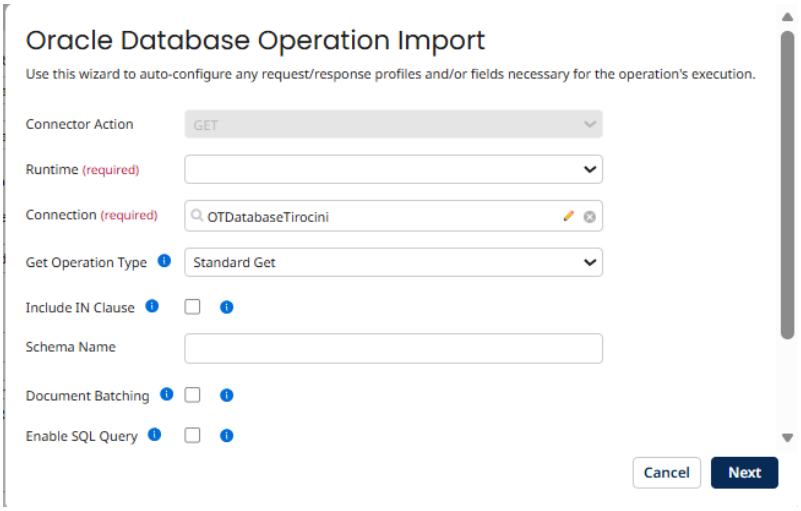


Figure 5.4: Example of the Import Operation wizard and the required fields

As illustrated below, in Figure [5.5], the workflow follows a linear and efficient logic. The raw data acquired from the source undergoes a primary transformation phase aimed at technical enrichment, where two fundamental metadata fields are appended to each record to ensure traceability and temporal alignment within the ETL ecosystem:

- **JOBID:** this technical field stores the unique execution identifier, formatted as a timestamp string (`"yyyyMMddHHmmss"`). It serves as a correlation key for all records processed during a specific run
- **INS_TIME:** this technical field captures the exact insertion timestamp in the standard SQL format (`"dd/MM/yyyy HH:mm:ss"`), providing a high-precision audit trail for data lineage

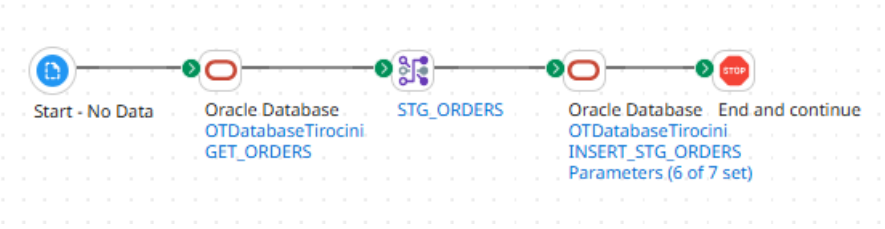


Figure 5.5: `STG` workflow

The Map Step plays a critical role in this phase as it is not only used to construct the output profile required by the `STG` table but also to perform data type validation. To ensure that date fields strictly adhere to the Oracle Database expected syntax, a Custom Map Function is invoked to reformat Boomi's internal date-time representation into the canonical string required by the target schema (`"dd-MON-yy"`).

Finally, the Oracle Database Connector is invoked to persist the enriched dataset into the `STG` table.

The underlying `INSERT` statement is implemented as a fully parameterized query. These parameters are dynamically mapped within the connector's *"Parameters"*

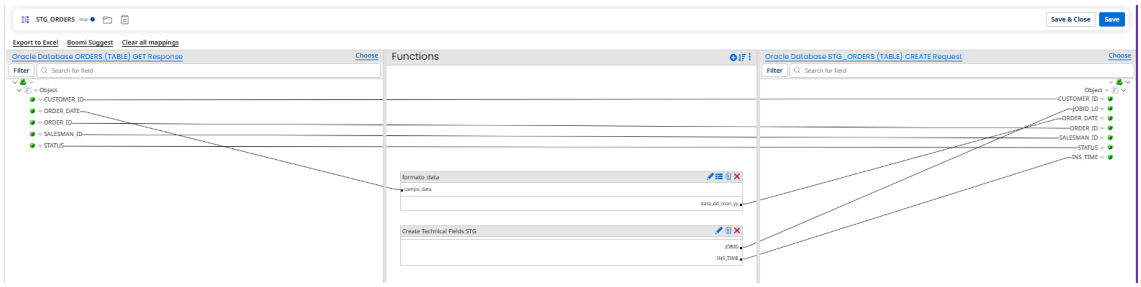


Figure 5.6: Configuration of the Map Step for the `STG` Stage

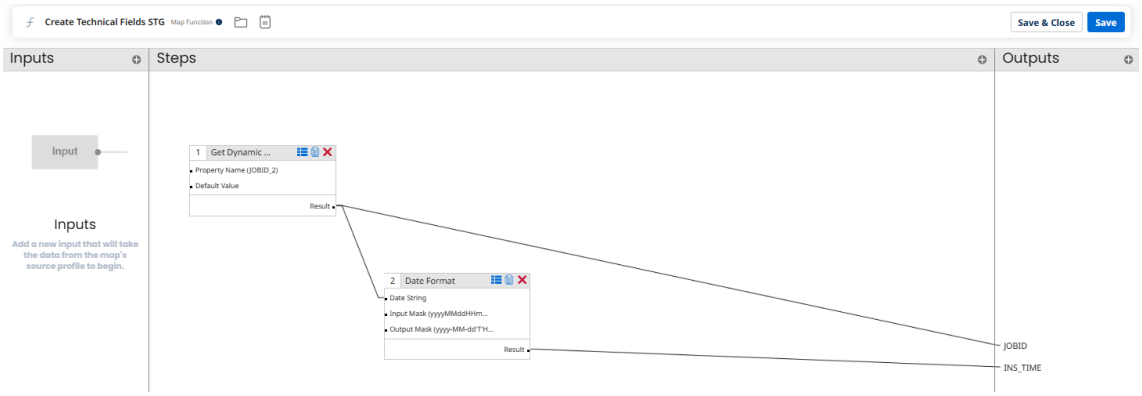


Figure 5.7: Logic of the custom function

tab, retrieving values directly from the output profile of the preceding Map Step. This architectural choice is fundamental since it prevents SQL injection vulnerabilities by separating the query logic from the data, and it maintains strict type safety, ensuring that each field adheres to the Oracle-defined data types.

```
INSERT INTO STG_ORDERS(ORDER_ID,CUSTOMER_ID,STATUS,SALESMAN_ID,
    ORDER_DATE,JOBID, INS_TIME)
VALUES (?, ?, ?, ?, ?, ?, ?)
```

However, a critical implementation detail concerns the positional nature of the parameters. Since SQL placeholders are defined using the `"?"` syntax, special attention must be paid to the ordinal mapping within the connector. The parameter order in the Boomi configuration must strictly match the column sequence defined in the SQL statement to ensure that data is routed to the correct database fields.

5.1.2 DLT

Once the ingestion into the `STG` table is complete, the framework must identify the "Delta", namely the subset of records that effectively changed or have been created since the previous execution. To achieve this, a `MINUS` logic is implemented to compare the current dataset against the previous run.

In a standard relational environment, a `MINUS` operation would be executed directly via a SQL query; however, the Boomi Oracle Connector is optimized for structured data retrieval rather than executing high-complexity relational operations like nested Common Table Expressions (CTEs) or heavy subqueries, which could impact performance and resource consumption.
Consequently, at first the logic was decoupled: the process retrieves the datasets for

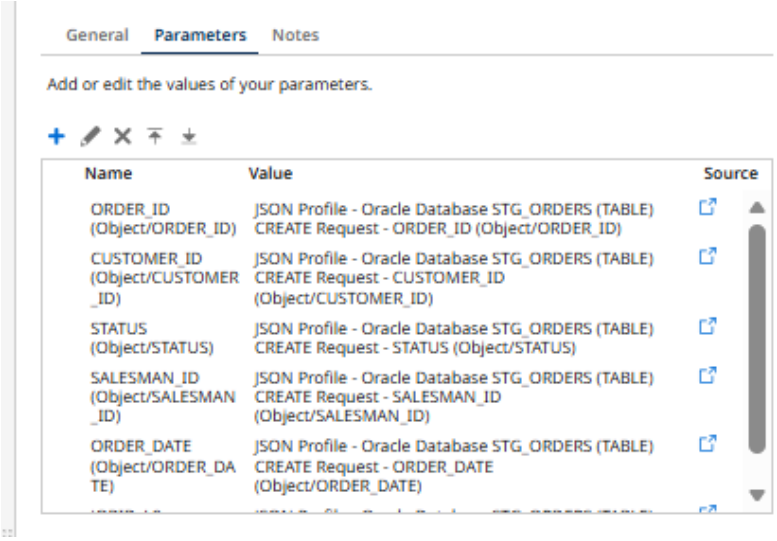


Figure 5.8: Parameters to insert into the parametric `INSERT` operation of the Oracle Database Connector

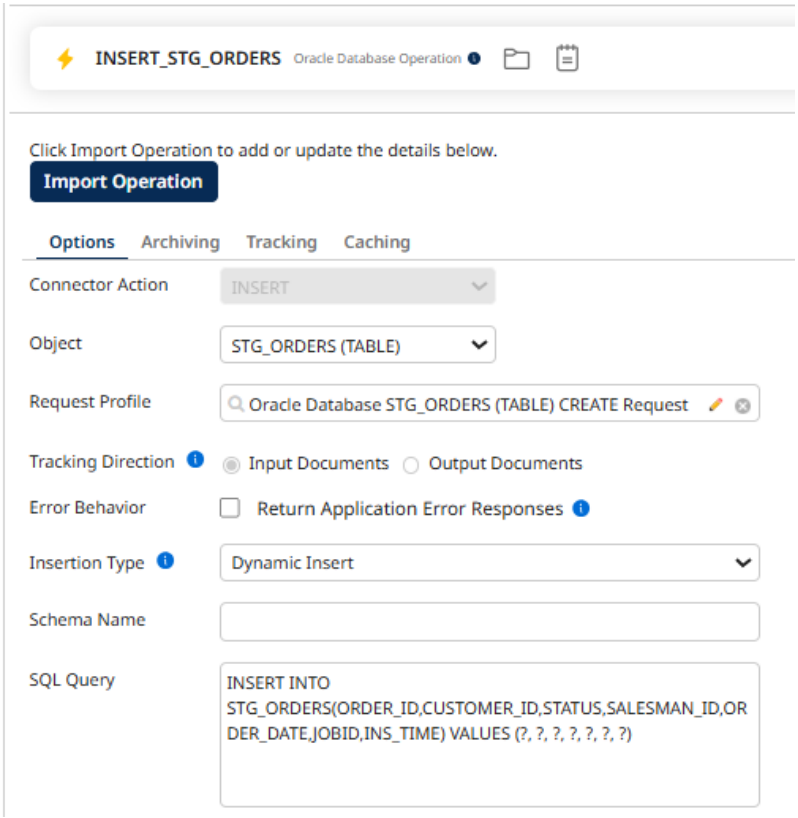


Figure 5.9: Connector Call to execute the `INSERT` operation on the `STG` table

both the current and the previous runs separately, using the `JOBID` technical field as the primary filter; one set corresponds to the current execution timestamp, while the other corresponds to the timestamp of the last successful execution.

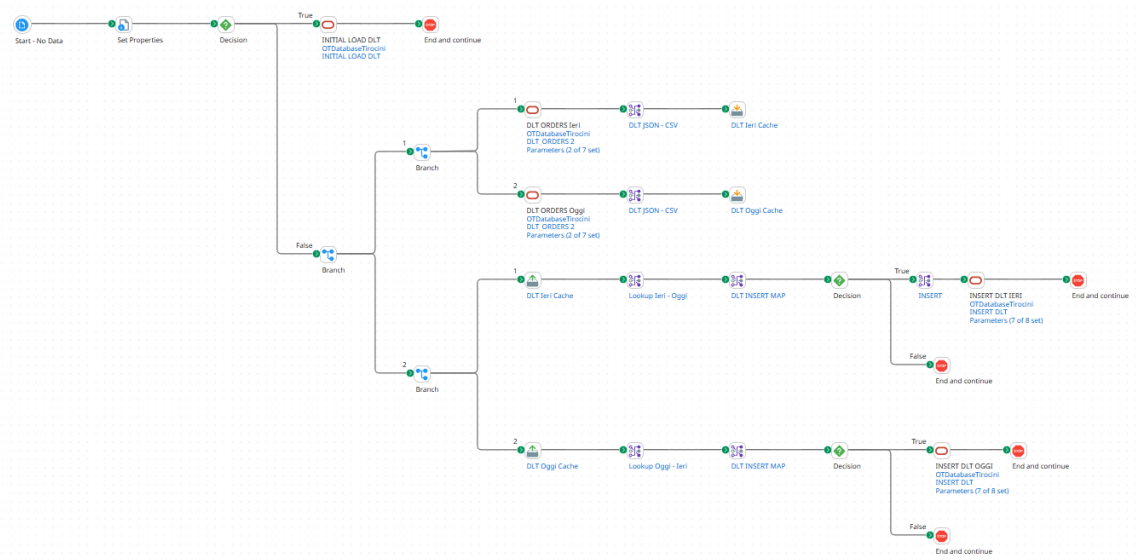


Figure 5.10: First DLT workflow implementation for the L0 stage

The FLG_NEG logic

A fundamental component of the L0 layer is the introduction of the FLG_NEG technical attribute. While the STG phase focuses on raw ingestion, the DLT phase uses this flag to categorize the lifecycle state of each record identified during delta detection.

The FLG_NEG serves as a logical indicator for downstream layers:

- **FLG_NEG = 0:** indicates a new insertion or an update to an existing record, signaling that the data must be promoted through the validation pipeline
- **FLG_NEG = 1:** signifies a logical deletion, indicating that the record is no longer present in the source dataset and should be handled accordingly in the target ODS

By assigning this flag prior to the validation stage, the framework ensures that the subsequent layers (L1 and L2) can apply specialized business logic based on the operational nature of the change.

Initial Load

A critical consideration in the design of the DLT layer is the management of the **Initial Load**. During the first execution, a "Last Execution Date" does not exist; Boomi typically initializes this system property to a default baseline date (e.g., "1970-01-01"). In this scenario, extracting a delta is unnecessary and computationally redundant, as all records are inherently new.

As illustrated in Figure [5.11], a Decision Step is implemented as a gateway: it evaluates whether the process is running in "Initial Load" mode. If no valid previous execution date is detected, the workflow bypasses the delta computation and promotes all records directly to the DLT table.

```
INSERT INTO DLT_ORDERS(ORDER_ID,CUSTOMER_ID,STATUS,SALESMAN_ID,
    ORDER_DATE,JOBID_L0,INS_TIME,FLG_NEG)
SELECT ORDER_ID,CUSTOMER_ID,STATUS,SALESMAN_ID,ORDER_DATE,JOBID_L0,
     INS_TIME, 0 AS FLG_NEG
```

```
FROM STG_ORDERS
```

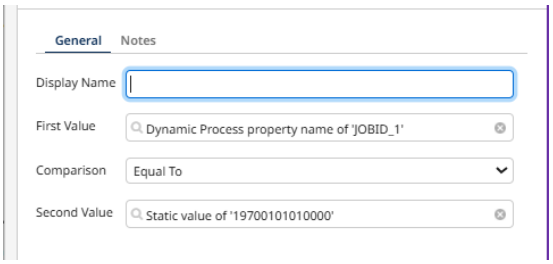


Figure 5.11: Decision step to determine whether the current execution is the initial run

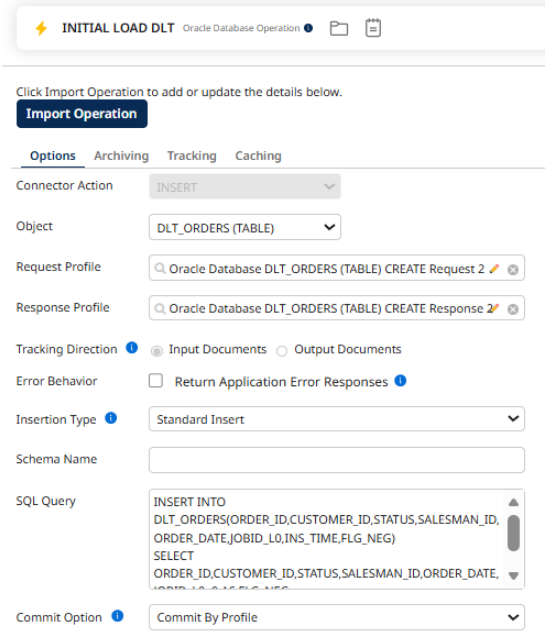


Figure 5.12: `DLT` computation for the initial load

MINUS Operation via Document Caching

For all subsequent executions, the `MINUS` operation is orchestrated within the Boomi runtime using an internal memory management strategy. This approach allows the framework to perform high-speed data comparisons without overstretching the source database's computational resources. The logic is structured into three phases:

1. **Orchestration of Data Storage:** records from the previous successful execution and the current execution are retrieved from the `STG` table and loaded into a *"Yesterday"* and *"Today"* **Document Cache**, respectively. To enable precise identification, each record is indexed by a unique composite key, generated by the concatenation of all functional attributes, intentionally excluding technical metadata (e.g., `JOBID` or `INS_TIME`) to ensure the comparison focuses exclusively on business-relevant changes. The initial requirement for two caches comes from the need to perform a side-by-side comparison of different execution windows. Specifically, isolating the data from the previous run into its own cache component is a prerequisite for the Map-based lookup logic, which identifies the Delta by cross-referencing the current flow against the historical snapshot

2. **Comparison Mapping:** the records from these executions are processed through a Map Step, where a **Document Cache Lookup** is performed using the composite key to verify the presence and state of each record in the referenced cache

3. **Differential Filtering:** by evaluating the results of the lookup, the process isolates *"Delta"* records, specifically, those present in the current run or the

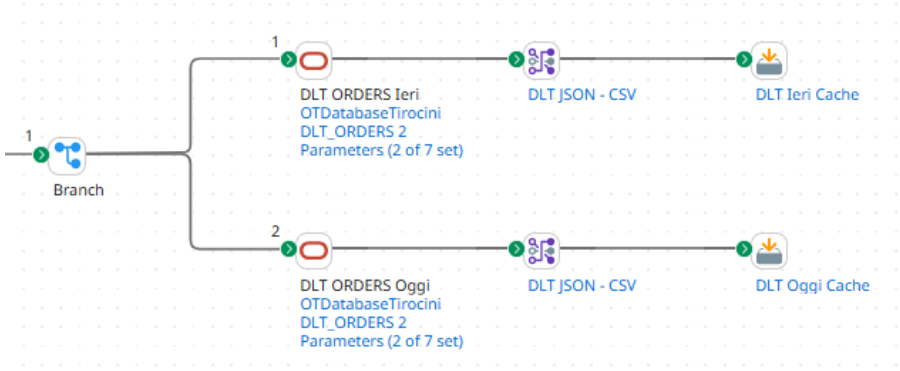


Figure 5.13: Close-up of the DLT workflow: definition and population of the caches

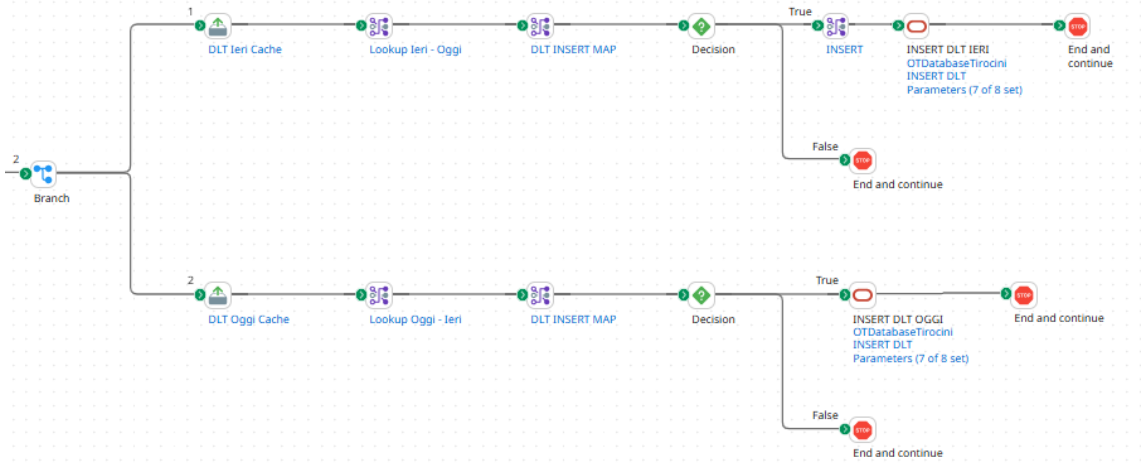


Figure 5.14: Close-up of the DLT workflow: map steps to perform the bidirectional lookup

previous run, and that are either absent from the cache or contain different attribute values. These records are then selected for insertion in the DLT table.

This caching strategy ensures that the delta computation is performed within the Boomi runtime, providing a scalable solution that maintains the integrity of the incremental load process.

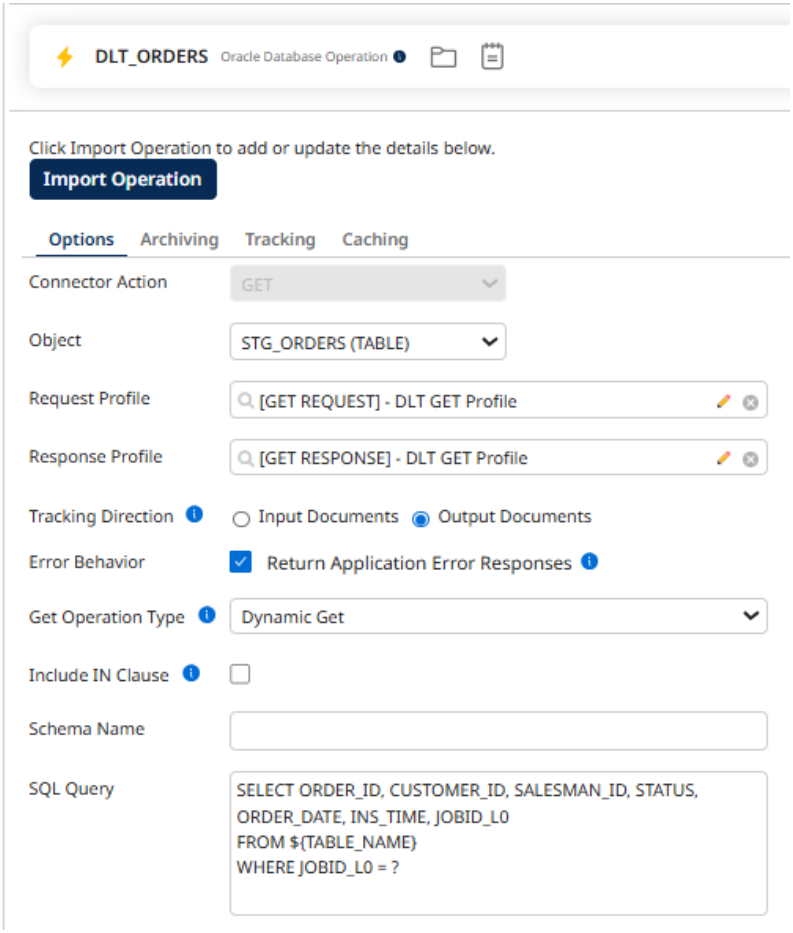


Figure 5.15: Oracle Database GET Operation to acquire data from the `DLT` table, both for the current execution and the last successful execution

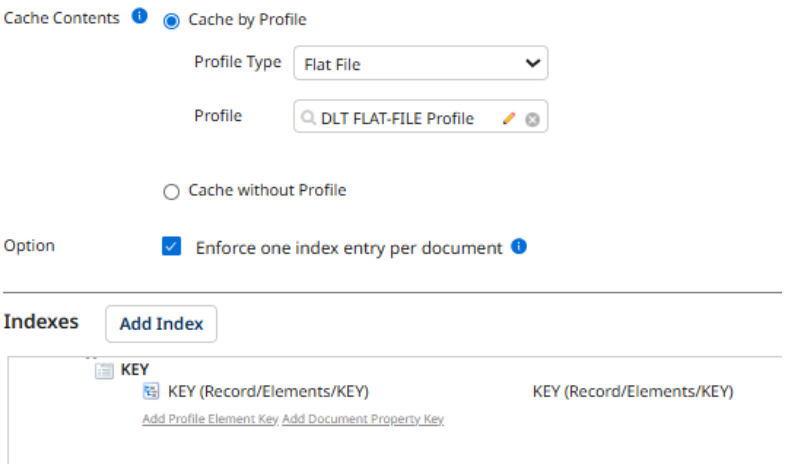


Figure 5.16: Creation of the Cache component that will hold both the new and the previous records

As mentioned beforehand, after the population of the respective Document Cache components, a bidirectional lookup strategy is employed to categorize each record. This phase is critical for distinguishing between unchanged data, new insertions, updates, and logical deletions. The Map Step serves as the orchestration engine

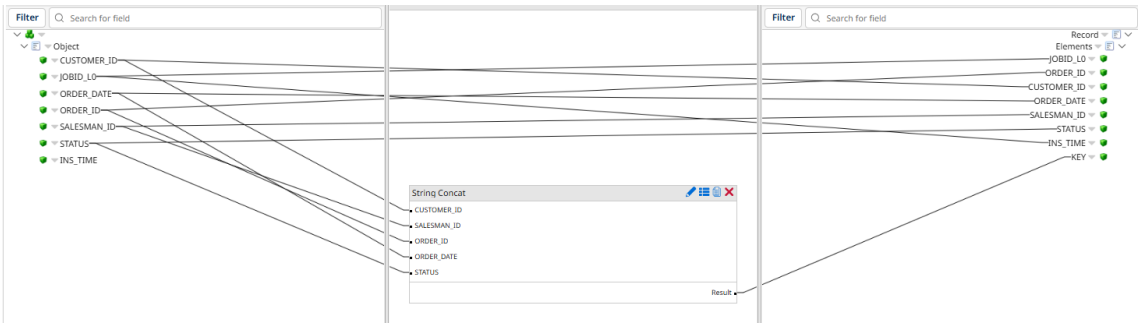


Figure 5.17: Mapping of source data into the cache profile and creation of the `KEY` field

for this logic, executing lookups against the *"Historical"* (Yesterday) and *"Current"* (Today) caches based on the previously defined composite key.

The implementation follows a specific conditional logic to determine the data's lifecycle state:

- **Hit in "Yesterday" Cache:** if a record from the current execution returns a successful match in the historical cache, it indicates that all functional attributes remain identical. Consequently, the record is discarded from the `MINUS` operation
- **Hit in "Today" Cache:** if a historical record is matched within the current cache, it confirms the record's persistence in the source system without modification, leading to its exclusion from the delta
- **Miss in "Yesterday" Cache:** a lookup failure for a current record indicates either a new insertion or an update of one or more functional fields. This record satisfies the `MINUS` computation and is staged for insertion in the `DLT` table with a flag `FLG_NEG = 0`
- **Miss in "Today" Cache:** if a historical record is not found in the current execution set, it signifies a logical deletion or a change in the business key attributes. This record satisfies the `MINUS` computation and is staged for insertion in the `DLT` table with a flag `FLG_NEG = 1`

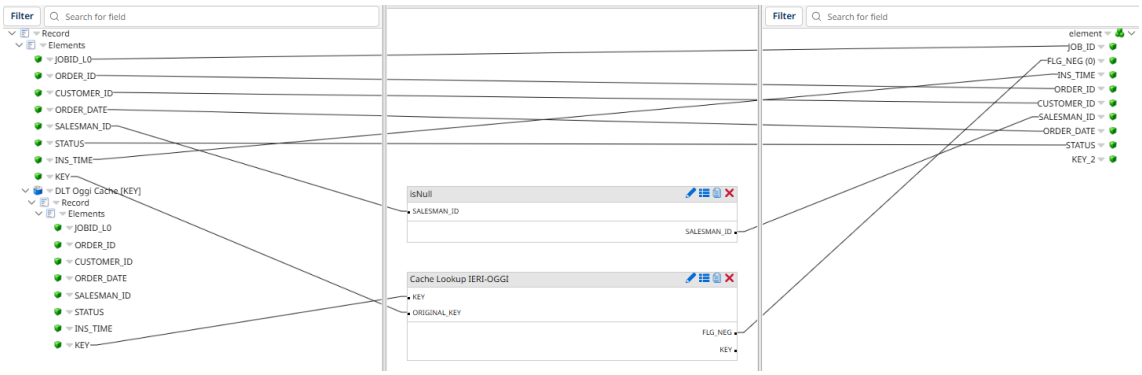


Figure 5.18: Computation of the first `MINUS` Operation, to determine which records were present in the last execution that are not in the current one

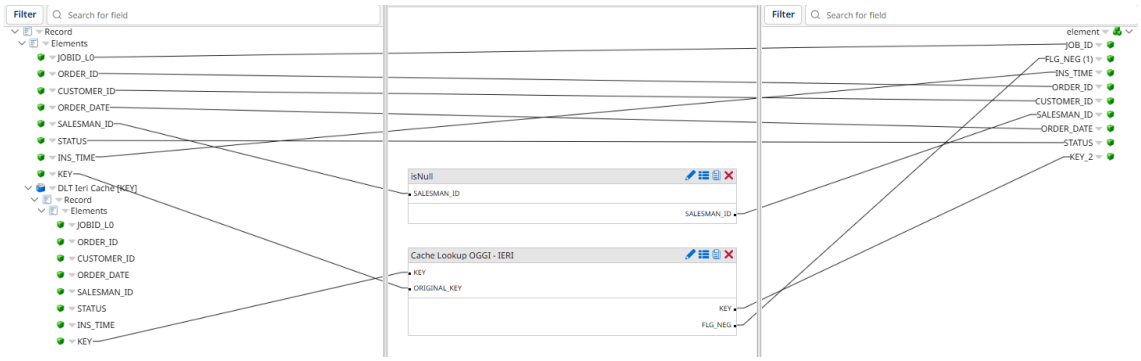


Figure 5.19: Computation of the second `MINUS` Operation, to determine which records are present in the current execution that were not in the last one

Custom Scripting for Flag Assignment

To manage the lookup failures returned by the Document Cache, custom Groovy functions are integrated within the Map Step. These scripts evaluate the presence of the business key to assign the correct operational flag.

For the lookup against the current dataset (to identify deletions), the following logic is applied:

```
if (KEY == null || KEY.trim().length() == 0) {
    FLG_NEG = 1; // Record no longer present: Logical Delete
}
```

For the lookup against the historical dataset (to identify new data):

```
if (KEY == null || KEY.trim().length() == 0) {
    FLG_NEG = 0; // Record is new: Insert/Update
}
```

Handling Nullable Fields in Boomi Profiles

A specific technical challenge identified during the testing phase involves the handling of nullable database fields, such as the `SALESMAN_ID`. By default, the Boomi engine omits elements from the output profile if the source value is *null* and this behavior causes the subsequent Oracle `INSERT` operation to fail due to a profile mismatch with the parameterized SQL statement.

To ensure structural integrity, a scripting strategy is implemented for all nullable attributes, ensuring that a literal *null* string or an empty placeholder is maintained in the profile, preventing the field from being dropped:

```
if (S_ID == null || S_ID.trim().length() == 0) {
    S_ID = 'null'; // Preserve field presence for SQL parameter
   mapping
}
```

This approach guarantees that the resulting `DLT` profile is always consistent with the database schema, regardless of the data density.

Upon successful execution of the bidirectional comparison logic and resolution of the decision paths, the identified "Delta" records are promoted to the final `DLT` table.

This operation represents the conclusion of the L0 layer, ensuring that only the refined incremental changes are prepared for the subsequent validation. For every processed record, the `JOBID` and `INS_TIME` fields are updated to reflect the current execution timestamp. This ensures that the `DLT` table provides a clear audit trail, associating each record with its specific processing cycle, maintaining the structural integrity of the data while ensuring that the operational flags (`FLG_NEG`) assigned during the Map phase are correctly persisted.

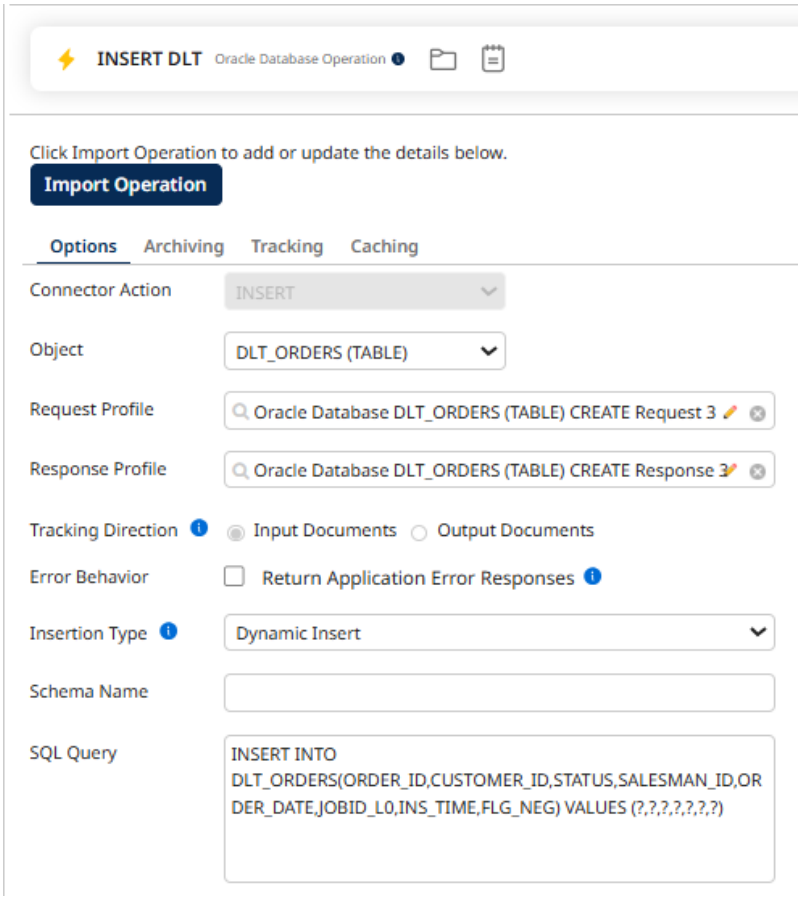


Figure 5.20: Insertion of the records into the `DLT` table

5.2 L1 - Operational Data Storage

The L1 layer represents the "Intelligence" stage of the ETL framework, dedicated to Data Cleansing, Business Rule Validation, and Exception Handling. The primary objective of this layer is to ensure that only records adhering to strict quality standards are promoted toward the final publication area, thereby preventing *"data poisoning"* in the downstream systems.

The architecture of the L1 process is designed around a multi-path orchestration (branching logic) that evaluates each record against predefined validation criteria, systematically directing the data flow based on compliance outcomes:

- **Functional Validation:** the initial evaluation phase, where business-specific rules are enforced, including mandatory field checks, date range verification,

handling duplicates, and referential integrity, to determine the document's eligibility for processing

- **Exception Management:** the subsequent routing mechanism triggered by validation failures; it isolates non-compliant records into a dedicated error table (`E$`), thereby shielding the primary integration pipeline from interruptions caused by individual data anomalies

Logical Flow and Branching Strategy

As illustrated in Figure [5.21], the L1 layer operates as a logical gateway; each record identified in the `DLT` phase is processed through an `OK` stage and later on an `ODS` stage.

The implementation of a rigorous Data Validation stage is paramount to ensure the structural and functional integrity of the processed dataset. This phase serves as a checkpoint to guarantee that data is compliant with technical requirements, as well as business rules and master data coherence. By enforcing these checks within the Boomi orchestration layer, the framework acts as a Quality Gateway, shielding the target Operational Data Storage (ODS) from non-compliant, duplicated, or corrupted information.

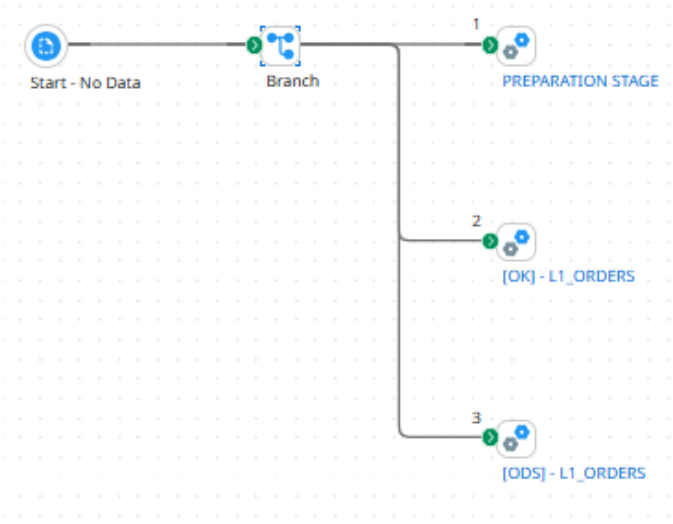


Figure 5.21: Logical flow of the L1 layer

5.2.1 OK

The OK Step represents the validation stage of the L1 layer; records that pass through this stage undergo a validation process against all technical constraints and business logic and, once validated, are prepared for synchronization with the Operational Data Storage (ODS).

As shown in Figure [5.22], the `OK` validation stage is orchestrated in two distinct phases, designed to ensure data isolation and logical consistency:

1. **Truncate Phase:** the primary branch executes a `TRUNCATE` operation on the `OK_ORDERS` table via a Program Command shape. This DDL (Data Definition

Language) operation is critical for maintaining process idempotency. By resetting the staging table at the database level before each execution, the system ensures that only the delta records from the current job run are considered candidates for consolidation. This prevents *"data bleeding"* or cross-contamination from residual records of previous integration cycles

2. **Validation and Promotion Phase:** the secondary branch initiates the data quality logic. To optimize the rejection of non-compliant data, the validations are executed in a hierarchical sequence:

   - **Referential Integrity Check:** incoming records are first validated against Master Data using Document Cache lookups to ensure all foreign key constraints are satisfied
   - **Business Rules Check:** once referential integrity is confirmed, the records are subjected to business-specific logic, such as status value verification and chronological date checks

By decoupling the environment cleanup from the validation logic through this branching architecture, the framework guarantees that the final synchronization of the `ODS` table operates exclusively on a validated, isolated, and current data set.

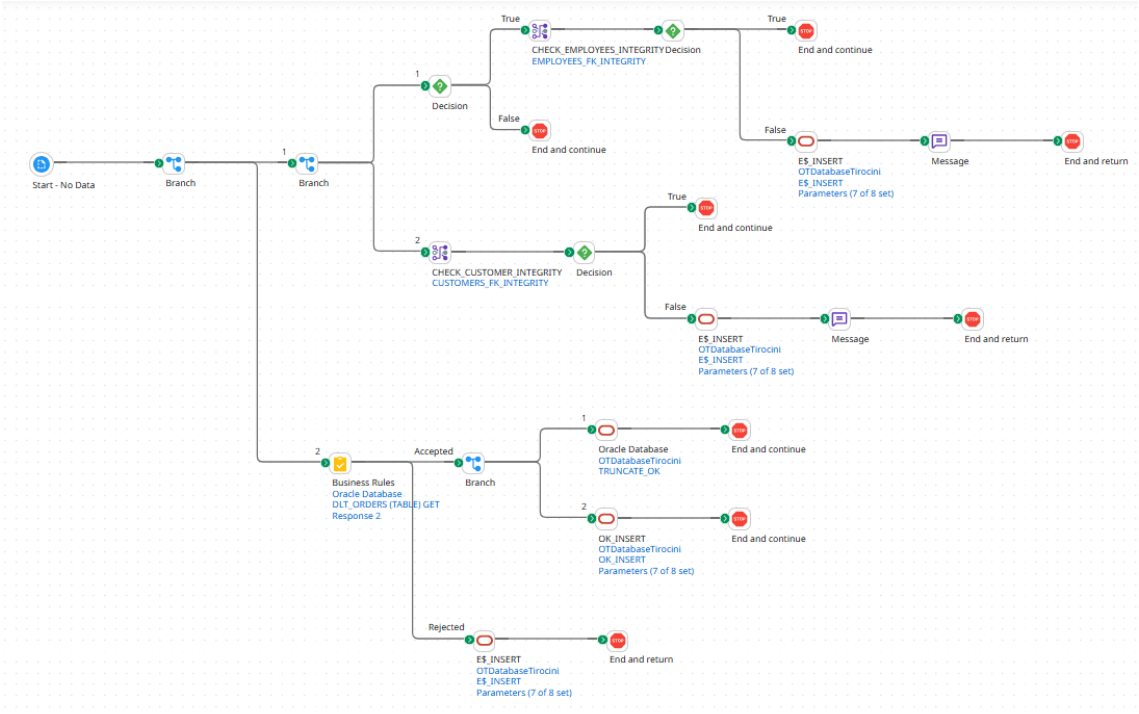


Figure 5.22: Representation of the `OK` Step and its structure

Initialization - Truncate

Before the validation logic is triggered, the workflow executes an initialization phase for the `OK_ORDERS` staging table. This operation is performed via a `TRUNCATE` statement, implemented through a Program Command Step utilizing a Legacy Database connection. Since the `OK_ORDERS` table is designed as a transient staging area (or

"Gold Zone") for the current execution's validated delta, it must be initialized to a clean state to prevent data cross-contamination or the promotion of stale records from previous cycles.

The decision to employ a `TRUNCATE` operation instead of a standard `DELETE` command is based on specific data architecture and performance requirements:

- **DDL Performance and Logging:** unlike the `DELETE` command, which is a Data Manipulation Language (DML) operation, `TRUNCATE` is a **Data Definition Language (DDL)** command. It operates by deallocating the data pages associated with the table rather than removing rows individually. This significantly reduces the overhead on the Oracle database's undo and redo logs, ensuring high-speed execution even with large datasets
- **Process Idempotency and Data Isolation:** this step is vital for ensuring process idempotency. By clearing the staging area at the beginning of each execution, the system eliminates the risk of *"data contamination"* from residual records of previous runs or partial failures. This guarantees that the `OK` table exclusively contains the validated delta of the current execution cycle

The use of the **Program Command Step** is recommended for this task, as standard Profile-based Database connectors are designed for structured DML operations and do not natively support *"fire-and-forget"* DDL commands that lack an output response profile.

```
TRUNCATE TABLE OK_ORDERS
```

Compliance with Master Data - Referential Integrity

Arguably, the most important validation within the L1 layer is the verification of referential integrity against the system's Master Data. This process is vital to ensure that the incoming transactional records correctly reference existing entities within the relational schema, preventing the creation of *"orphaned"* records that would compromise the analytical reliability of the ODS.

In this initial Boomi Integration Process, this is implemented through a series of Document Cache Lookups; specifically, for the current example, two foreign key constraints are evaluated:

- **Customer Validation:** the `CUSTOMER_ID` must correspond to a valid entry in the `CUSTOMERS` master table
- **Salesman Validation:** the `SALESMAN_ID`, when present (considering that it can assume a *null* value), must reference a valid `EMPLOYEE_ID` within the `EMPLOYEES` master table

By enforcing these constraints at the integration layer, the framework ensures that the `OK` table contains only records that are functionally consistent with the global database state. Should any of these lookups fail, the record is immediately flagged as a "Referential Integrity Violation" and routed to the `E$` error management path.

To support these lookups, a **Preparation Stage** is executed beforehand. During this phase, all Master Data involved in referential integrity constraints is fully

ingested from Oracle into dedicated Document Caches, ensuring that the validation logic is performed in-memory, significantly reducing latency compared to direct database queries.

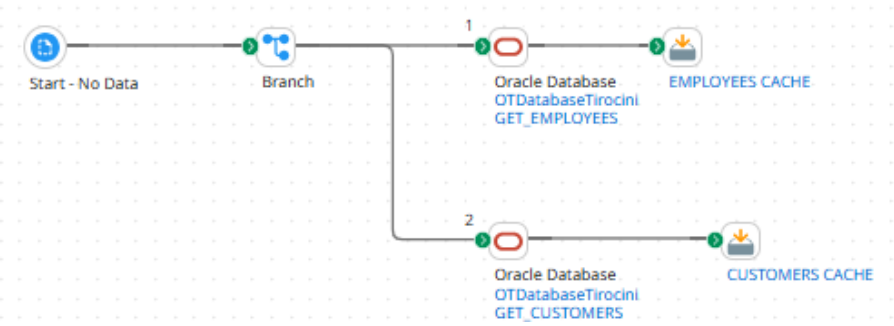


Figure 5.23: Structure of the preparation stage

For the `CUSTOMERS` validation, the Map Step invokes a lookup against the Master Data cache using the `CUSTOMER_ID` as the index. To handle potential misses, a Custom Map Function was developed to evaluate the lookup result and assign a binary status flag. This flag is subsequently evaluated by a Decision Shape positioned immediately after the Map.

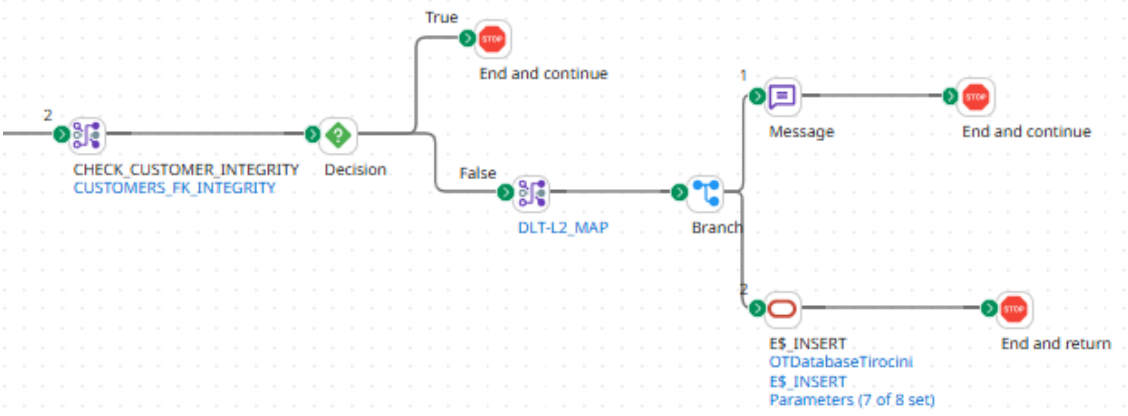


Figure 5.24: Close-up of the `OK` workflow: handling customer referential integrity

The logic for the custom function is defined as follows:

```
FLAG = 0
if(CUSTOMER_ID== null ||CUSTOMER_ID.trim().length() == 0 ){
    FLAG = 1
}
```

If the `FLAG` is set to "1", the record is diverted to the `E$` error management path. Otherwise, the document proceeds to the following validation layers.

The validation of the `SALESMAN_ID` against the `EMPLOYEES` master data requires a more sophisticated architectural approach since the field can assume a *null* value; therefore, unlike the mandatory customer reference, a missing `SALESMAN_ID` is a valid business state and should not trigger a referential integrity error.

To account for this, the logic was structured into a hierarchical check:

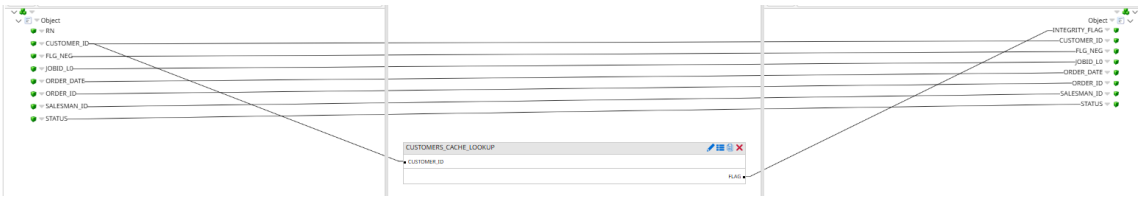


Figure 5.25: Map Step to check the presence of the field value in the `CUSTOMERS` table, through Document Cache Lookup

Figure 5.26: Custom Function, built to deal with the Cache Lookup for the `CUSTOMERS` table

- **Null Existence Check:** a Decision Shape first evaluates if the `SALESMAN_ID` is present in the document as showcased in Figure [7.8]
- **Bypass Path:** if the field is null, the referential check is bypassed entirely, and the record is promoted to the next stage, preserving the null value
- **Validation Path:** if a value is present, the process triggers a Document Cache Lookup identical to the Customer validation logic. In this case, the `SALESMAN_ID` must exist within the `EMPLOYEES` cache; a miss at this stage results in a routing to the `E$` table. The same custom function is applied to verify the presence of the record in the cache:

```
FLAG = 0
if(EMPLOYEE_ID == null ||EMPLOYEE_ID.trim().length() == 0 ){
    FLAG = 1
}
```

Error Reporting and Management

To ensure full observability, a dynamic Error Reporting mechanism was implemented to capture and categorize validation failures. Whenever a record fails a referential integrity check or a business rule, it is routed to a dedicated exception-handling branch designed to preserve document context while generating a diagnostic payload.

An important architectural consideration involves managing *"document-consuming"* shapes. As the **Oracle Database Connector** replaces the current document payload with a database response, any subsequent step in a linear sequence would lose access to the original profile elements. To prevent this data loss, the reporting logic is decoupled through a **Branch** shape:

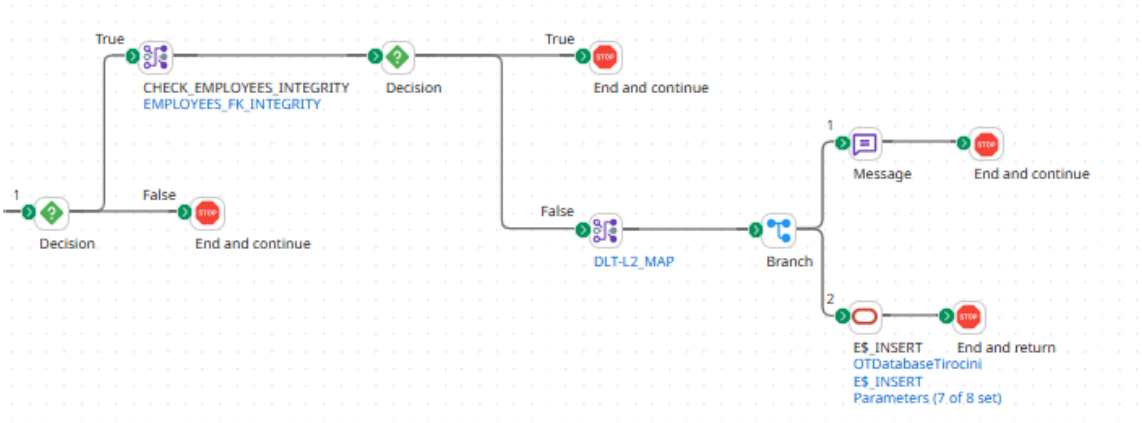


Figure 5.27: Close-up of the `OK` workflow: handling employees referential integrity

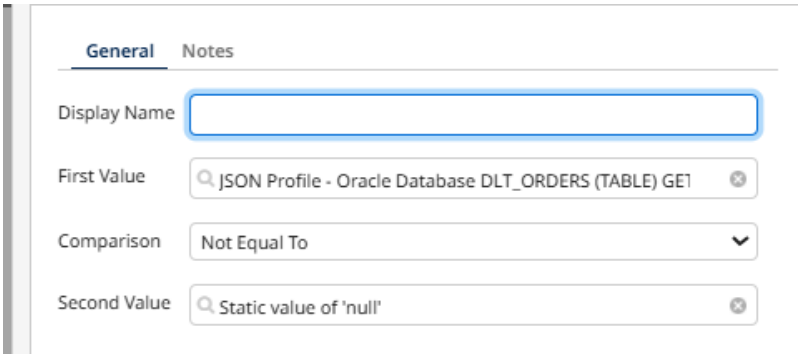


Figure 5.28: Decision Step to check the presence of the field `SALESMAN_ID`

- **Reporting Path:** the original document metadata (e.g., `ORDER_ID`, `CUSTOMER_ID`) is passed to a **Message Shape**. As shown in Figures [5.31, 5.32], this step dynamically generates a context-specific error message, distinguishing between different failure types
- **Persistence Path:** following the notification, a clone of the document is processed by the Oracle Database Connector and persisted into the `E$` staging area

Consistent with the database-centric architecture of the project, these exceptions are propagated to the Oracle environment rather than residing solely within the Boomi Atom logs. The `E$` table acts as a centralized Error Repository, where each entry includes the original record data and the associated `JOBID`; this ensures a robust audit trail, allowing data stewards to query, analyze, and remediate anomalies within a governed Oracle-based framework.

Technical Conformity - Business Rules

Once referential integrity is confirmed, the dataset undergoes a final Technical Conformity check. This stage ensures that the data is compliant with both the physical constraints of the Oracle Database and the logical requirements of the business domain. This dual validation prevents the promotion of records that, while structurally correct, are logically incoherent.

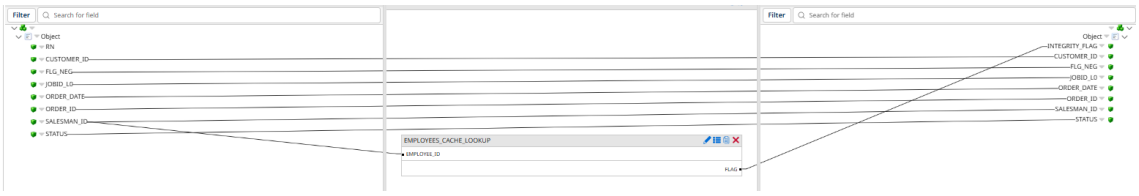


Figure 5.29: Map Step to check the presence of the field value in the `EMPLOYEES` table, through Document Cache Lookup

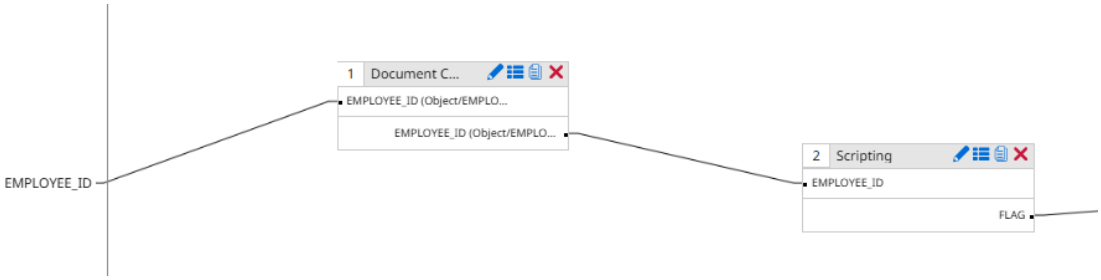


Figure 5.30: Custom Function, built to deal with the Cache Lookup for the `EMPLOYEES` table

In this implementation, each record of the `ORDERS` table is evaluated against two primary business constraints within a Business Rules Shape:

- **Status Field:** to maintain consistency, the `STATUS` field is validated against a set of allowed values: *"Canceled"*, *"Pending"*, or *"Shipped"*. Any variation or misspelling triggers an immediate reaction
- **Order Date:** the `ORDER_DATE` must satisfy two conditions:
  1. it must adhere to the canonical Oracle `DATE` format to ensure successful ingestion
  2. it must pass a logical "future-date" check, ensuring the `ORDER_DATE` is not greater than the current system date (`SYSDATE`)

Deterministic Data Promotion and Collision Management

Records that successfully navigate the structural, business, and referential integrity checks are classified as "Business-Ready," signaling their eligibility for promotion to the `OK_ORDERS` table. However, the L1 layer must also address potential record collisions within the incoming delta, scenarios often triggered by upstream reprocessing, late-arriving corrections, or overlapping ingestion cycles. To ensure that the validation layer remains free of duplicates, a deterministic selection strategy is embedded directly into the logic.

Rather than utilizing a standard `INSERT`, the implementation adopts a conditional *"Insert-if-not-exists"* pattern. This approach groups records by their natural business key (`ORDER_ID`) and ensures that a record is only persisted if it represents a more recent or equivalent state compared to what is already residing in the target table. This safeguards the system against the promotion of obsolete data during parallel processing.

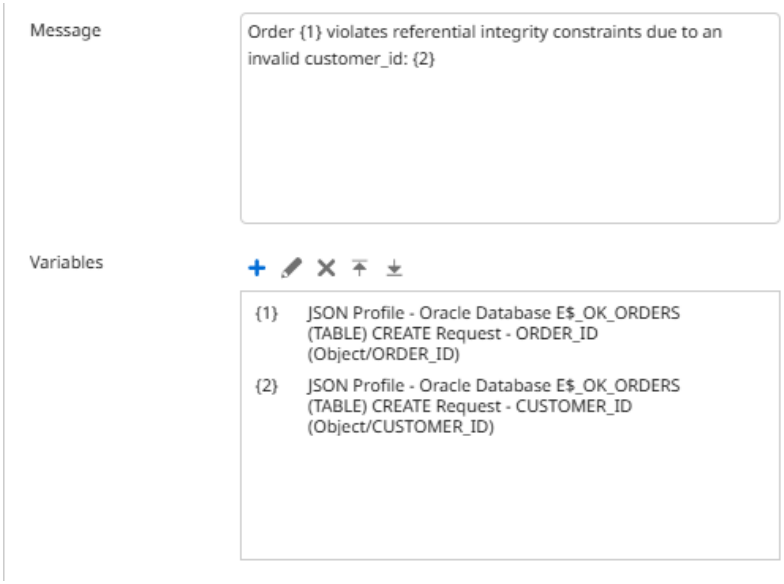


Figure 5.31: Error Message for `CUSTOMERS` table integrity failure

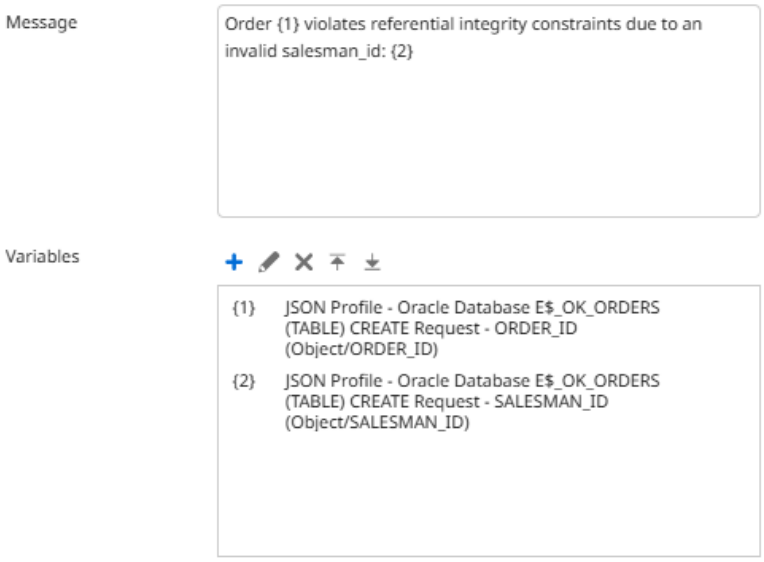


Figure 5.32: Error Message for `EMPLOYEES` table integrity failure

```
INSERT INTO OK_ORDERS (ORDER_ID, CUSTOMER_ID, STATUS, SALESMAN_ID,
   ORDER_DATE, JOBID_L1, INS_TIME, FLG_NEG)
SELECT ?, ?, ?, ?, ?, ?, ?, ?
FROM DUAL
WHERE NOT EXISTS (
    SELECT 1 FROM OK_ORDERS
    WHERE ORDER_ID = ?
    AND (
        JOBID_L1 > ?
        OR (JOBID_L1 = ? AND INS_TIME > ?)
        OR (JOBID_L1 = ? AND INS_TIME = ? AND FLG_NEG <= ?)
    )
);
```

Following this logic, the validated records are persisted through the Oracle Database

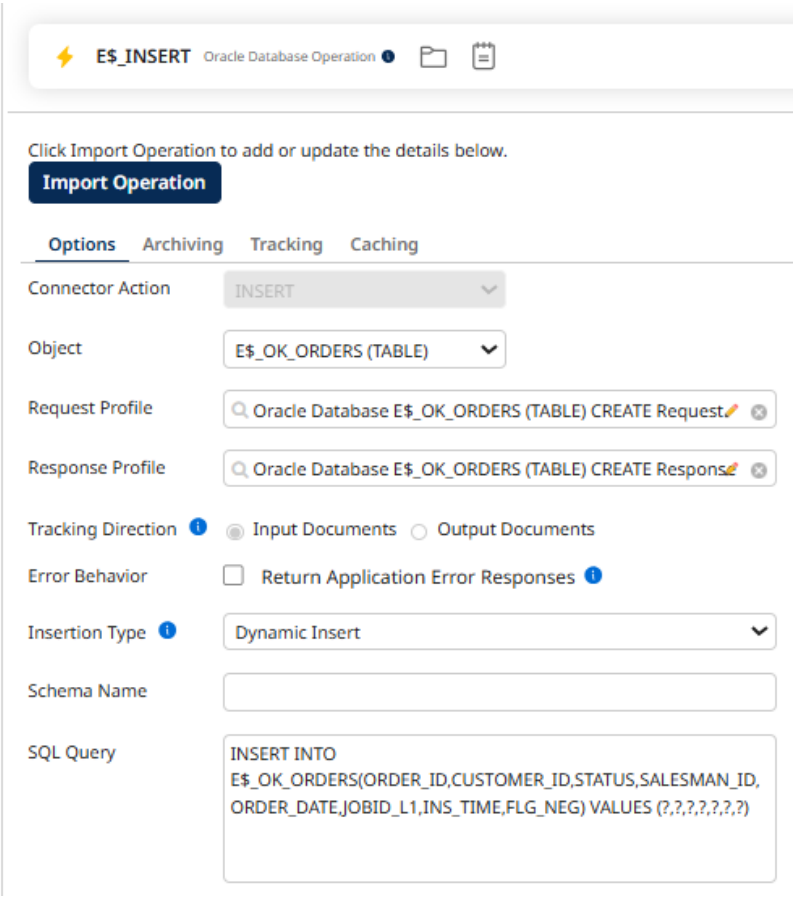


Figure 5.33: Insertion of non-compliant records into the `E$` table for audit

Connector, utilizing a fully parameterized statement. By enforcing these technical conformity checks at the point of insertion, the workflow ensures that the final Operational Data Storage (ODS) synchronization, typically performed via a `MERGE` operation, acts upon a dataset that strictly reflects the success path of the current execution, maintaining data lineage and consistency.

Data Transformation and Format Normalization

It is important to highlight that an additional **Map Step** was integrated into the workflow prior to the database inserts. This architectural choice was necessary due to the stringent data type requirements of the Oracle Database Connector, specifically, while the source data may represent dates in various string formats, the Oracle JDBC driver requires the `ORDER_DATE` field to strictly adhere to a canonical format (`"dd-MON-yy"`).

Although this additional mapping layer introduces a degree of structural redundancy and increases the complexity of the process flow, it serves as a critical **Data Normalization** stage. Without this explicit transformation, the integration would be susceptible to runtime exceptions. By centralizing the formatting logic within a dedicated Map Step, the framework ensures that all outbound documents are technically compliant with the target schema, thereby maintaining integrity and preventing data type mismatches.

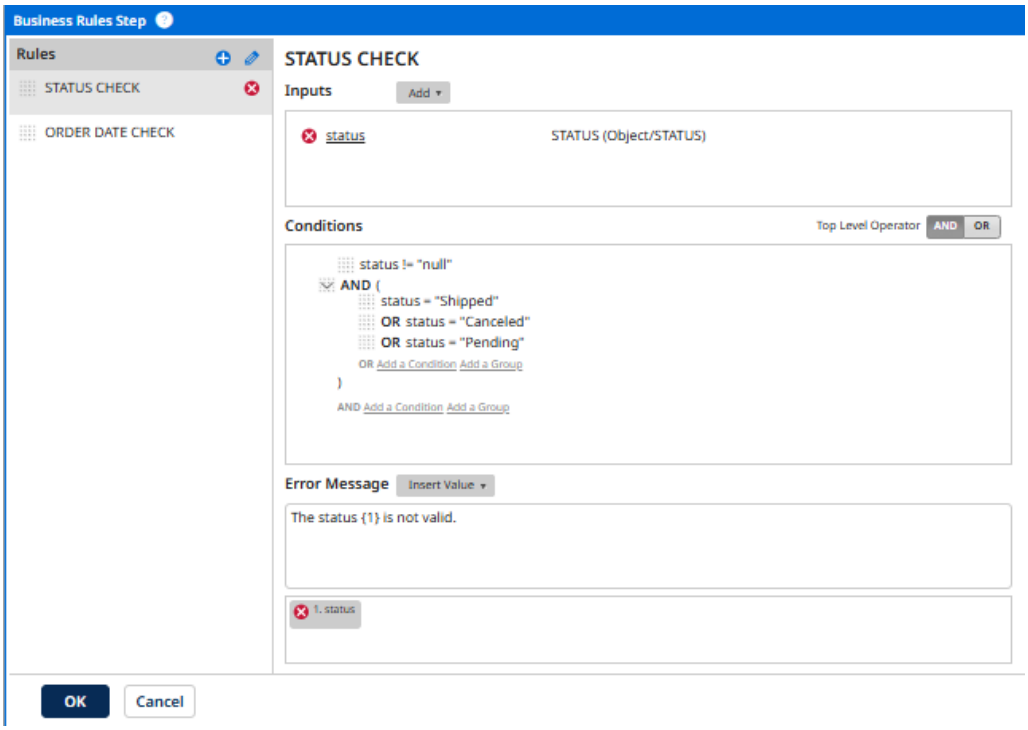


Figure 5.34: Business Rule to check the compliance of the `Status` field with respect to the expected value

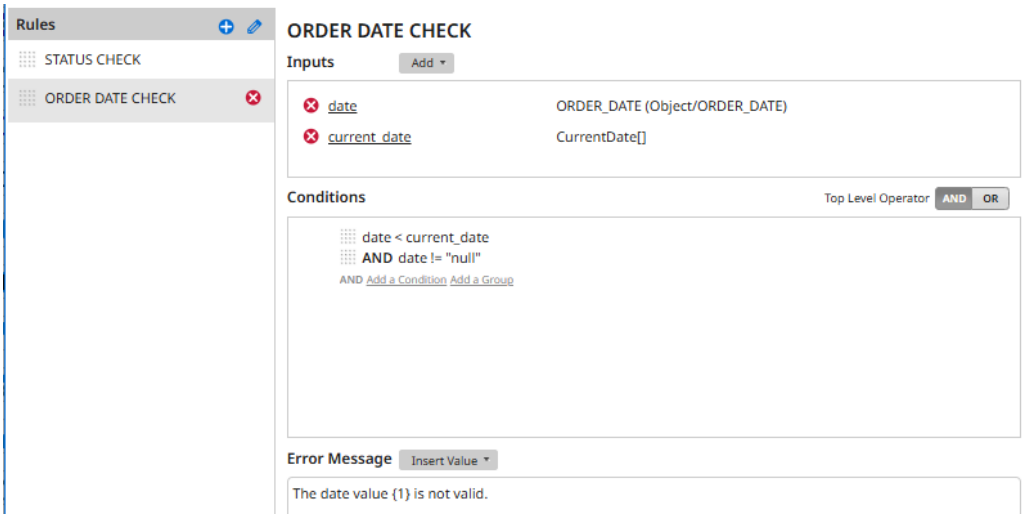


Figure 5.35: Business Rule to check the compliance of the `Order_date` field with respect to the expected value

5.2.2 ODS

To manage the promotion of data from the `OK` staging area to the `ODS`, the framework implements an Upsert (Update + Insert) logic. This strategy is essential for maintaining a synchronized state, as the incoming delta dataset could contain a mix of entirely new records, functional updates to existing entries, and logical deletions.

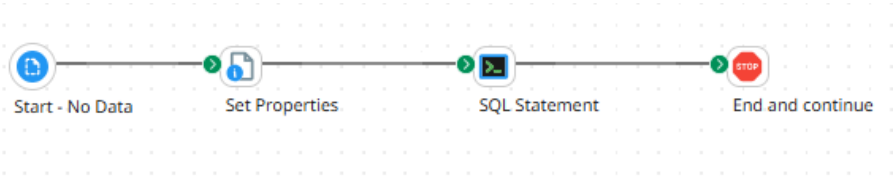


Figure 5.36: Structure of the ODS Stage

Dynamic Metadata Enrichment

Before invoking the database connector, a Set Properties Step is employed to dynamically generate the temporal metadata required for the MERGE operation.

As illustrated in Figure 5.37, the following properties are initialized:

- **Current Execution Timestamp:** captured at runtime and assigned to a UPD_TIME variable
- **Process Execution ID:** mapped to the JOBID_L1 property

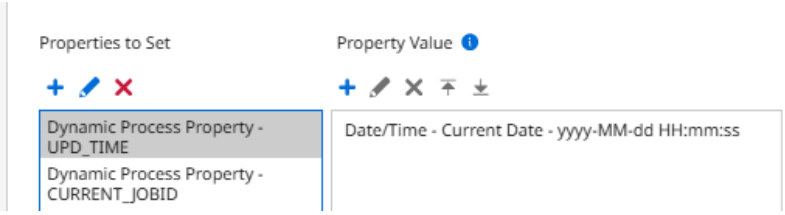


Figure 5.37: Set Properties Step to account for the new technical metadata

The Relational Merging Logic

The core of this stage lies in the conditional handling of record states. By leveraging a single atomic MERGE statement, the framework executes a dual-path logic based on the presence of the Primary Key (PK) within the target ODS:

1. **Update Path (Matched Records):** if a record with the same Primary Key already exists in the ODS, the framework performs an in-place update. In this scenario, the business attributes are refreshed, and the technical fields JOBID_UPD, FLG_NEG, and UPD_TIME are updated to reflect the current execution context. This preserves the original JOBID_INS and INS_TIME, providing a clear distinction between when a record was first created and when it was last modified
2. **Insert Path (Unmatched Records):** if no matching key is found, the record is treated as a new entry. The framework initializes both the "Insertion" and "Update" metadata symmetrically:
   - JOBID_INS and JOBID_UPD are both set to the current JOBID
   - INS_TIME and UPD_TIME are both set to the current execution date
   - FLG_NEG is persisted to define the initial state of the record

This logic is translated into the SQL query:

```
    MERGE INTO ODS_OK_ORDERS TGT
USING (
    SELECT * FROM OK_ORDERS
    WHERE JOBID_L1 = ? -- CURRENT_JOBID
) SRC
ON (TGT.ORDER_ID = SRC.ORDER_ID)
WHEN MATCHED THEN
    UPDATE SET
        TGT.CUSTOMER_ID = SRC.CUSTOMER_ID,
        TGT.STATUS      = SRC.STATUS,
        TGT.SALESMAN_ID = SRC.SALESMAN_ID,
        TGT.ORDER_DATE  = SRC.ORDER_DATE,
        TGT.JOBID_L1_UPD = SRC.JOBID_L1,
        TGT.FLG_NEG     = SRC.FLG_NEG,
        TGT.UPD_TIME    = ? -- UPD_TIME
WHEN NOT MATCHED THEN
    INSERT (
        ORDER_ID,
        CUSTOMER_ID,
        STATUS,
        SALESMAN_ID,
        ORDER_DATE,
        JOBID_L1_INS,
        INS_TIME,
        FLG_NEG,
        JOBID_L1_UPD,
        UPD_TIME
    )
    VALUES (
        SRC.ORDER_ID,
        SRC.CUSTOMER_ID,
        SRC.STATUS,
        SRC.SALESMAN_ID,
        SRC.ORDER_DATE,
        SRC.JOBID_L1,
        ?,  -- UPD_TIME
        SRC.FLG_NEG,
        SRC.JOBID_L1,
        ?   -- UPD_TIME
)
```

This distinction is vital for auditing purposes, as it allows the client to track the complete history of a record, from its inception to its most recent modification. By centralizing this logic within an SQL `MERGE` command triggered by Boomi, the system achieves a high level of performance and transactional consistency.

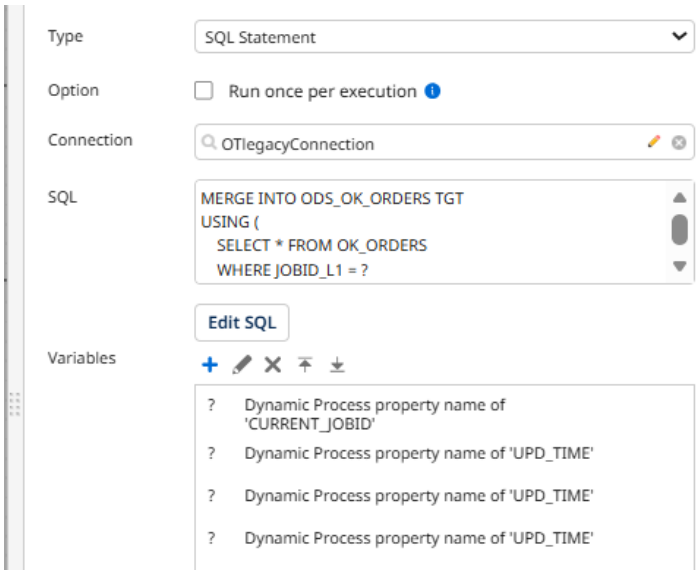


Figure 5.38: Program Command Step configuration to manage the `UPSERT` logic in the `ODS` table

5.3 Architectural Decisions and Technical Constraints

The development of the ETL framework required a series of strategic choices to balance Boomi's abstraction layers with the high-performance requirements of the Oracle Database; following is the rationale behind the connector selection and the deliberate design omissions regarding Master Data Management and Output Delivery.

5.3.1 Architectural Overview: Oracle Database Connector vs. Database (Legacy) Connector

During the implementation of the L0 and L1 layers, a deliberate distinction was made between the use of the Oracle Database Connector and the Database (Legacy) Connector, based on the nature and execution characteristics of the SQL operations involved.

The Oracle Database Connector was adopted as the standard mechanism for Data Manipulation Language (DML) operations, such as the `INSERT` and `SELECT` statements executed during the `STG` and `DLT` Stages. This connector operates on a profile-based abstraction layer, leveraging Boomi's mapping engine to translate structured JSON elements into positional JDBC placeholders (*"?"*). While this approach guarantees strong metadata alignment and design-time validation, it also introduces a degree of structural rigidity; as SQL complexity increases, this rigidity can become a limiting factor, particularly when dealing with advanced DML constructs or non-relational execution patterns.

This limitation is more evident in the context of Data Definition Language (DDL) operations. For example, the Truncate phase within the L1 layer relies on the

TRUNCATE statement, which is a DDL command that deallocates data pages and does not return a conventional result set. Unlike DML statements, DDL operations do not conform to a predictable response structure; therefore, the Oracle Database Connector, which expects a response profile for execution validation, is not well-suited for such administrative commands.

To address this constraint, the Program Command shape, backed by the Database (Legacy) Connector, was used to bypass the profile-validation layer entirely, allowing raw SQL statements to be passed directly to the underlying JDBC driver. This approach provides greater execution flexibility while maintaining transactional correctness at the database level.

The same architectural rationale guided the implementation of the ODS synchronization logic; although the MERGE statement is technically classified as a DML operation, its atomic upsert semantics, combining MATCHED and NOT MATCHED execution paths, introduce significant complexity when modeled within a single Boomi profile. Attempting to represent this dual-path logic through standard mapping would substantially increase process complexity and reduce maintainability.

By executing the MERGE operation through the Legacy Connector, the framework intentionally shifts from a traditional ETL (Extract-Transform-Load) pattern to an ELT (Extract-Load-Transform) paradigm [8]. This design choice offloads relational joins, conditional branching, and data reconciliation logic to the Oracle Database Engine, allowing its native optimizer to manage execution plans and transactional integrity. As a result, the Boomi Atom is relieved from performing resource-intensive, record-by-record comparisons in JVM memory, preventing it from becoming a computational bottleneck and ensuring better scalability and performance under high data volumes.

5.3.2 MDM and OUT Strategy

Beyond the active orchestration of the L0 and L1 layers, two critical architectural decisions were made regarding the Master Data Management (MDM) strategy and the handling of outbound data (OUT). These choices reflect the client's current architecture, where data comes from a single source and is persisted to Oracle at the end of each stage, while still allowing for future enterprise-scale growth.

Master Data Management (MDM) and Future Scalability

In a traditional integration landscape, a Master Data Management (MDM) Stage is utilized to reconcile, de-duplicate, and synchronize records across heterogeneous source systems. However, within the current scope of this project, a dedicated MDM layer was deemed unnecessary as the architectural ecosystem is entirely centralized within the **Oracle Database** environment. Since the referential integrity and master data consistency are natively managed at the relational level, implementing an external MDM solution at this stage would introduce redundant complexity without immediate functional benefit.

Should future business requirements evolve, the framework is designed to facilitate a seamless transition toward **Boomi DataHub** [7]. This strategic shift would be particularly relevant if the integration landscape expands to include disparate

Business Units or external platforms such as e-commerce and third-party CRM systems. Adopting a DataHub-centric model would enable centralized governance and the enforcement of data quality standards across diverse, non-Oracle sources, ensuring a *"single source of truth"* as the ecosystem grows.

OUT Architecture

The strategy for the data output (`OUT`) layer has been streamlined to align with the core objective of this thesis: the definition of a robust, pure Integration Framework. In conventional enterprise ETL architectures, the `OUT` stage serves as the final refinement layer, responsible for transforming processed data into highly specialized formats for external consumption. This typically includes the generation of Surrogate Keys (SK) for dimensional modeling, the execution of complex business aggregations, or the restructuring of datasets into star schemas optimized for Data Warehousing [8].

In this implementation, given the absence of a predefined target consumer and the focus on architectural orchestration, the `OUT` phase is performed implicitly. As records are successfully validated and promoted through the functional Stages, they are persisted directly into dedicated, purpose-built Oracle tables. This ensures that the high-fidelity data is immediately available for Business Intelligence tools and downstream applications without the added latency of an auxiliary extraction layer.

Should the operational requirements evolve, the framework is designed for rapid specialization. Future iterations can easily incorporate a dedicated `OUT` stage to define specific delivery formats, precise structural mappings, or the SK logic required for advanced Data Warehouse construction, adapting the final output to the exact specifications of the end-users.

Chapter 6

Enhanced Error Management

In the initial architectural iteration, records that failed to satisfy data integrity constraints were routed directly to a static discard table (`E$`), where they remained pending manual assessment. This approach treated such records as terminal failures, introducing significant operational overhead and limiting the pipeline's ability to recover autonomously from transient data inconsistencies.

The improved implementation addresses this limitation by introducing a **Recycle Error** mechanism, fully aligned with Boomi's modular process design. Under this paradigm, error records are treated as *"temporarily non-compliant"*: they are persisted in an intermediate state and automatically re-evaluated in subsequent execution cycles, enabling a self-healing data flow and reducing manual intervention.

A frequent challenge in ETL workflows is the temporary violation of referential integrity; for instance, a record may be ingested before its corresponding master record is propagated from the source system. To mitigate this, the implemented recycling logic performs a re-injection before the next processing stage: at the start of each new execution, records stored in the `E$` table are fetched and fed back into the Boomi validation engine.

The logic follows a dual path based on the updated system state:

- **Successful Reconciliation:** if the missing dependency has been synchronized in the source, the recycled record passes validation, is moved to the `OK` layer, and promoted to the `ODS` following the standard promotion logic
- **Deferred Persistence:** if the integrity violation persists, the record is kept in the `E$` table. This ensures that the process flow continues without interruption while maintaining a complete audit trail of pending records

6.1 Automated Client Notification

To enhance operational transparency, an automated notification mechanism was integrated into the workflow, ensuring that the client is actively alerted of any data quality issues that require manual oversight or source-system correction.

6.1.1 Mail Connector

The **Mail Connector** is employed as an outbound communication interface, triggered after a failure of the Business Rules validation or referential integrity checks. Rather than batching notifications into a single generic alert, the connector is configured to handle the document flow dynamically, ensuring that the client receives timely information regarding records being moved to the `E$` table.

Key technical configurations include:

- **Connection:** configured with the client's SMTP server protocols and secure authentication (TLS/SSL)
- **Operation:** defined as a Send action, where the "From", "To", and "Subject" fields are often parameterized to allow for environment-specific routing

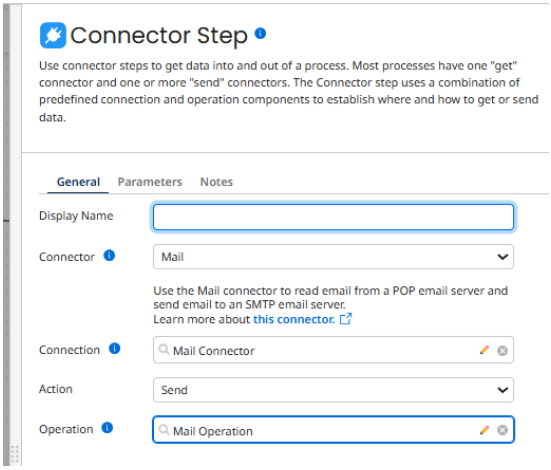


Figure 6.1: Base configuration of the Mail Connector

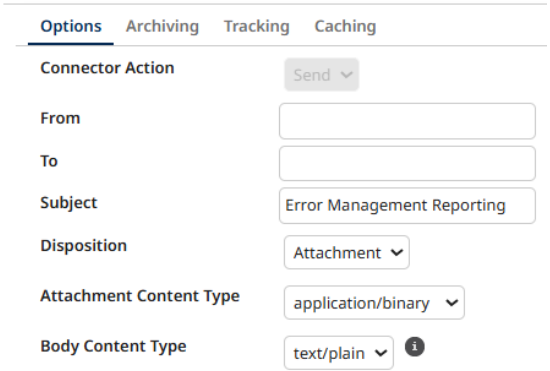


Figure 6.2: Configuration of the Mail operation, redacted for privacy purposes

6.1.2 Message Step

By leveraging Boomi variables and placeholders within the Message Step, the system dynamically generates readable diagnostic reports that specify the cause of failure and converts them into the email attachment body. This allows the client to immediately understand the cause (e.g., a specific missing foreign key) without needing to access the **Boomi Process Reporting** console.

This decoupling of technical logs from business notifications significantly improves the efficiency of the data governance cycle; unlike the standard Boomi message log (Fig. [6.4]), which requires the user to manually parse execution metadata to identify the root cause, the synthesized report (Fig. [6.5]) presents only the error message in a clear and comprehensive format, making it immediately accessible to the user.

6.1.3 Set Properties Step

This shape acts as the configuration layer for the outbound communication, where specific **Document Properties** are dynamically assigned to the document flow. While the preceding Message Step generates the data payload, the Set Properties

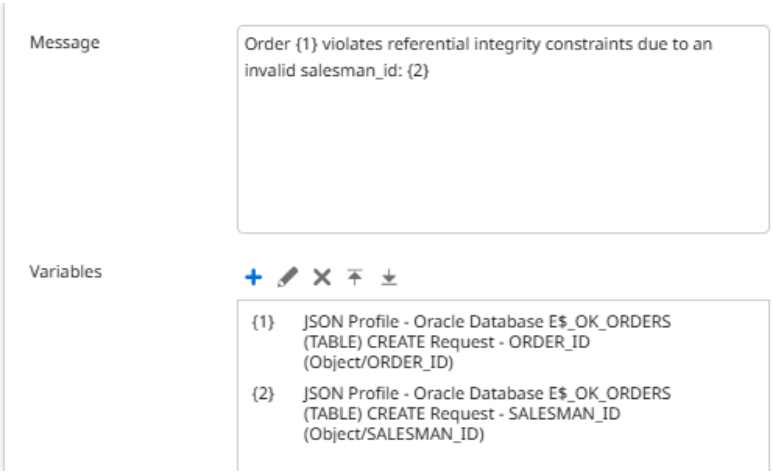


Figure 6.3: Attachment content, dynamically constructed at run time

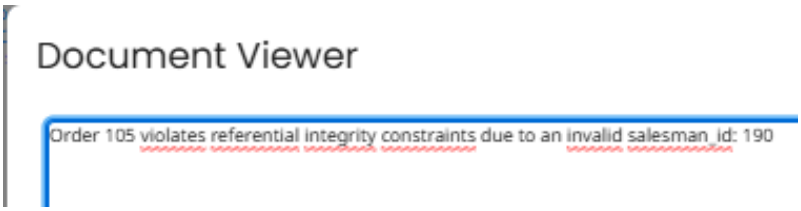


Figure 6.4: Error Message retrieved from the Process Execution Logs

```
The referential integrity check for order 105 failed. There is no salesman with id: 190.
```

Figure 6.5: Custom Email Diagnostic Report

shape orchestrates the metadata required for the delivery protocol.
Specifically, it is utilized to parameterize the *"Subject"* and *"File Name"* fields, thus making sure that the diagnostic information synthesized in the previous steps is delivered in a structured, professional, and actionable format.

6.2 Implementation of the Error Recycle Strategy

As mentioned beforehand, the L1 (`OK`) Stage was redesigned to incorporate an *"Error Recycling"* mechanism. This logic ensures that records residing in the error staging tables (`E$`) from previous executions are reconsidered alongside incoming data, particularly relevant in scenarios where a record initially failed due to missing master data or non-compliant field values that the client may have since corrected in the source system.

The primary challenge in implementing recycling logic lies in the complex retrieval of data from the `E$` table. Since it appears that the Oracle Database Connector does not provide sufficient flexibility or support advanced logic, the Database Legacy Connector was employed to execute a custom SQL query utilizing Common Table Expressions (CTE) and Window Functions, in its place.

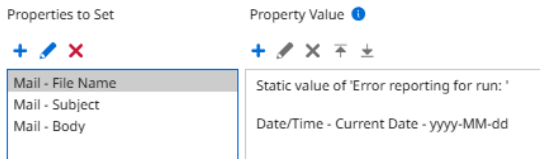


Figure 6.6: Set Properties step, dynamic construction of the file name

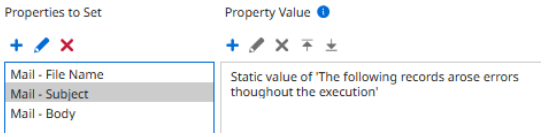


Figure 6.7: Set Properties step, dynamic construction of the subject

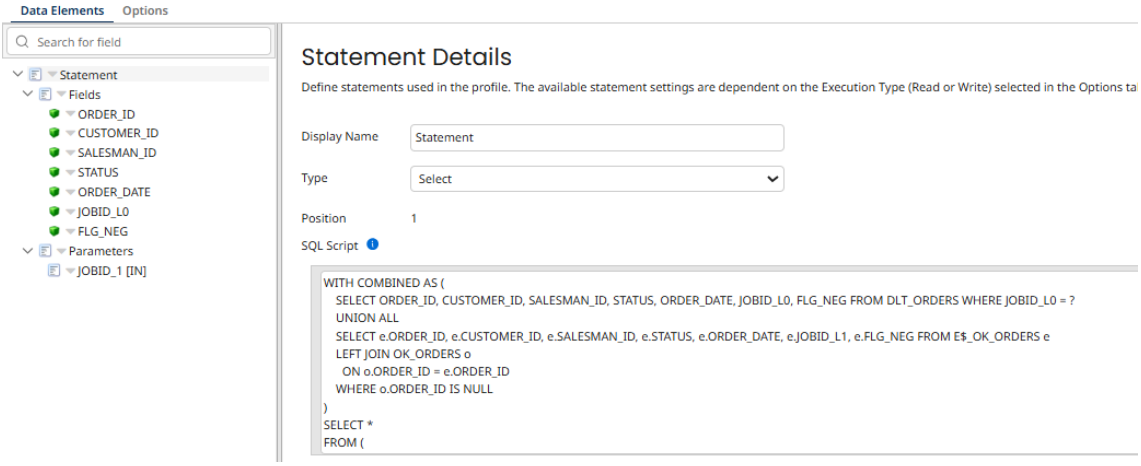


Figure 6.8: Database Legacy Connector and corresponding statement

The following query was implemented to merge new delta records with existing errors, while ensuring that only the most recent version of a record is processed:

```
WITH COMBINED AS (
    SELECT ORDER_ID, CUSTOMER_ID, SALESMAN_ID, STATUS, ORDER_DATE,
    JOBID_L0, INS_TIME, FLG_NEG FROM DLT_ORDERS WHERE JOBID_L0 = ?
    UNION ALL
    SELECT e.ORDER_ID, e.CUSTOMER_ID, e.SALESMAN_ID, e.STATUS, e.
    ORDER_DATE, e.JOBID_L1, e.INS_TIME, e.FLG_NEG FROM E$_OK_ORDERS
    e
    LEFT JOIN OK_ORDERS o
      ON o.ORDER_ID = e.ORDER_ID
    WHERE o.ORDER_ID IS NULL
)
SELECT *
FROM (
    SELECT c.*,
           ROW_NUMBER() OVER (
               PARTITION BY c.ORDER_ID
               ORDER BY c.JOBID_L0 DESC, c.FLG_NEG ASC
           ) AS RN
    FROM COMBINED c
) t
WHERE t.RN = 1
```

The Legacy Connector's exclusive operation with Database Profiles necessitated the introduction of a Map Step to transform the result set into a JSON format, essential for maintaining compatibility with the subsequent Oracle Database Connector, which utilises JSON profiles at the application level for its logical operations.

The records then undergo standard referential integrity and business rule checks.

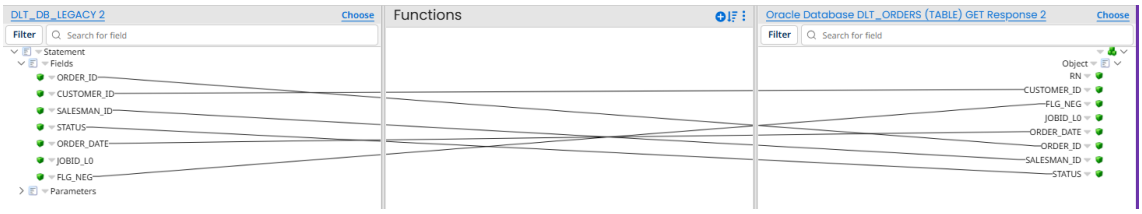


Figure 6.9: Mapping of data from Database to JSON profiles

The recycling logic assumes that during the interval between runs, the client may have updated master data (resolving referential errors) or corrected field-level values (resolving business rule violations).

If a record fails validation again, or if a new error is detected, the system triggers a notification; unlike the previous stage, where errors were simply logged and added to the Oracle table, this implementation features an automated Email Notification system. This involves:

- **Data Mapping**: a Map Step to format date fields and attributes according to Oracle's specific requirements
- **Payload Construction**: a combination of a **Data Process** step and a **Set Properties** step to dynamically generate an attachment containing the specific records that generated the error, providing the user a comprehensive view of the errors

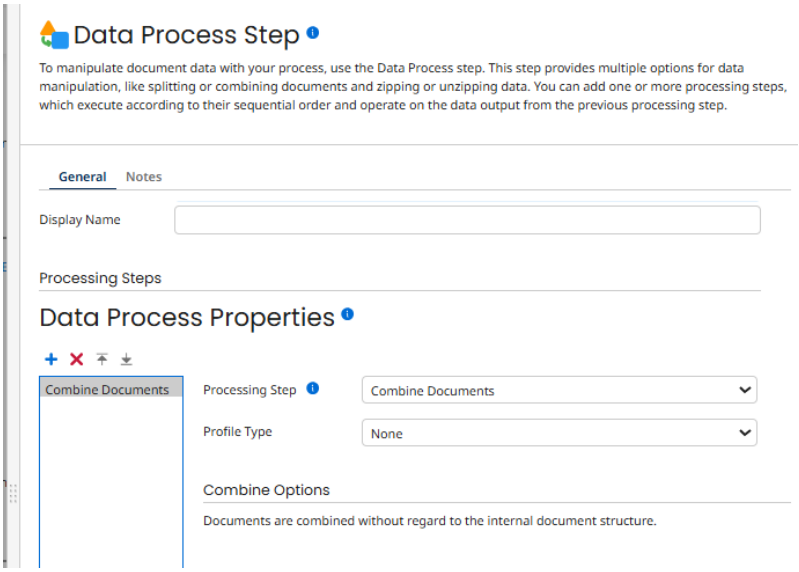


Figure 6.10: Data Process step

The final phase of the process handles the physical persistence of the data. Records that successfully pass all validation checks are inserted into the `OK` table with a current timestamp. The `E$` table is cleared if an error arising from it is resolved; in contrast, any records that fail the validation are inserted, or kept, into the `table E$` to await the next recycling cycle.

```
    DELETE FROM E$_OK_ORDERS e WHERE EXISTS (SELECT 1 FROM
   OK_ORDERS o WHERE o.ORDER_ID = e.ORDER_ID)
```

6.3 Final Workflow

The final process canvas is more elaborate than the initial design, reflecting a shift toward proactive error handling rather than a simple increase in technical complexity. By moving away from a strictly linear integration pattern, the workflow evolves into a closed-loop governance cycle, from the early detection of anomalies through remediation and, ultimately, successful persistence in the target system.

While the additional shapes and branching logic introduce a degree of redundancy, they also deliver tangible operational benefits that justify the expanded design. Resolved records are automatically purged from the `E$` tables, preventing the buildup of obsolete data over time. In parallel, distributing error reports via email removes the need for manual database inspections, significantly shortening remediation cycles and improving overall responsiveness.

The practical outcome of this automated reporting system is illustrated in Figure [6.11].

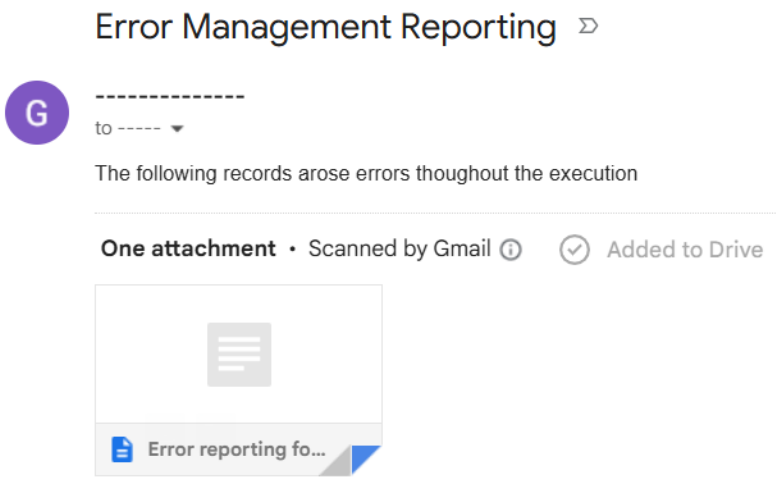


Figure 6.11: Automated email notification dispatched at the end of the execution, redacted for privacy purposes

As shown in the output, the email provides the developer with a complete summary and the necessary technical artifacts to address any issues immediately, without further manual research in the database.

6.4 Comparison and Rationale

Choosing between a standard integration pattern and a recycling-enabled architecture involves balancing architectural simplicity against operational resilience. In a conventional Boomi workflow, the design prioritizes throughput and minimal overhead: data moves in a single direction, and records that fail validation are written to an external staging table. This approach is highly effective in environments with consistently high data quality, as it avoids the cost of repeatedly reprocessing historical errors. However, it also shifts the entire remediation effort to manual, external

procedures.

Introducing the Recycle Error logic increases both the complexity of the process canvas and the sophistication of the underlying SQL, but it changes the nature of the integration. The workflow becomes state-aware, capable of responding to transient error conditions such as missing or delayed master data synchronization. In a purely linear flow, these records would remain in the `E$` tables until a developer intervenes, whereas in the recycling model, they are automatically re-evaluated at each execution, allowing the system to recover on its own as soon as the source data is corrected.

This iterative approach does introduce trade-offs that must be acknowledged. The use of complex CTEs and window functions increases the load on both the source database and the Boomi execution engine, leading to higher latency compared to a simple delta-based extraction. Maintenance is also more demanding, as schema changes require coordinated updates across advanced SQL queries and multi-layer mappings, from database profiles to JSON structures. For these reasons, the recycling pattern is best suited to Master Data domains where referential integrity is volatile or where the business impact of missing or delayed records outweighs the cost of additional processing.

Ultimately, the decision to implement recycling logic in this project was driven by the need for a proactive feedback loop. By combining automatic reprocessing of historical errors with structured email notifications, the architecture ensures that issues are not only reported but are continuously reassessed as source data evolves. In this way, the `E$` tables function as a dynamic buffer rather than a static repository of failures, making this approach particularly well-suited to complex business environments where data dependencies are the rule rather than the exception

Chapter 7

Architectural Refactoring of the Workflow

While the previous chapters established a resilient and functional pipeline, the final implementation phase focused on a radical refactoring and simplification of the process canvas, aimed at eliminating computational redundancy by consolidating multi-stage lookups into a **single, atomic validation** engine.
The preliminary architectural design relied on a sequential series of Map Shapes and Document Cache Lookups, designed to enforce referential integrity within the **OK** Stage and to execute bidirectional temporal delta checks, specifically "Yesterday-vs-Today" and "Today-vs-Yesterday" comparisons, within the **DLT** Stage; however, this multi-branch approach resulted in unnecessary overhead, prompting a more streamlined and efficient design.

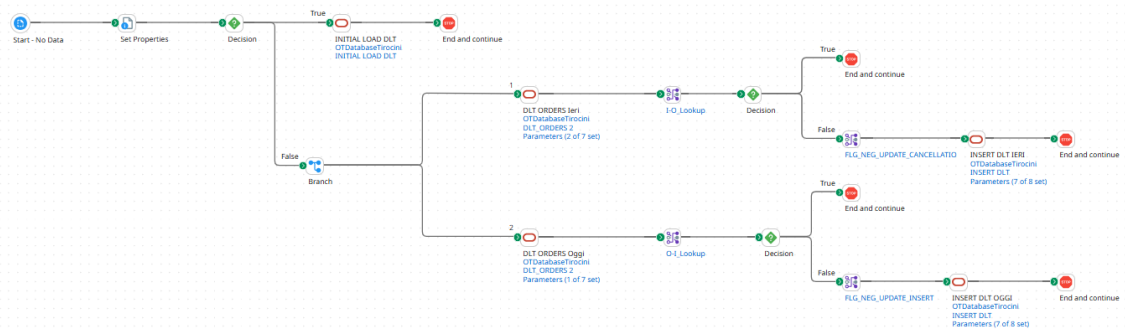


Figure 7.1: Optimized configuration of the `DLT` Stage

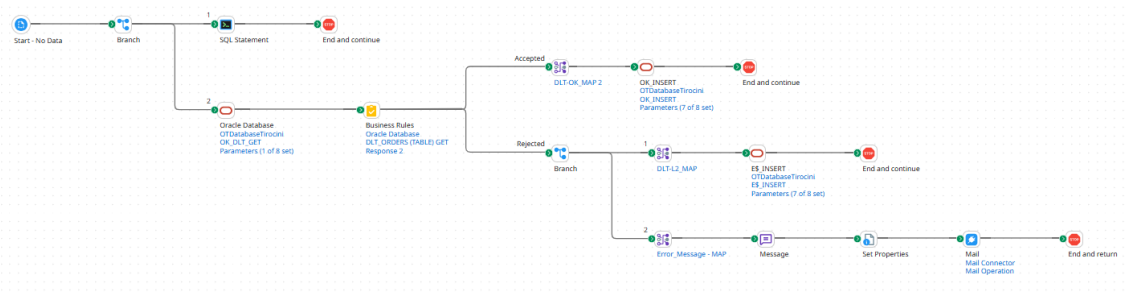


Figure 7.2: Optimized configuration of the `OK` Stage

7.1 Rationale for the Boomi Process Refactoring

The architectural evolution of the `OK` and `DLT` Stages represents a major step in the project's development, marking a transition from a rigid procedural framework to a rule-centric validation engine. Initially, referential integrity and bidirectional temporal delta checks were conceived as sequential workflows where documents traversed a series of Map Shapes, each specifically configured to execute isolated lookups against pre-populated Document Caches. Although functional, an in-depth analysis revealed that this decentralized structure introduced significant computational redundancies.

In the Boomi runtime environment, each transformation within a Map Shape requires a full cycle of data parsing and re-serialization. Consequently, chaining these operations generates a cumulative overhead that increases memory consumption and execution latency of the Atom. Furthermore, the dependence on Document Caches proved memory-intensive, as maintaining persistent access to indexed structures within the JVM heap or local storage requires substantial resources. This reliance on repeated I/O operations and frequent context switching between the transformation engine and cache storage created significant performance bottlenecks, which became increasingly critical as data volumes scaled.

The primary driver for the refactoring was the observation that Boomi's **SQL Lookup Function**, supported by a **Database (Legacy) connection**, could be leveraged for various tasks throughout the pipeline. By integrating this capability directly into the `DLT` Map Functions and `OK` Business Rules, the need for intermediate mapping stages and pre-populated caches was entirely eliminated. This shifted the design from a *"mapping-first"* approach to an *"atomic validation"* model, substantially simplifying the process canvas.

This streamlined architecture also facilitates a robust **Error Recycling** mechanism, providing a unified entry point for both new and reprocessed records. Rather than forcing documents through a redundant sequence of maps and memory-heavy cache calls, the system funnels "temporarily non-compliant" data directly into the consolidated rules engine. This ensures that each record, regardless of its origin, is subjected to the same rigorous standards of integrity in a synchronized, optimized operation. By eliminating the need for separate, resource-heavy recovery flows, the pipeline is fundamentally reshaped into a resilient, self-healing ecosystem where all data is validated against the most current master records with minimal latency.

Beyond internal optimization, this shift carries significant upstream implications. By centralizing the core logic downstream, the requirement for a dedicated Preparation Stage in the L1 layer was removed. Data can now flow directly from the source to the validation engine, reducing the overall hop-count and decreasing the probability of state-desynchronization errors.

7.1.1 Transition to the Atomic Validation Model

The optimization of the ETL architecture was uncovered through a systematic performance review and an exhaustive exploration of Boomi's native component capabilities during the development cycle. This investigative phase was prompted by the observation that the traditional mapping approach began to exhibit non-linear latency growth as data volumes increased. Initial profiling of the `DLT` and `OK` stages revealed critical bottlenecks resulting from redundant maps, embedded script logic, and cache-heavy operations required for referential integrity and business rule validation. Specifically, execution logs showed that a significant portion of the total processing time was not spent on data movement, but rather on the initialization of mapping contexts and the repeated invocation of external lookups across multiple, fragmented shapes in the process canvas.

The implementation strategy shifted toward a declarative model by leveraging the **SQL Lookup function**, which natively supports complex logic without the need for custom scripting or extensive branching. In the legacy implementation, record validation followed a *"step-by-step"* procedural logic: each requirement, ranging from referential integrity to temporal delta checks, was handled by a separate Map Shape with dedicated error handling. This fragmentation could result in multiple points of failure and significant memory overhead, as the Atom had to maintain state and execution context across several different shapes as part of a single unit of work.

By transitioning to an **Atomic Validation** model, these disparate checks were consolidated into a single, high-performance evaluation engine, ensuring that the document profile is parsed only once, after which the engine evaluates the entire set of constraints in a centralized logic pass. This strategy is applied with specific functional variations across the pipeline:

- **DLT Stage Synchronization:** the optimization involves using SQL lookups within the **Map Step** to execute temporal comparisons. This allows the system to perform "Yesterday-vs-Today" and "Today-vs-Yesterday" logic directly during the mapping phase. By embedding the lookup within the map, the pipeline can determine the record's delta without the overhead of complex branching, thus effectively flattening the logic
- **OK Stage Efficiency:** the **Business Rules** shape was configured to perform direct SQL lookups as internal validation steps. By treating referential integrity as a native condition within this component rather than an external procedural challenge, the process achieves a significantly higher degree of computational efficiency. If a record fails any condition, be it a missing foreign key or a business constraint violation, the component flags it immediately with a specific error message. This granularity ensures that subsequent pipeline stages receive actionable feedback while maintaining the same rigorous detail as the previous multi-map implementation within a more stable canvas

Ultimately, this consolidated approach ensures that the data enters and leaves the validation engine as a single unit. By reducing the number of branches and stages that the data must navigate, the refactored architecture provides superior control over the validation results, leading to a more scalable and robust execution environment.

7.1.2 The Optimized Standard Pipeline

Following the architectural shift, the ETL process was organized into a standardized pipeline where the transition to an atomic model is applied through specific functional variations.
The following subsections highlight how the consolidation of implementation logic and the reduction of component overhead are practically realized within the two primary modules, ensuring a balance between throughput and long-term maintainability: the **DLT Stage**, focused on state synchronization, and the **OK Stage**, dedicated to relational integrity and business constraints [11].

DLT Stage: Bi-directional Delta Detection via Map Function

As previously described, the DLT stage serves as the pipeline's synchronization layer, responsible for identifying changes between the incoming dataset and the persisted system state. In the optimized architecture, the challenge of comparing large-scale datasets was addressed by moving away from memory-intensive document caches toward a database-centric lookup strategy.

The core of this stage is the implementation of a bidirectional temporal check, referred to as "Yesterday-vs-Today" and "Today-vs-Yesterday" logic. This mechanism is crucial for determining the operational status of each record:

- **Forward Synchronization (New/Updated):** comparing current source data against the historical state to identify inserts or modifications

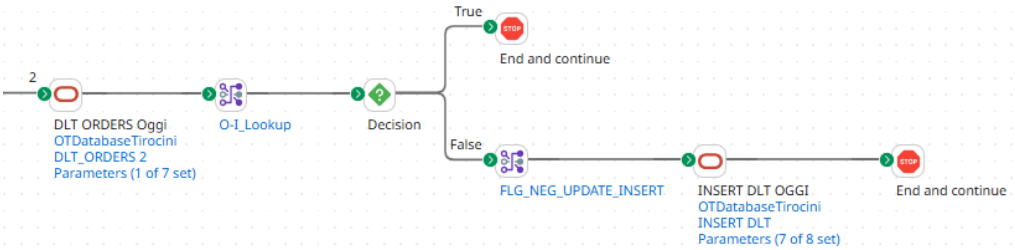


Figure 7.3: Close-up of the DLT workflow: map steps to perform the lookup in the "Yesterday" cache

- **Backward Synchronization (Deleted):** evaluating the historical state against the current feed to detect records that are no longer present in the source system

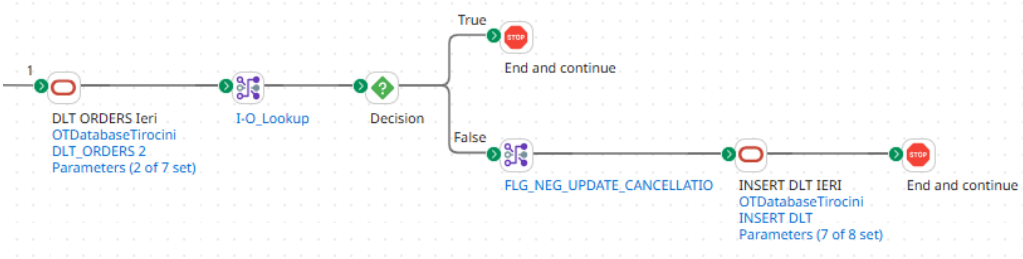


Figure 7.4: Close-up of the DLT workflow: map steps to perform the lookup in the "Today" cache

By embedding **SQL Lookup functions** directly within the **Map** step, the `DLT` stage performs these high-frequency comparisons without the computational cost of loading entire tables into the Atom's local memory. This allows the mapping engine to resolve the delta state of each document in real-time as it traverses the profile, ensuring that only necessary updates are propagated downstream.
The technical core of this stage lies in a **Custom Map Function** that consolidates this implementation logic into an atomic execution unit. As shown in Figure [7.5], its complexity stems from the need to synchronize full record attributes, environment variables, and metadata prior to the lookup.

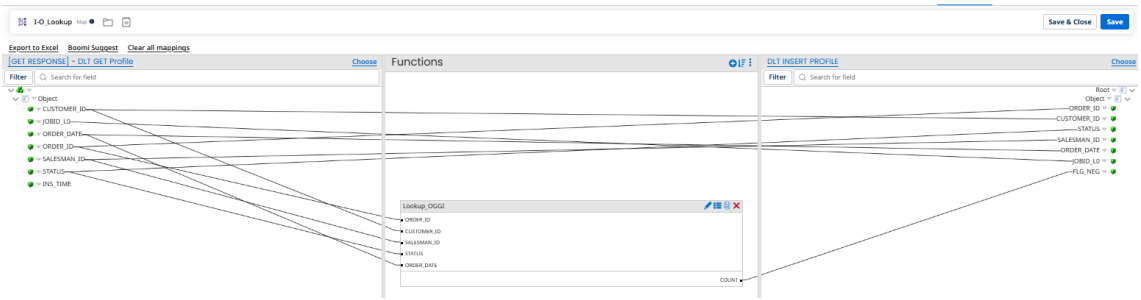


Figure 7.5: Map Step with custom Lookup Function to handle delta computation

This stage, given the nature of the source data, employs the `MINUS` strategy through two symmetric evaluation paths:

- **Yesterday-vs-Today (Deletions/Updates):** for each existing record in the historical dataset, the system performs a lookup against the current `STG` table (filtered by the `JOBID` of the current run). If no match is found across functional attributes, the record is flagged as deleted or updated
- **Today-vs-Yesterday (Insertions/Updates):** incoming records are cross-referenced against the `STG` table associated with the `JOBID` of the previous execution. A match indicates a persistent record, while a *"no-hit"* result identifies a new insertion or an update

For these custom lookup functions to be successful, they must be rigorously aligned with the underlying database schema and its specific data format constraints. Particular emphasis must be placed on the temporal attributes, as the **Oracle Database** environment strictly enforces the `"dd-MON-yy"` format for date fields. To ensure compatibility during the comparison process, the mapping logic includes a transformation layer that normalizes date strings, preventing type-mismatch errors that would otherwise invalidate the delta detection.

To ensure accuracy, the SQL queries are dynamically parameterized using the **"Get Dynamic Process Property"** function within the map step to retrieve the unique identifiers for the synchronization window, specifically the current and previous `JOBID`s. This dynamic injection of metadata allows the SQL engine to scope its search precisely, ensuring that the `MINUS` strategy remains performant even as the `STG` table grows in volume.

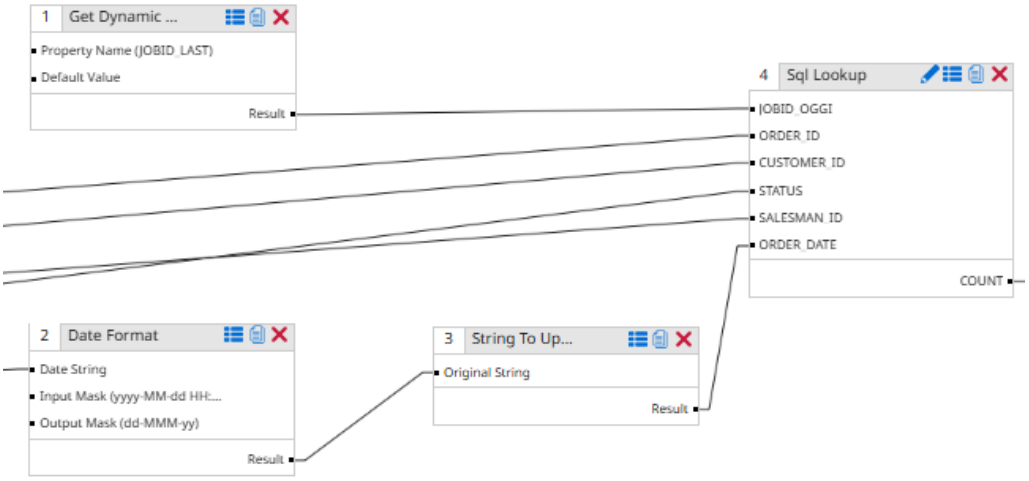


Figure 7.6: Custom function employed to execute the `MINUS` computation

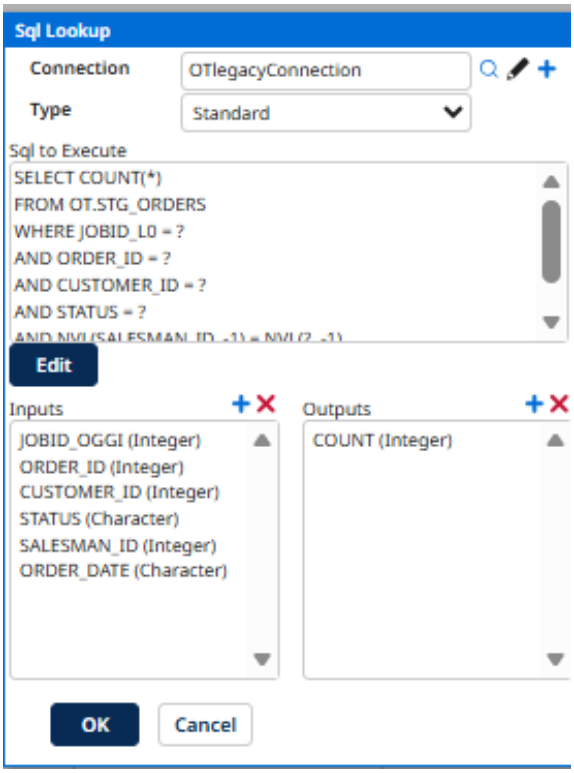


Figure 7.7: SQL Lookup operation employed to account for modifications, deletions, or insertions

The outcome of these specular lookups is tracked using a technical field, the `FLG_NEG`, which acts as a discriminant for delta detection and is populated via custom logic within the mapping function: a "zero-hit" result indicates that no match was found, while a non-zero value confirms the presence of an identical record that will consequently be filtered out by a downstream Decision Shape.

Once a record has been identified as a delta, the `FLG_NEG` is repurposed as a functional status indicator, assuming values of `"0"` for insertions/updates or `"1"` for deletions, and is then persisted into the `DLT` table (Fig. [7.9, 7.10]). This granular

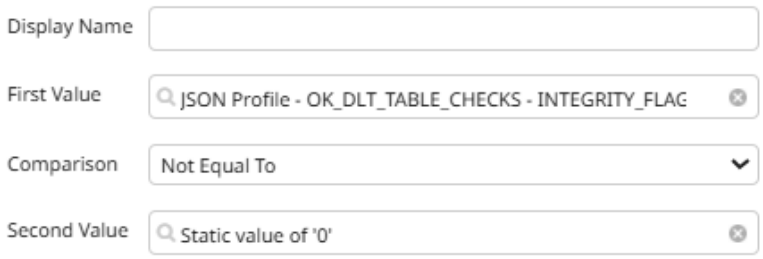


Figure 7.8: Decision Shape to determine if the record has to be saved into the `DLT` table

flagging ensures that the final pipeline stage can distinguish state changes with high precision, maintaining the historical integrity of the data track.

Figure 7.9: New Map Step to handle update or insertion of new records

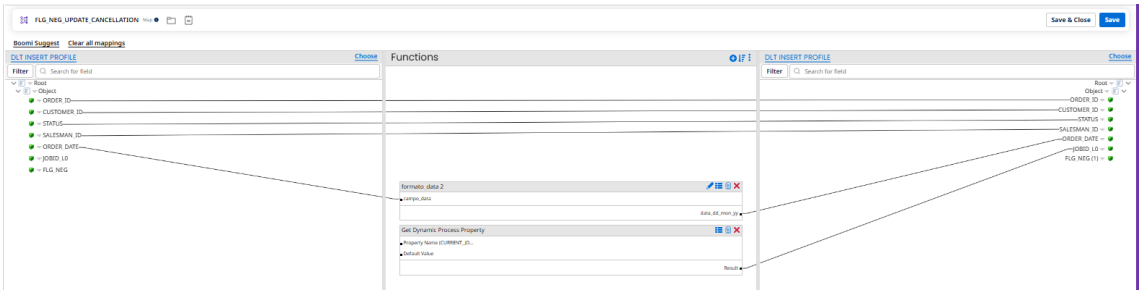


Figure 7.10: New Map Step to handle cancellation or update of old records

OK Stage: Multi-conditional Integrity via Business Rules

By consolidating multiple integrity checks (e..g, foreign key existence verification, multi-conditional business rule validation) into a single, high-performance logical gate, the system determines the "Accepted/Rejected" classification of each record in a single processing pass. This approach not only reduces overhead by minimizing document manipulation but also improves maintainability by introducing a single, centralized logical gate.

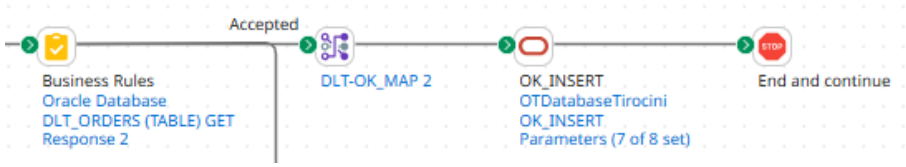


Figure 7.11: Close-up of the `OK` workflow: accepted branch of the Business Rules Shape

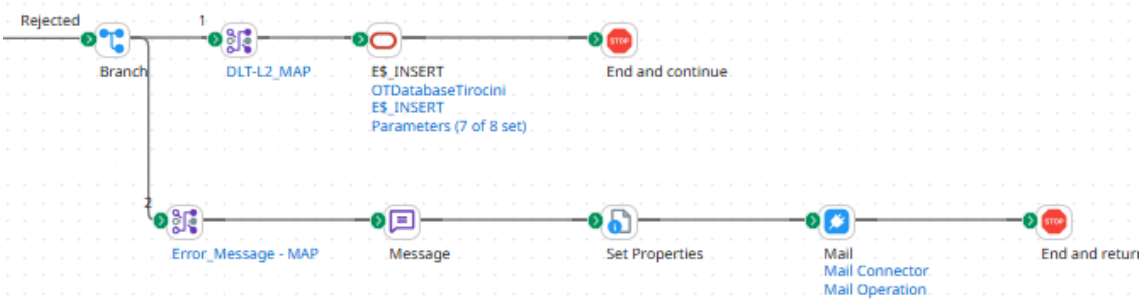


Figure 7.12: Close-up of the `OK` workflow: rejected branch of the Business Rules Shape

The refactored Business Rules shape enforces a sequential execution order while preserving validation priority. Specifically, using the examples referenced throughout this thesis, it first checks referential integrity for `SALESMAN_ID` (nullable, with explicit null-handling logic), followed by `CUSTOMER_ID` (mandatory match required).
A custom formula processes the raw SQL Lookup results from the Oracle Connector, converting hit counts into logical inputs for the Business Rules to enable conditional logic. Specifically, it verifies whether a matching record exists in the master table for the input foreign key via primary key lookup; a zero count signals referential integrity failure (no matching record), triggering downstream rejection.

Post-referential checks, the shape applies a sequence of field-level business rules across all input table columns, covering format validation, range constraints, and semantic consistency. This declarative approach ensures uniform standards by consolidating the original multiple components into a single, maintainable unit, while preserving execution tracing for debugging.

While the core architecture of the remaining pipeline elements remains consistent with the previously defined framework, specific adjustments were required to propagate errors identified by the Business Rules engine to the end user effectively. To facilitate this, a dedicated Map Step was implemented prior to the Message and subsequent Mail shapes, serving to capture and format the error metadata destined for downstream notification.

The error output generated by the Business Rules engine is stored as a **Document Property**, which can be programmatically retrieved through a custom function. As illustrated in Figures [7.17, 7.18], this function is embedded within the mapping stage, where the property is extracted, transformed, and mapped onto a purpose-built XML profile. The scripting logic utilized to retrieve the error message is a critical component of the flow; without this intermediate transformation, the raw

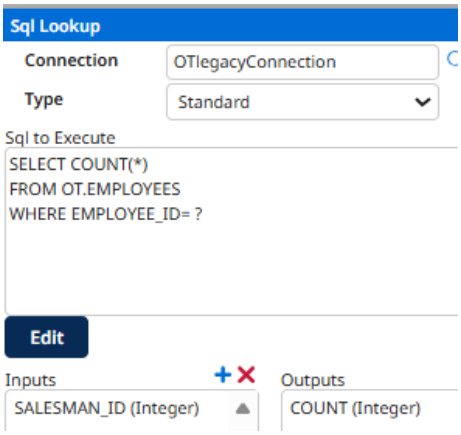


Figure 7.13: Custom logic employed to check the referential integrity constraint on the `SALESMAN_ID` field

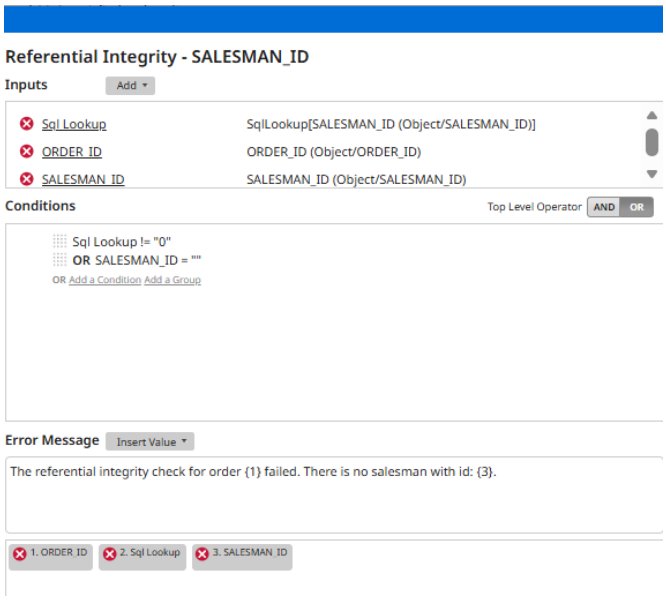


Figure 7.14: Business Rule implementation of the `SALESMAN_ID` constraint

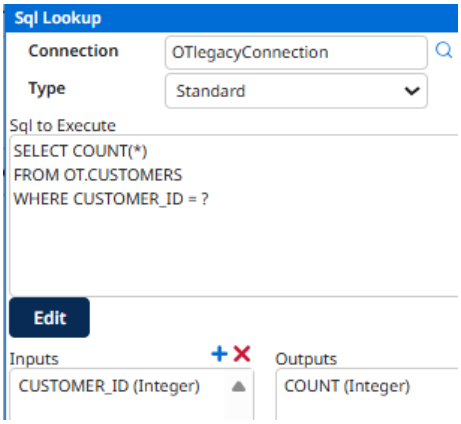


Figure 7.15: Custom logic employed to check the referential integrity constraint on the `CUSTOMER_ID` field

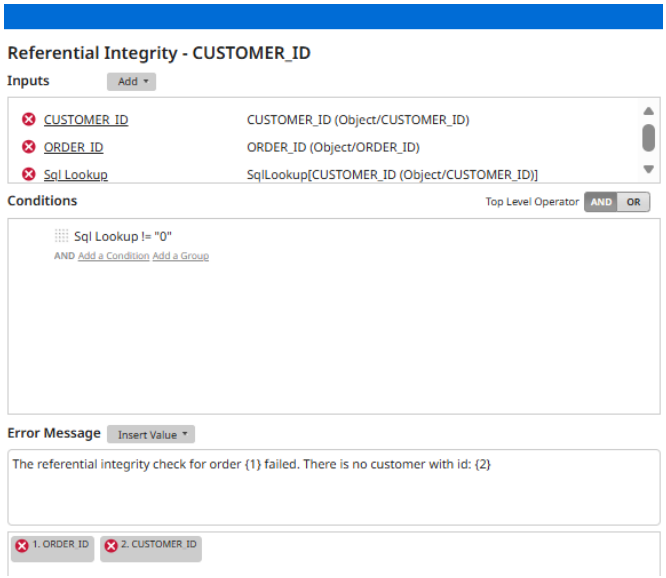


Figure 7.16: Business Rule implementation of the `CUSTOMER_ID` constraint

property data would fail to translate correctly into a structured XML format, leading to potential data corruption or loss of diagnostic detail.

Once constructed, the error payload is included as an attachment in the automated email notification. It is important to note that the **"Combine Documents"** feature is enabled within the Mail connector configuration. This setting consolidates all error messages from an execution batch into a single exhaustive notification, thereby preventing the transmission of individual emails for each failed record, which would otherwise lead to an inefficient and overwhelming alert volume for the client.

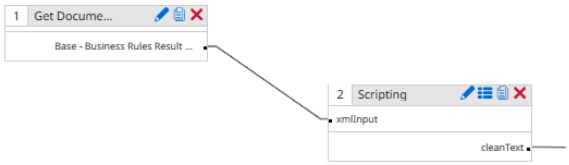


Figure 7.17: Custom function employed to extract the error message from the Business Rule Result

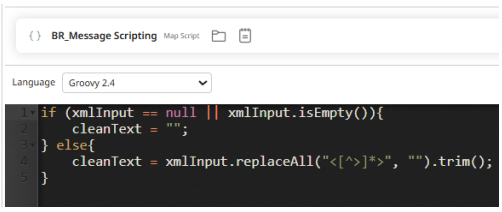


Figure 7.18: Custom Scripting to extract the error message from the component response

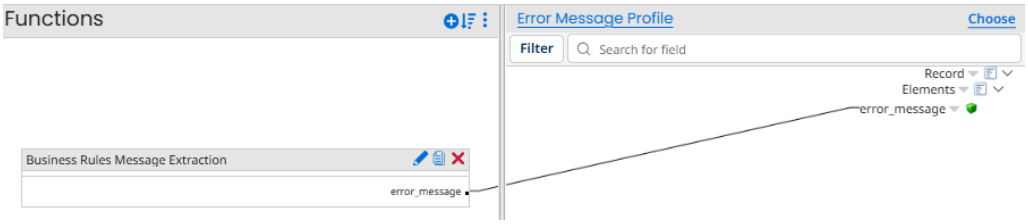


Figure 7.19: Map Step placed before the Message Step, to extract the error message

As shown in Figures [7.21, 7.22], the automated email notification is constructed within the Mail operation to deliver the client a structured report of the validation results. The specific configuration of the email components is:

- **Subject:** configured with the static value *"Error Management Reporting"*, ensuring that the communication is easily recognizable within the target's inbox as a specific report for order data integrity
- **Body:** contains the static message *"The following records arose errors throughout the execution"*, serving as a clear introductory header for the data steward regarding the status of the batch
- **Attachment:** the payload is delivered as an attached file, which includes:
  - **File Name:** dynamically generated by concatenating the static prefix *"Error reporting for run: "* with the Current Date provided by the Boomi runtime, formatted as `"yyyy-MM-dd"`. This naming convention facilitates efficient log archiving and ensures that each report is uniquely identifiable by its execution date
  - **Content:** the consolidated XML data produced. Each entry in the XML provides a granular breakdown of the specific failures captured from the Business Rules document properties, mapping each `ORDER_ID` to its corresponding validation error

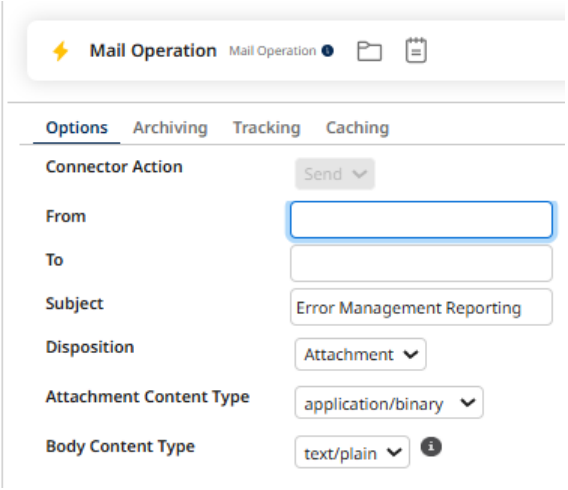


Figure 7.20: Example of the Mail Connector configuration

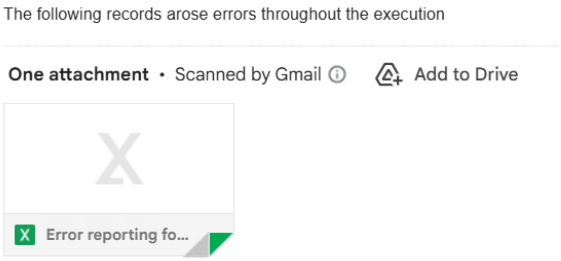
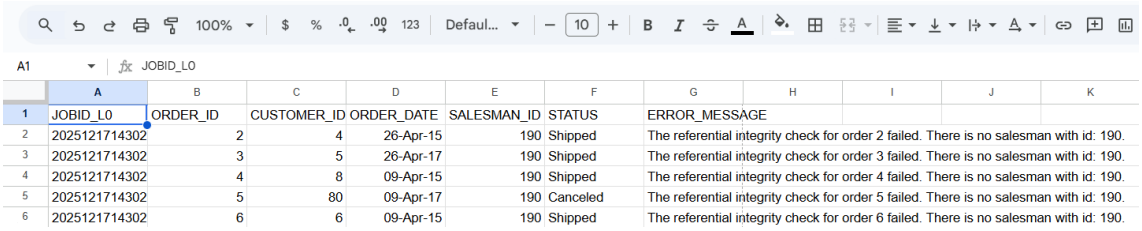


Figure 7.21: Email generated from the Boomi process



	A	B	C	D	E	F	G
1	JOBID_L0	ORDER_ID	CUSTOMER_ID	ORDER_DATE	SALESMAN_ID	STATUS	ERROR_MESSAGE
2	2025121714302	2	4	26-Apr-15	190	Shipped	The referential integrity check for order 2 failed. There is no salesman with id: 190.
3	2025121714302	3	5	26-Apr-17	190	Shipped	The referential integrity check for order 3 failed. There is no salesman with id: 190.
4	2025121714302	4	8	09-Apr-15	190	Shipped	The referential integrity check for order 4 failed. There is no salesman with id: 190.
5	2025121714302	5	80	09-Apr-17	190	Canceled	The referential integrity check for order 5 failed. There is no salesman with id: 190.
6	2025121714302	6	6	09-Apr-15	190	Shipped	The referential integrity check for order 6 failed. There is no salesman with id: 190.

Figure 7.22: Example of an attachment

7.1.3 The Optimized Recycle Error Pipeline

Following the successful optimization of the standard `OK` Stage, the same atomic validation principles were applied to the **Error Recycling** strategy. This phase necessitated a critical infrastructural change: the replacement of the standard Oracle Database connector with a Database (Legacy) Connector. This modification supports the execution of high-complexity SQL queries, designed to merge records from the `E$` staging tables into the main `DLT` table, while simultaneously performing advanced operations, such as result sorting and calculating `ROW_NUMBER` window functions.
As established in the preceding Chapters, these specific relational operations are not natively supported by the standard connector's abstraction layer, making the legacy implementation the only viable option for this logic.

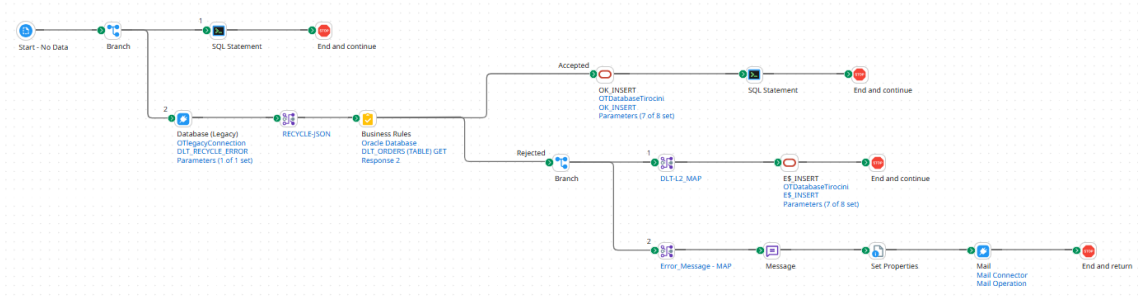


Figure 7.23: Optimized `OK` Stage configuration incorporating Recycle Error logic

With the retrieval logic consolidated, an intermediate Map Shape was introduced to ensure data consistency and normalize the disparate date formats from various staging tables into a single, unified standard. Additionally, a specific technical adjustment was required within the Business Rules engine to handle temporal comparisons.

Figure 7.24: Map Step implemented before the execution of the Business Rules Component

Given that the Boomi runtime captures the current date in the `"yyyyMMddHHmmss"` format, a transformation was implemented within the rule itself to reformat the document's date field. This transformation is strictly confined to the Business Rule component, preserving the document's original formatting for the rest of the process, and is crucial for the accurate execution of the integrity check on the order date (`order_date`$\leq$`current_date`).

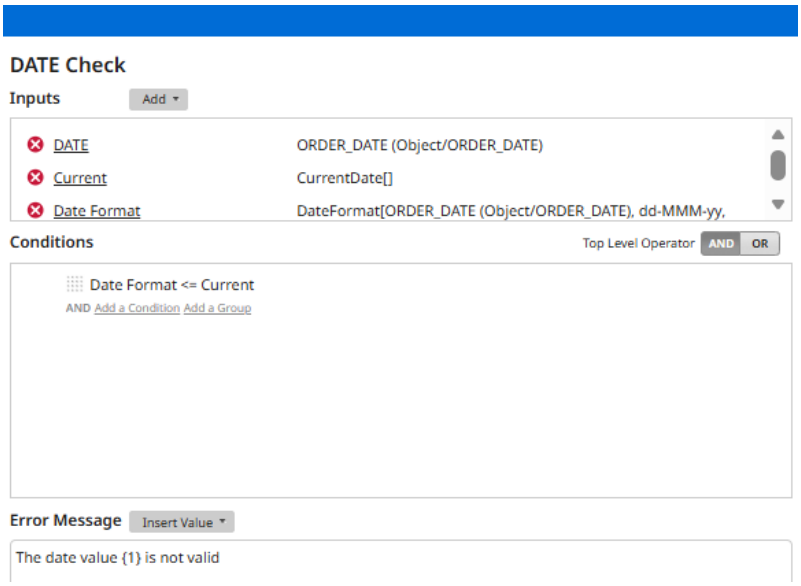


Figure 7.25: New implementation of the `ORDER_DATE` check to account for the different date formats

The downstream logic remains synchronized with the standard pipeline, with the final addition of a Cleanup Stage; once a recycled record successfully passes all validation criteria and is inserted into the `OK` table, a targeted Program Command operation is triggered to delete the corresponding entry from the `E$` table. This ensures the integrity of the recycling loop and prevents redundant processing in future execution cycles, effectively completing the self-healing lifecycle of the data.

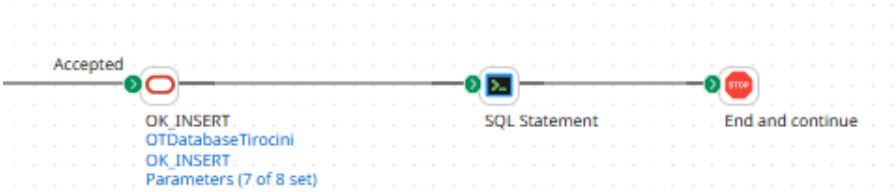


Figure 7.26: Close-up of the `OK` workflow with Recycle Error Optimization: accepted branch of the Business Rules Shape

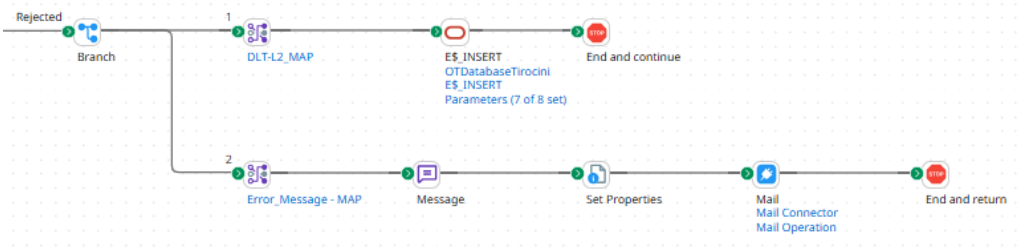


Figure 7.27: Close-up of the `OK` workflow with Recycle Error Optimization: rejected branch of the Business Rules Shape

7.2 Quantitative and Qualitative Gains

The architectural transition from a procedural, map-heavy validation to an atomic, rule-centric engine yielded significant improvements in both measurable performance metrics and overall integration framework quality. These gains can be categorized into computational efficiency (quantitative) and systemic robustness (qualitative).

The most immediate advantage of the refactoring is the reduction in execution latency and structural overhead; while the baseline model suffered from a linear accumulation of processing time, the current architecture eliminates these redundant cycles by centralizing logic within a single atomic execution.

Empirical Validation via Execution Logs

The theoretical advantages of this transition are explicitly demonstrated by a comparative analysis of execution logs. The baseline cache-based model reveals a fragmented execution pattern, where the Atom is burdened by repeated **Document Cache Load** and **Retrieve** operations, incurring significant overhead for cache management. This architecture creates a procedural bottleneck, as the system must persistently allocate JVM heap space to maintain document snapshots for comparison.

In contrast, the refactored SQL Lookup model streamlines the synchronization logic within the mapping phase. By offloading the computational *"heavy lifting"* of the `MINUS` strategy to the database engine, the transition to direct lookups optimizes resource utilization. While the initial connector call reflects the inherent complexity of the Oracle query, the subsequent document processing is near-instantaneous: the delta state is resolved via the `FLG_NEG` flag within a single **Decision Shape** pass. This empirical data confirms the following improvements

- **Reduced Memory Footprint:** by eliminating the reliance on multiple, large-scale Document Caches that occupied significant JVM heap space, the Atom's memory consumption stabilized. The transition from multiple map-based integrity checks to a direct SQL Lookup via the Database (Legacy) connection allowed for "on-demand" validation without pre-populating massive caches in memory
- **Optimization of Execution:** the log analysis reveals a substantial reduction in the number of shapes traversed. The baseline model required approximately 15 discrete shapes (including multiple branches and decision steps) to achieve full validation; in contrast, the refactored process funnels the document through a single Business Rules Shape or custom Map function, minimizing context switches and CPU cycles dedicated to managing document state
- **Elimination of Parsing Redundancy:** in the procedural model, document profiles were subjected to parsing and re-serialization through at least four consecutive Map shapes. The atomic model reduces this to a single primary parsing within the logic engine, effectively "flattening" the process

It is important to note that while the execution time reduction per single document appears incremental (∼15% improvement, from 934 ms to 797 ms), it significantly lowers the probability of JVM heap exhaustion and CPU contention. By minimizing the "per-document overhead" inherent in Boomi's shape-to-shape data routing, the atomic model ensures a predictable and linear performance curve.

Qualitative Enhancements and Systemic Robustness

Beyond raw speed, the refactoring introduced qualitative improvements that fundamentally changed how the pipeline is maintained and monitored:

- **Centralized Logic Management:** the consolidation into a single Business Rules gate provides a *"Single Point of Truth"*. Developers no longer navigate a complex web of maps and decision branches to update validation criteria; all constraints are housed within a single, declarative interface
- **Granular and Parallel Error Reporting:** unlike the procedural model, where execution was halted by a Stop or Message Shape immediately after a specific Decision failure, the consolidated engine evaluates multiple conditions simultaneously. This leads to more descriptive feedback, as the system can report multiple independent failures in a single pass
- **Elimination of Custom Scripting Debt:** using Boomi's native SQL Lookup functions within the Business Rules shape, instead of fragmented map-scripts, ensures that the solution adheres to "Low-Code" best practices, providing long-term compatibility and reducing technical debt

Ultimately, the shift to atomic validation transformed the `OK` and `DLT` Stages from a potential bottleneck into high-performance gateways. The refactoring successfully mitigated the risks of memory exhaustion and execution timeouts, providing a resilient foundation for the self-healing ecosystem that now characterizes the entire ETL pipeline.

Comparative Performance Metrics

The structural impact of the transition to the atomic model is summarized in a high-level overview of the performance gains and architectural improvements observed across the entire validation period.
These comparisons are based on execution logs collected over the full evaluation window, obtained by analyzing runs that include the improvements discussed in the previous sections, with a focus on shape-traversal efficiency, memory footprint, and total execution latency. Merging these metrics into a single comparative structure, it is possible to observe how the atomic refactoring addresses systemic bottlenecks while ensuring a consistent and scalable data flow across all pipeline modules.

Metric	Initial Implementation	Atomic Refactoring
Logic Structure	**OK:** fragmented; multiple branches/decision steps **DLT:** multi-branch workflow using sequential shapes for temporal checks	**OK:** centralized logic within a single evaluation step **DLT:** consolidated bidirectional delta-check logic in custom Map functions
Data Access	**OK:** multiple cycles; documents re-serialized across Map stages **DLT:** in-memory; reliance on Document Cache Add and Retrieve	**OK:** single cycle; parsed once at each Business Rules entry **DLT:** real-time; direct SQL Lookups via Database connection
Memory Management	Reliance on persistent Document Caches and high JVM heap consumption for both `OK` and `DLT`	Optimized: on-demand SQL Lookups, reducing memory footprint for both processes
Execution Flow	**OK:** cumulative latency from discrete steps (Total ∼934 ms) **DLT:** procedural; cumulative latency from cache I/O	**OK:** consolidated; single execution pass (Total ∼797 ms) **DLT:** streamlined; handled natively using `FLG_NEG` indicators
Efficiency	**OK/DLT:** sequential, step-by-step verification leveraging separate Decision and Cache shapes	**OK/DLT:** concurrent execution; temporal state synchronizations implemented as atomic lookup functions
Scalability	Non-linear latency growth due to procedural overhead and sequential checks	Linear performance; stable execution via native libraries and atomic passes

Table 7.1: Consolidated Comparative Analysis of Validation (`OK`) and Delta Detection (`DLT`) Strategies after the Refactoring

Chapter 8

Transitioning to DataHub

A significant evolution in the integration framework involves the adoption of Boomi DataHub, a cloud-native Master Data Management (MDM) solution designed to centralize, synchronize, and enrich data across the enterprise. Unlike traditional relational databases, DataHub acts as an authoritative repository that enforces data quality and consistency for entities originating from multiple, heterogeneous sources.

1. **Centralized Governance**: business rules are defined once within the DataHub model and applied to all contributing sources, ensuring consistency across the ecosystem
2. **Automated Enrichment**: the platform can automatically normalize fields (e.g., standardizing addresses or status codes) before the data reaches the `ODS` Stage
3. **Advanced Referential Integrity**: rather than configuring manual SQL lookups in Business Rules and managing the hit/miss logic via count values, the system automatically maps each field to its reference at ingestion time

In the current implementation, referential integrity and business rules are enforced during the L1 (OK) Stage via Business Rules and custom Map Functions, employing SQL Lookup strategies.
To overcome the limitations of this approach, the proposed evolution shifts the responsibility of data validation from the Boomi Integration process to the DataHub repository, as shown in Figure [8.1].

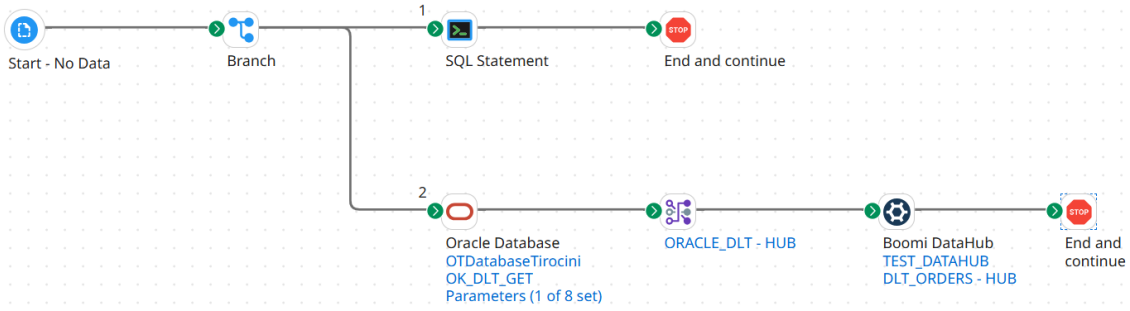


Figure 8.1: Implementation of the OK stage leveraging Boomi DataHub

Records that violate these constraints are automatically routed to a **Quarantine area**. Unlike traditional integration flows that rely on automated error recycling,

where failed records are blindly re-injected into the pipeline, DataHub introduces a governed remediation process. Here, a **Data Steward** can analyze failures through a dedicated interface, manually correcting values or resubmitting records once underlying issues (e.g., missing master references) are resolved.

8.1 DataHub Architecture

Boomi DataHub functions as the central governing node in a Hub-and-Spoke architecture; disparate systems (Sources) contribute data to a centralized Repository, which persists Golden Records, the single, authoritative "truth" for each business entity.

The key features introduced by the DataHub environment are [7]:

- **Repositories:** virtual containers hosted in the Boomi Hub Cloud that act as runtimes for validated data. A single Repository can host multiple data domains (e.g., Customers, Orders, Employees), isolating their datasets while providing a unified endpoint for querying Golden Records
- **Models:** the logical definition of a master data domain. A Model defines the structure, data types, and validation rules that every incoming record must satisfy before being committed to the repository
- **Source Management:** DataHub tracks the origin of every attribute through Source Rankings and Cross-Reference tables. This ensures that if multiple systems provide conflicting information, it can automatically determine which source is most trustworthy based on predefined policies
- **Match and Merge Engine:** to prevent data duplication, the platform utilizes configurable Match Rules that allow the system to identify if an incoming record from one source corresponds to an existing record from another
- **Governance Workflow:** introduces human-in-the-loop governance so that records that fail validation or trigger matching conflicts are diverted to the Quarantine, where Data Stewards can perform manual remediation, ensuring that the integrity of the Golden Records is never compromised

By decoupling data quality from the integration logic, DataHub transforms the middleware from a simple transport layer into a robust governance engine. This shift ensures that data consistency is maintained across all systems, providing a scalable foundation for advanced analytics and operational reliability.

As data flows from sources into the Hub, it undergoes a multi-stage validation process to ensure it is complete, conforming, accurate, and unique:

1. **Validation Phase:** ensures that the batch data adheres to the domain model definition, checking for required fields and correct data formats
2. **Enrichment Phase (Data Quality Steps):** external checks or business rules are applied to enhance the data. This may involve calling a Boomi Integration process or executing internal business logic

3. **Incorporation Phase:** identify potential duplicates between incoming records and existing Golden Records. Advanced rules may utilize "Fuzzy Matching" algorithms to identify similar records based on probability thresholds rather than exact equality

This multi-stage process culminates in the **Active Remediation** workflow. While traditional middleware would discard a record at the first sign of failure, DataHub persists the failed state within the stewardship layer. This allows the Data Steward to act as a final *"intelligent filter"*, resolving conflicts that automated logic cannot handle, thereby ensuring that the Golden Record state represents the highest possible standard of data integrity.

8.2 Implementation of the Integration Process

The integration of Boomi DataHub into the ETL framework addresses the inherent complexities of managing high-volume data validation within a standard middleware layer.

In the legacy workflow, enforcing referential integrity relies on a lookup-intensive pattern: to validate a transactional record (e.g., an Order), a Business Rules shape must execute targeted SQL queries to verify the existence of its corresponding master entities (e.g., a Customer). This binary logic, routing records to error tables based on a `COUNT` result, presents significant scalability bottlenecks. As data models grow in complexity, managing multiple foreign key dependencies and evolving business requirements through fragmented, manual lookups becomes increasingly unsustainable.
Transitioning these responsibilities to DataHub replaces these iterative queries with a **centralized MDM-engine**. This shift consolidates data quality into a proactive service, ensuring ecosystem-wide consistency by resolving dependencies at the orchestration layer rather than through individual integration tasks.

Architectural Optimization and Implementation

The transition to an **MDM-led** architecture optimizes the integration workflow through automated synchronization and centralized governance. Once source data is cleansed and transformed into the Hub's canonical model, DataHub natively manages cross-references, enabling automated entity resolution across heterogeneous sources with minimal manual intervention. This provides a scalable, maintainable alternative to procedural, hard-coded lookup logic.

Moreover, this process eliminates the risk of human error inherent in manual validation. In the procedural SQL lookup model, developers must explicitly define the logic for each field, along with managing hit/miss flags and subsequent triggers. This approach is highly prone to oversight: a single missed field in the lookup configuration, a logical error in flag handling, or an incorrectly mapped return value can allow corrupted data to bypass the integrity check undetected. By transitioning to DataHub, these manual failure points are removed; referential integrity is enforced natively by the platform's engine based on the model's structural definitions, ensuring that no record can evade validation due to configuration fatigue or procedural

oversights.
This architectural shift offers several technical advantages:

- **Reduced Development Overhead**: while the initial definition of Models for master and transactional tables requires a configuration overhead, it significantly adds long-term maintainability. By abstracting business logic from the Boomi Integration process into the Model definition, the system achieves greater modularity. Future modifications to validation rules or additional master tables are managed via model updates, eliminating the need for complex process restructuring
- **Schema Alignment and Automated Model Definition**: to streamline the configuration lifecycle and minimize manual entry errors, DataHub facilitates automated model generation via JSON profile ingestion. Since these profiles are natively produced by the Oracle Database Connector during the schema import phase, the synchronization between the physical database layer and the DataHub logical model is inherently maintained, ensuring structural consistency across the ecosystem

Enhanced Data Quality and Referential Integrity

DataHub enables fine-grained, field-level control exceeding what is typically achievable with standard database profiling mechanisms through:

- **Field-Level Constraints:** requirements such as mandatory fields, specific data types, and privacy-compliant data masking are enforced at the ingestion layer. This proactive enforcement prevents malformed data from reaching the database, avoiding generic exceptions
- **Automated Referential Integrity:** Reference Fields linked to Master Data models trigger automated integrity checks during Initial Load. Provided that data types and mappings are consistent, the Hub resolves foreign key relationships without requiring manual lookup logic
- **Custom Business Logic:** business rules are implemented intuitively through **Match Rules** (to identify duplicates) and **Data Quality Steps (DQS)**. Notably, the ability to invoke an entire Boomi Integration process as a DQS expands the scope of possible validations, allowing for complex, externalized logic

Observability and Active Remediation

A fundamental advantage of DataHub over "pure" Boomi Integration is the superior observability provided by the **Quarantine area**. While standard Boomi logs provide a process-centric view (success or failure of the execution), the Quarantine interface delivers a record-centric diagnostic. It provides an exhaustive breakdown of the specific metadata, validation rules, or referential links that triggered a failure, allowing for immediate identification of the root cause.

Furthermore, as mentioned, DataHub enables Active Remediation, allowing Data Stewards to manually correct and resubmit individual records. This eliminates the

need for full process re-testing or redundant executions, facilitating the resolution of transient issues and improving overall system throughput.

In this context, the architecture implicitly incorporates the **Recycle Error** logic previously discussed, transforming it into a native self-healing mechanism. By utilizing the Quarantine as a managed state of suspension, the system eliminates the need for a dedicated, manually-built recycling pipeline. Once a Data Steward remediates a record or the missing master data is synchronized, the Hub automatically re-evaluates the entity, ensuring that the recovery lifecycle is handled as a governed business workflow rather than a fragmented technical task.

8.3 Translation of the Integration Process

The transition from a standard Boomi Integration workflow to a DataHub-led architecture begins with aligning the physical Oracle Database schema with the DataHub logical layer. By designing Models that reflect the target tables, each relational attribute is faithfully represented within the Hub's canonical model, ensuring a smooth and consistent schema transition.

Beyond schema replication, this phase involves moving validation logic into DataHub.

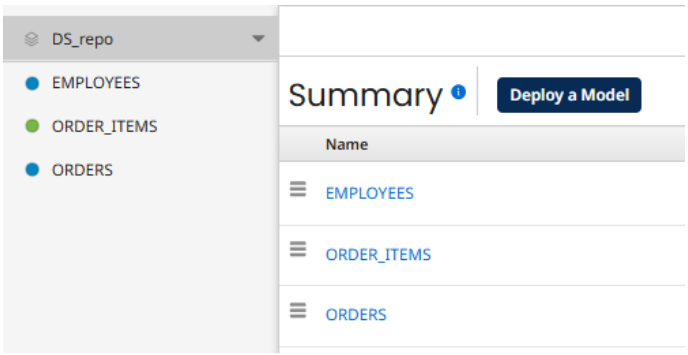


Figure 8.2: Logical structure of the Repository showing the deployed Models and their status

Business rules and quality checks, previously managed through Boomi Integration shapes, are re-defined as declarative constraints and native Data Quality Steps (DQS). This shifts data validation from the middleware processes directly into the repository, ensuring consistent rule application across every record.

It is essential to note that implementing DataHub does not eliminate the need for Boomi Integration; rather, it redefines its role as a transport and orchestration layer. In this implementation phase, only the L1 (OK) Stage logic is migrated to DataHub, and consequently, the previous workflow is significantly streamlined: the complex, multi-branched process, formerly characterized by extensive decision steps, business rules, and manual error-handling calls to the `E$` tables, is replaced by a lightweight, modular flow.

In this revised architecture, the Boomi Integration process is reduced to two primary execution paths:

1. **Environment Initialization**: managing the `TRUNCATE` operations on the staging tables
2. **Data Ingestion**: extracting the `DLT_ORDERS` from the Oracle connector, mapping the fields into the canonical format required by the DataHub Model, and dispatching the payload via the **Boomi DataHub Connector**

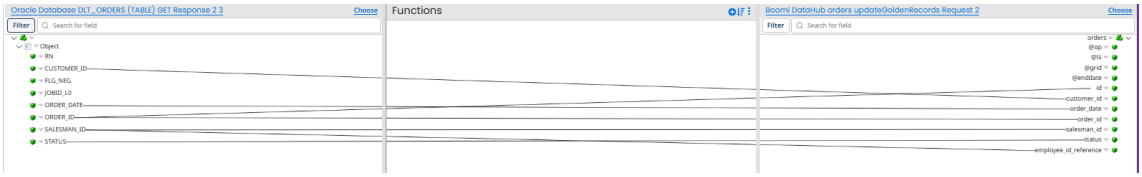


Figure 8.3: Mapping to each field of the `DLT_ORDERS` table to a DataHub compliant format

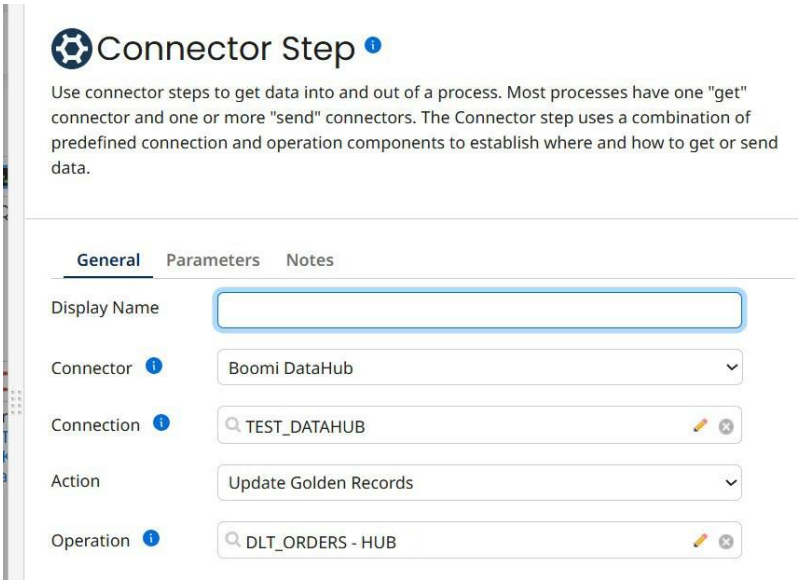


Figure 8.4: Base configuration of the DataHub connector in the Boomi Integration Canvas

By offloading validation and error persistence to the Hub, the integration process remains focused on data movement, while DataHub handles the governance, matching, and quarantine of non-compliant records.

A critical distinction between this MDM-centric architecture and the previously discussed refactored architecture lies in the requirement for a Preparation Stage. While the refactoring sought to eliminate intermediate preparation to minimize computational overhead, the DataHub implementation necessitates a dedicated stage to fully populate and promote Master Data models (e.g., Employees, Customers) to Golden Record status before ingesting transactional data (e.g., Orders). This is due to DataHub's automated referential integrity checks, which are not performed against a live external database, but against its own internal repository of established entities. Consequently, the preparation stage serves as a prerequisite to guarantee that the reference models are "known" to the Hub, preventing systemic quarantine of transactional records due to unresolved foreign key lookups.

8.3.1 Models

To maintain continuity with the previous chapters, the same dataset is utilized, focusing specifically on the `EMPLOYEES` and `ORDERS` tables to showcase the proposed solution; this approach facilitates a direct comparison between the legacy procedural validation and the governed, model-driven framework provided by DataHub.

The initial phase of the implementation involves the definition of the Models that will be referenced throughout the entire process.

Boomi's import wizard enables direct injection of profiles automatically generated by the Oracle Database Connector into DataHub, streamlining the definition process and minimizing setup time.

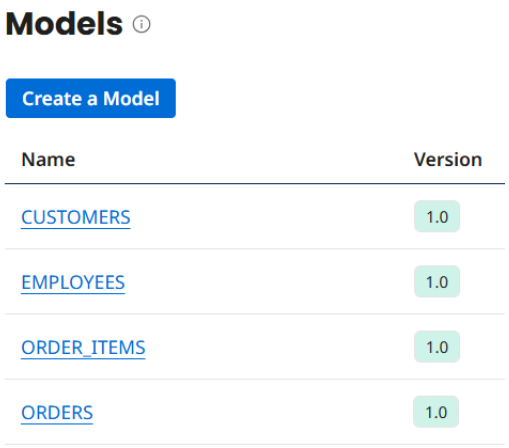


Figure 8.5: Definition of the Models to be referenced

To ensure reliable cross-reference creation and referential integrity, data ingestion follows a strictly defined sequence. Master Data entities must be fully synchronized and promoted to Golden Record status in DataHub before any transactional data, such as the records from the `DLT_ORDERS` table, is ingested.

As illustrated in Figure [8.6], the implementation includes a dedicated **Preparation Stage** to populate DataHub with the master records. This approach is critical: without a pre-established master dataset, transactional records (e.g., Orders) would be automatically quarantined during ingestion, as the Hub's engine cannot validate references against unresolved entities in the canonical repository.

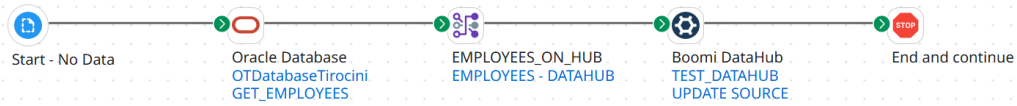


Figure 8.6: Preparation Stage in which the master data is injected into DataHub

By enforcing this loading hierarchy, the system ensures that every incoming transactional record can be immediately and automatically linked to its corresponding master record, fulfilling the core objective of a unified, MDM-led architecture.

Fields

Once the base schema is established, each Model is enriched with granular governance rules. Unlike standard database profiles, DataHub allows for the definition of advanced constraints that dictate how data is validated, stored, and protected:

- **Data Integrity**: fields are configured with strict criteria, including specific data type constraints and mandatory "Required" status, to ensure incoming records' completeness
- **Advanced Referential Integrity**: attributes representing foreign keys are converted into **Reference Fields**. This allows DataHub to inherently manage linkages between models, replacing manual lookup logic with automated cross-referencing
- **Privacy and Security**: sensitive attributes can be protected using **Data Masking**. By applying such policies, PII (Personally Identifiable Information) remains secured, ensuring the solution adheres to data privacy regulations while maintaining auditability
- **Field Validation**: custom rules, such as regex patterns or minimum/maximum length constraints, are applied to standardize data formats before they reach the Golden Record state

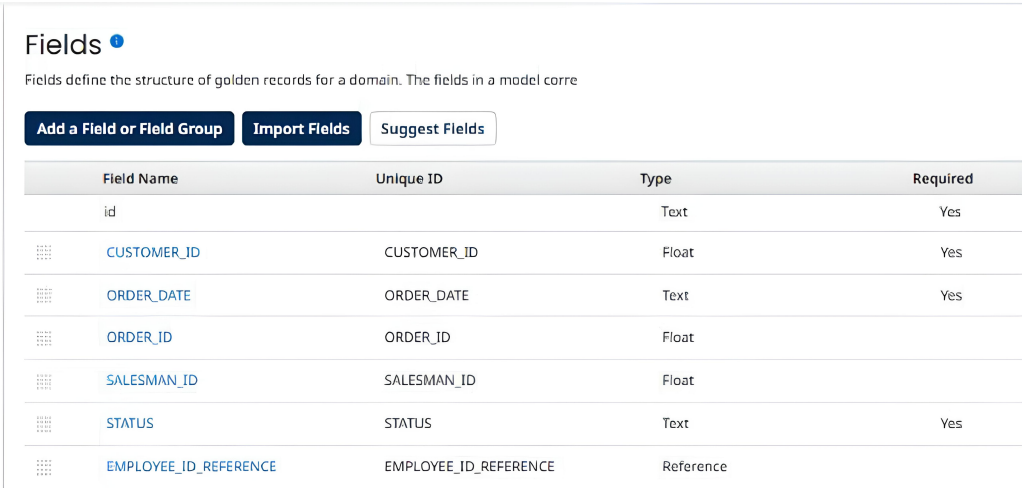


Figure 8.7: Representation of the `ORDERS` table in DataHub

8.3.2 Sources

The Sources are the external entities (e.g., applications, databases, file systems) that interact with the DataHub Model. In this architecture, a Source represents more than a simple connection; it establishes the governance framework for data contribution and synchronization. In the current implementation, the **Oracle Database** is configured as a primary source, acting both as a contributor of raw data (inbound) and a recipient of validated Golden Records (outbound).

Key configurations within the Source layer include:

Field Name	Unique ID	Type	Required
id		Text	Yes
EMAIL	EMAIL	Text	Yes
EMPLOYEE_ID	EMPLOYEE_ID	Float	
FIRST_NAME	FIRST_NAME	Text	Yes
HIRE_DATE	HIRE_DATE	Text	Yes
JOB_TITLE	JOB_TITLE	Text	Yes
LAST_NAME	LAST_NAME	Text	Yes
PHONE	PHONE	Text	Yes
MANAGER_ID	MANAGER_ID	Float	

Figure 8.8: Representation of the `EMPLOYEES` table in DataHub

- **Entity Attachment**: each source must be bound to a specific Model. This association defines the **Source Entity ID**, which is the unique identifier used by the external system to reference a record within the Hub
- **Contribution and Outbound Settings**:
  - **Contribute**: permissions that authorize the source to submit new records or updates to the Hub for processing
  - **Inbound Management**: rules governing how DataHub handles incoming payloads, including the enforcement of field-level permissions and validation logic
- **Source Ranking**: when multiple systems contribute data to the same Golden Record, DataHub employs Source Ranking to resolve conflicts. This ensures that the most authoritative system for a specific attribute prevails, guaranteeing high data integrity across the Hub
- **Channel Configuration**: channels define how updates to Golden Records are propagated back to the source systems. In this project, the channel is optimized to synchronize the Hub with the destination Oracle tables, ensuring that any manual remediation performed by a Data Steward is reflected downstream

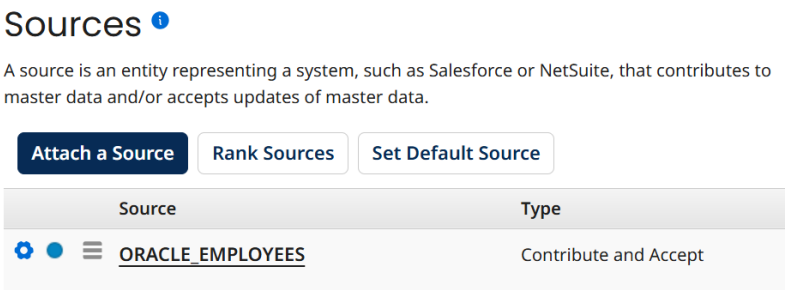


Figure 8.9: Source that contributes to master data

8.3.3 Quality Checks

To ensure that only high-quality data is promoted to a Golden Record state, a multi-layered validation framework is implemented within the model. This goes beyond basic data typing by enforcing semantic and logical correctness through two complementary mechanisms: Match Rules and Data Quality Steps.

DQS

The Data Quality phase serves as a gatekeeper, verifying that incoming data meets the required standards before further processing.

In particular, validation and enrichment logic can be implemented directly within DataHub through native Business Rules, provided that the requirements remain declarative and domain-centric. For more complex or computationally intensive scenarios, DataHub can instead invoke dedicated Boomi Integration processes, allowing advanced procedural logic to be applied externally while preserving centralized governance and traceability within the Hub.

As the requirements for this project were primarily domain-specific, the implementation was optimized using Business Rules. This approach not only ensures data integrity but also streamlines model maintenance by centralizing logic within the DataHub repository. Specifically, the following rules were established:

- **Temporal Consistency:** implemented across both `ORDERS` and `EMPLOYEES` models to ensure chronological coherence. By validating the `ORDER_DATE` and `HIRE_DATE` fields, the system prevents the ingestion of future-dated or illogical timestamps, thereby preserving the transactional integrity and business logic of the records
- **Consistency on `STATUS`**: within the `ORDERS` model, a rule was applied to restrict the `STATUS` field to a predefined set of three states, automatically quarantining any records that contain non-standard values
- **Consistency on `EMAIL`**: in the `EMPLOYEES` model, a rule checks the format of the `EMAIL` field, preventing the ingestion of non-compliant records

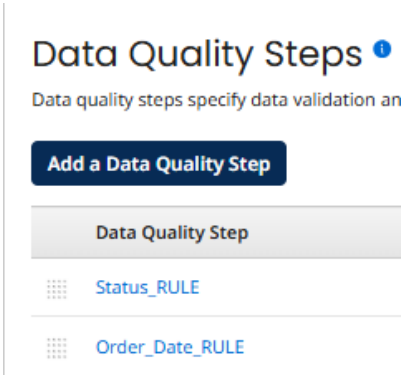


Figure 8.10: Data Quality Step on the `ORDERS` model

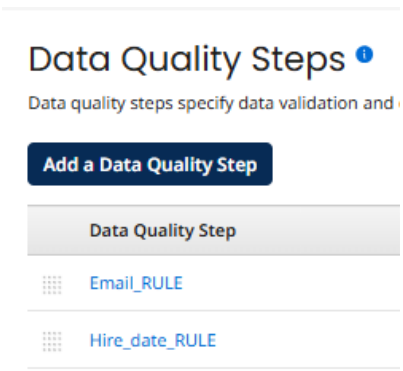


Figure 8.11: Data Quality Step on the `EMPLOYEES` model

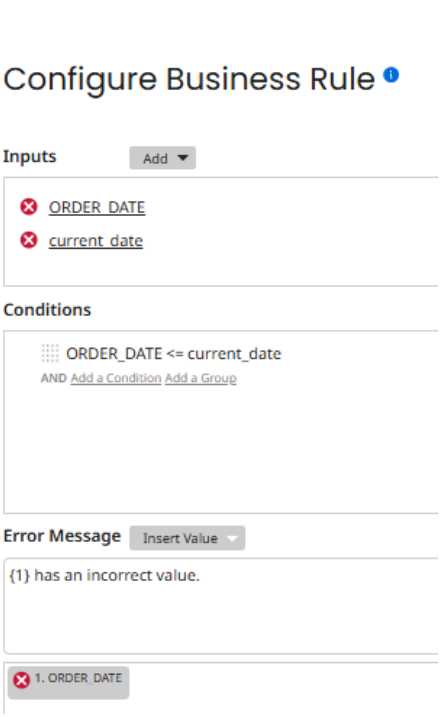


Figure 8.12: Business Rule on the date format and coherence

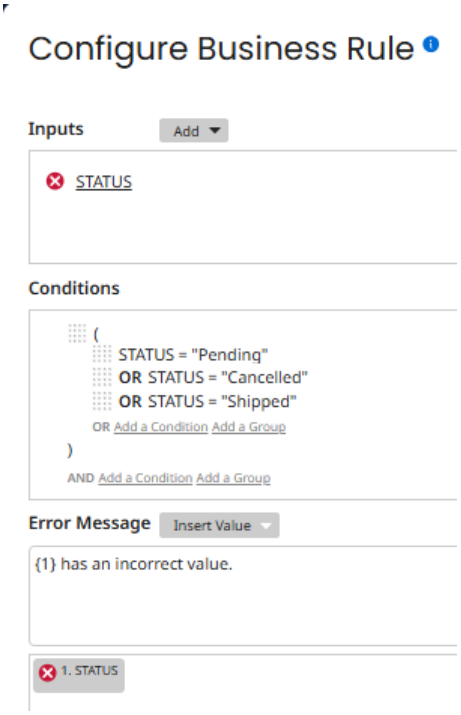


Figure 8.13: Business Rule on the accepted values for the Status field

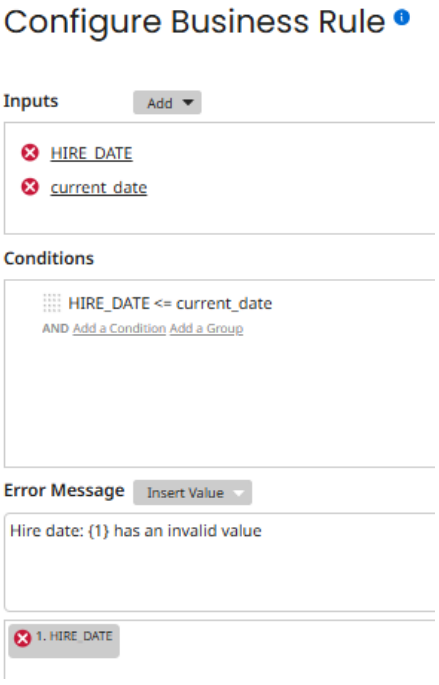


Figure 8.14: Business Rule on the hired date format and coherence

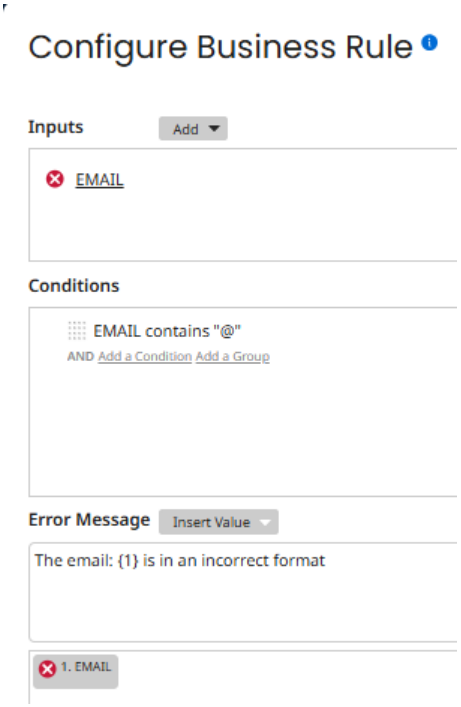


Figure 8.15: Business Rule on the email domain value

Match Rules

Once a record has passed all quality checks, DataHub applies Match Rules to determine its identity. The goal is to establish whether the incoming entity is a new entry or an update to an existing Golden Record, thereby ensuring a "single version of truth" is maintained.
The deduplication engine operates through a sequential identity resolution process, comparing inbound source data against the established pool of Golden Records. The system facilitates this through Simple Matches first, which rely on the exact equality of unique key attributes, such as an employee identifier.
To account for data entry inconsistencies or typos, DataHub can additionally employ Fuzzy Matching logic, identifying similar records based on probability thresholds and phonetic or distance-based algorithms. Furthermore, multiple rules can be organized into logical groups using `OR` operators, allowing the system to flag a duplicate if a record matches either a primary identifier or a specific combination of secondary attributes.

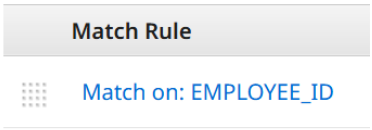


Figure 8.16: Match Rules on the `EMPLOYEES` model

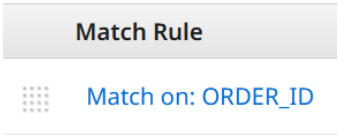


Figure 8.17: Match Rules on the `ORDERS` model

8.4 Stewardship

Data Stewardship represents the operational layer of the MDM governance framework, where manual intervention resolves data anomalies that automated logic cannot conclusively handle. In this architecture, stewardship centers on the **Quarantine**, a dedicated staging area that isolates non-compliant or ambiguous records to protect master repository quality.

Golden Records

A record is promoted to "Golden Record" status only after successfully clearing the entire governance pipeline, which includes schema validation, Data Quality Steps, and Match Rules.

In the developed implementation, the process begins with ingesting the `EMPLOYEES` dataset. As these records constitute the core master data, they are submitted to the DataHub repository and established as Golden Records before any transactional data is introduced.

Subsequently, once master records are consolidated, the workflow proceeds with the standard Boomi Integration process to populate the `DLT_ORDERS` staging table. These transactional records are then delivered to DataHub for business rules and validation logic application. As observed in the stewardship interface, every record that satisfies the defined governance conditions is successfully incorporated into the

Golden Record repository [8.18]. Each entry in the stewardship page provides a comprehensive view of the data, including:

- **Attributes**: a granular view of field-level data, in their normalized, canonical format
- **Created Date**: timestamp of the initial ingestion into the Hub
- **Updated Date**: timestamp of the most recent synchronization or delta update

This structured approach ensures that the `ORDERS` records can dynamically reference the already-established `EMPLOYEES` Golden Records, creating a fully governed and interconnected data ecosystem.

Updated Date	CUSTOMER_ID	ORDER_DATE	ORDER_ID	SALESMAN_ID	STATUS	EMPLOYEE_ID_REFERENCE
22 Dec 2025 12:07:30	46.0	2017-02-21 00:00:00	71.0	54.0	Shipped	2d41280f-8924-40e5-964d-3f8b2c0feb
22 Dec 2025 12:07:29	1.0	2016-11-17 00:00:00	105.0	54.0	Pending	2d41280f-8924-40e5-964d-3f8b2c0feb
22 Dec 2025 11:46:22	8.0	2017-02-14 00:00:00	8.0		Shipped	
22 Dec 2025 11:46:22	9.0	2017-02-14 00:00:00	9.0		Shipped	
22 Dec 2025 11:46:21	6.0	2015-04-09 00:00:00	6.0		Shipped	
22 Dec 2025 11:46:21	48.0	2016-02-17 00:00:00	73.0		Shipped	
22 Dec 2025 11:46:21	52.0	2017-02-19 00:00:00	37.0		Shipped	
22 Dec 2025 11:46:21	47.0	2016-11-29 00:00:00	13.0		Shipped	
22 Dec 2025 11:46:21	7.0	2017-02-15 00:00:00	7.0		Shipped	
22 Dec 2025 11:46:21	48.0	2017-03-07 00:00:00	33.0		Shipped	
22 Dec 2025 11:46:21	46.0	2016-11-29 00:00:00	12.0		Shipped	
22 Dec 2025 11:46:21	4.0	2015-04-26 00:00:00	2.0		Shipped	
22 Dec 2025 11:46:21	45.0	2016-11-29 00:00:00	11.0		Shipped	
22 Dec 2025 11:46:21	47.0	2017-03-09 00:00:00	32.0		Shipped	
22 Dec 2025 11:46:21	16.0	2017-02-10 00:00:00	75.0		Shipped	

Figure 8.18: Example of Golden Records for the `ORDERS` model that pass all checks and referential integrity

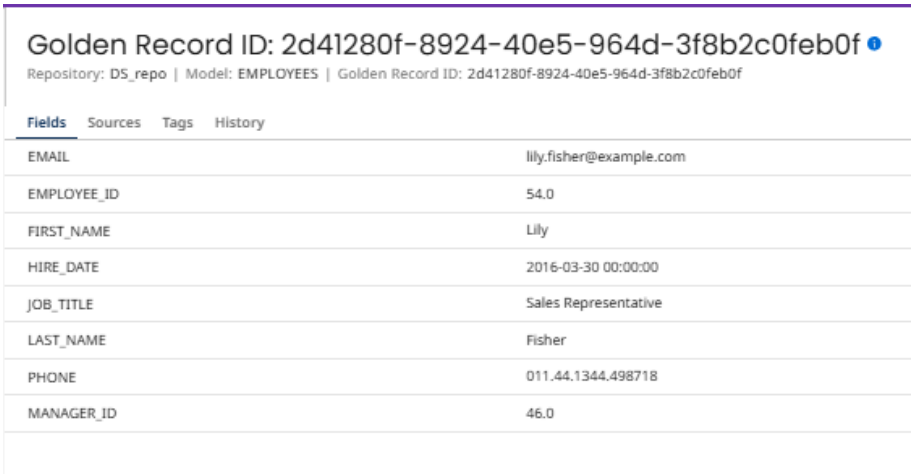


Figure 8.19: Detail on the Golden Record information

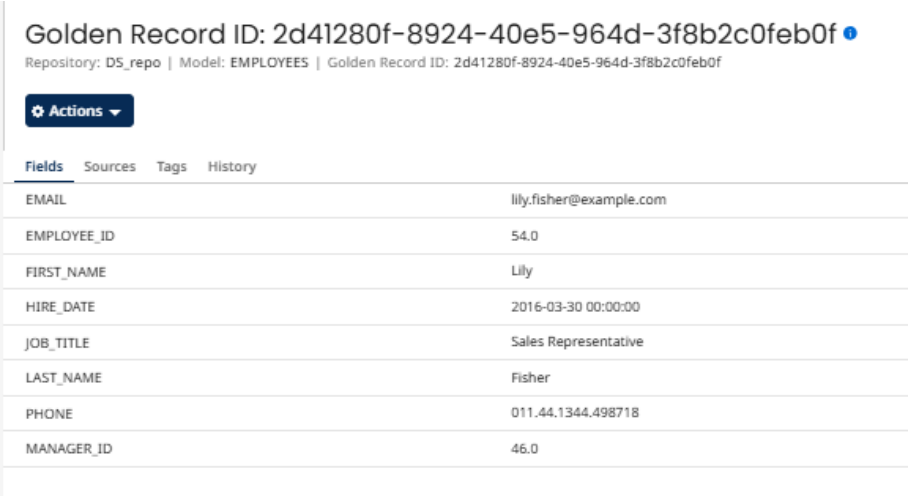


Figure 8.20: Example of Golden Records for the `EMPLOYEES` model

Quarantine

When a record enters the Quarantine, it is categorized based on the specific failure it encountered. The stewardship process involves analyzing and correcting these records through several standard procedures:

- **Data Quality Errors**: these encompass records that fail the previously defined **Business Rules** (e.g., an invalid `STATUS` or a malformed `HIRE_DATE`) are held here. A Data Steward can manually correct the erroneous values directly within the Hub interface or reject the record, prompting a correction in the source system
- **Matching Conflicts**: these arise when the **Match Rules** identify potential duplicates with a low confidence score or find multiple candidate matches but cannot definitively resolve their identity. Stewardship allows for a manual "Merge" or the creation of a "New Record" if the steward confirms the entities are distinct
- **Data Incorporation Errors**: this category identifies technical and structural anomalies, such as unmapped reference fields or schema mismatches, detected during the ingestion phase, for administrative review
  - **Unknown Reference Value**: occurs when a reference field points to a non-existent entity. The record is held in quarantine until the referenced master data is created, after which the steward can "Retry" the ingestion

As a demonstration, initial testing revealed that several records failed to achieve Golden Record status due to referential integrity violations; specifically, transactional records from the `DLT_ORDERS` table referenced employee identifiers that had not yet been established within the `EMPLOYEES` master model. While a standard procedural integration would typically result in database-level foreign key failures or the creation of orphaned data, DataHub proactively identifies these missing logical links. By isolating non-compliant records within the Stewardship layer, the platform preemptively safeguards the consistency of the master repository.

Simultaneously, the quarantine area captured records containing values that conflicted with the implemented business rules. Rather than being limited to a static error log, each entry in the quarantine is accompanied by a specific description of the cause of the failure. This operational transparency allows the Data Steward to immediately distinguish between systemic errors, which require correction on the source, and contextual anomalies that can be resolved through manual intervention or a conditional "Retry" once the missing master data is synchronized. Through this proactive approach, the quarantine is transformed from a simple repository for rejected data into a strategic tool for the continuous improvement of corporate data quality.

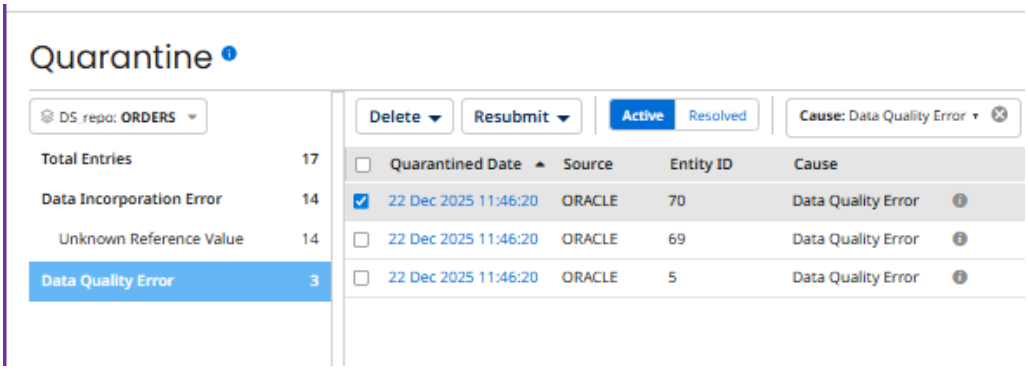


Figure 8.21: Examples of records quarantined for a DQE

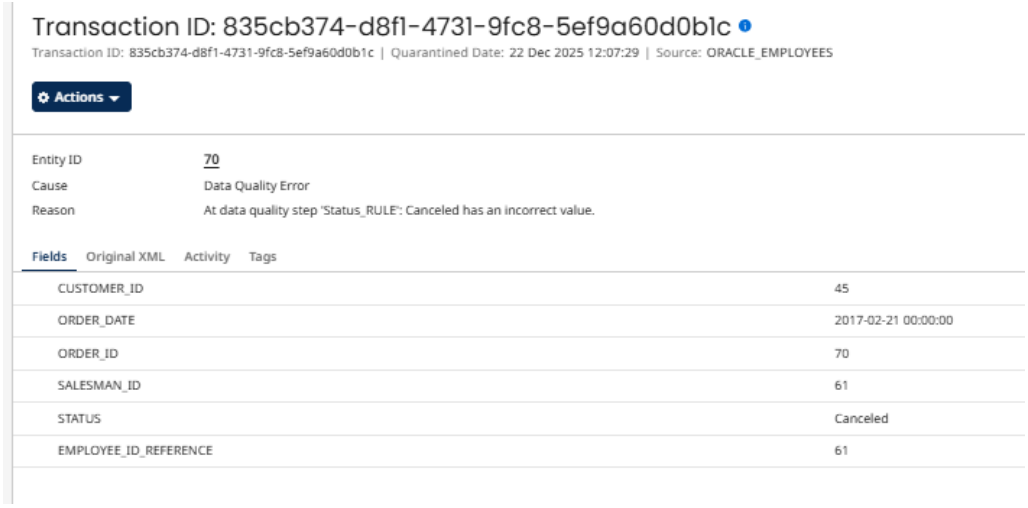


Figure 8.22: Detailed information on the quarantined record and the reason for the quarantine, namely the `STATUS` field

8.5 Activity Reporting and Monitoring

One of the key advantages of the DataHub architecture is its ability to deliver a centralized view of data processing and movement. The platform provides dedicated Inbound and Outbound Activity dashboards that enable granular tracking of data batches, offering a comprehensive audit trail that shows how information is ingested, validated, and synchronized.

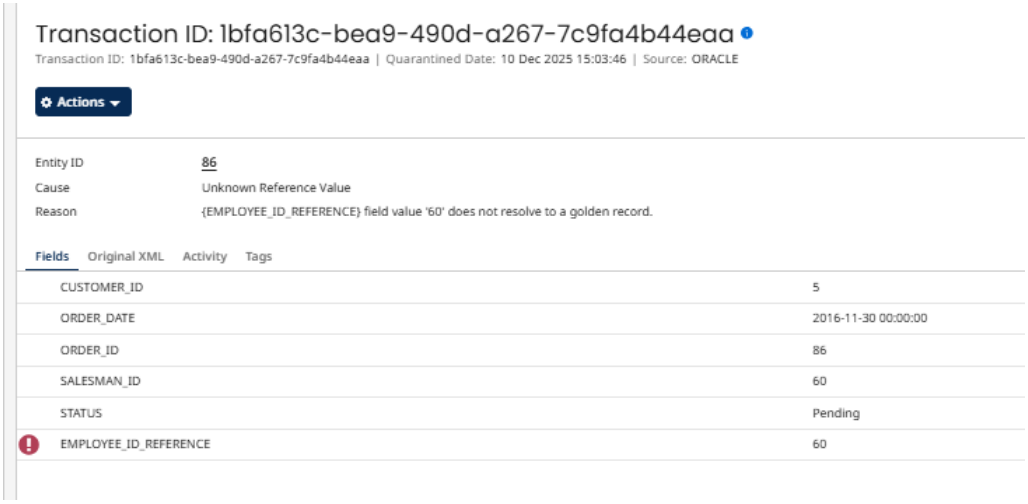


Figure 8.23: Detailed information on the quarantined record and the reason for the quarantine, namely the `EMPLOYEE_ID` referential integrity

Inbound Activity

The Inbound Activity section is the primary interface for monitoring data ingested into the Hub from the Oracle Database via the Boomi Connector. Each execution is logged as a distinct batch, enabling administrators to trace the exact ingestion timestamp and the corresponding data volume processed. The interface exposes critical metadata for operational oversight:

- **Report Time and Status**: indicated by a status icon (success or error), providing immediate visual feedback on the outcome of the ingestion
- **Traceability Links**: for batches submitted via Boomi Integration, the interface provides direct linkage to the specific execution in Process Reporting, allowing administrators to bridge the gap between middleware transport and MDM governance
- **Source Entity ID**: every inbound record is tracked via its original identifier from the Oracle source, ensuring bi-directional synchronization and reliable cross-referencing

The "Report Time" and "Repository" fields indicate the temporal context and the logical target of the execution, while the "Model" and "Source" columns identify the data domain and the originating system. This level of traceability is critical in multi-source architectures, where data from multiple databases must be consolidated and synchronized within a single Hub.

The most relevant and insightful capability is provided by the execution-level metrics. "Batch Count" and "Entities" represent the total volume of inbound data received, while the following metrics deliver a detailed breakdown of processing outcomes:

- Quarantined: the number of records that failed validation or matching rules and were routed to quarantine for manual review

- Created, Updated, and Deleted: the impact on the Hub, indicating how many Golden Records were newly created, how many existing records were updated, and how many were marked for deletion

Together, these indicators provide full visibility into inbound processing behavior, data quality enforcement, and the operational impact on the Hub repository.

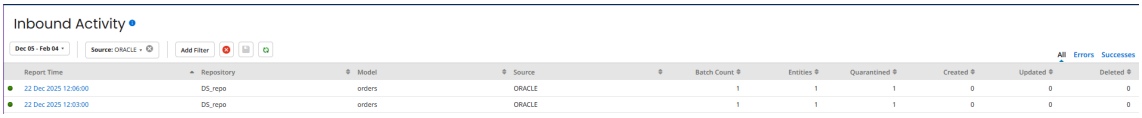


Figure 8.24: Inbound activity section of DataHub

Outbound Activity

Complementary to ingestion, the Outbound Activity section tracks the propagation of updates from the Hub back to downstream systems or other attached sources. This is particularly relevant for ensuring that any modifications performed by a Data Steward are successfully communicated back to the target environment.
To achieve this, DataHub utilizes Channels to manage the delivery of update requests. This ensures that the "Single Version of Truth" established within the Golden Records is synchronized across the entire enterprise architecture, closing the loop of the data management lifecycle through:

1. **Acknowledgement Mechanism**: DataHub tracks whether the target system has successfully received and committed the update. If a synchronization fails, the system provides a "Retry" mechanism, preventing data drift between the MDM and the database

2. **Stewardship Loopback**: one of the platform's core qualities is the ability to propagate manual corrections. When a Data Steward remediates a record in Quarantine, the resulting "Golden" version is automatically queued in the Outbound channel, ensuring that the manual improvement of data quality is synchronized enterprise-wide

8.6 Comparison

Moving from a standard Boomi Integration workflow to a DataHub-led architecture marks a significant change in how data integrity is enforced. In the procedural paradigm, validation logic is embedded directly in the integration pipeline. Leveraging SQL lookups that handle database-level verification, alongside custom mapping functions that manage complex branching logic, enables targeted dataset processing and allows specific business rules to be enforced through parameterized queries, therefore ensuring both precision and adaptability in ETL frameworks. Yet this efficiency comes at the expense of decentralization; validation rules remain "trapped" within individual processes, leading to maintenance challenges, most notably the need to manually update SQL statements for schema changes, and a reactive approach to error handling. In the legacy workflow, failures are either relegated to static Oracle staging tables (`E$`) or partially addressed through custom-built "recycle error" architectures which, although designed to reprocess failed records and automatically mitigate data loss, frequently introduce technical redundancy. Without a human-in-the-loop remediation layer, such mechanisms lack the governed oversight necessary to address the root cause of an anomaly, ultimately risking the automation of error persistence rather than facilitating its resolution.

In contrast, the MDM-led approach provided by Boomi DataHub shifts the focus from data movement to global governance. By defining rules at the Model level, the system ensures that every contributing source is subjected to the same quality standards, effectively centralizing the enforcement of a "Single Source of Truth". While this introduces a degree of architectural complexity and higher processing overhead compared to direct SQL statements, it fundamentally transforms the data lifecycle. The most significant shift is observed in the remediation phase: technical failures are no longer dead ends buried in log files, but are instead converted into governed business workflows within the Quarantine area, where automated cross-referencing replaces manual lookup shapes and a dedicated interface enables business users to resolve anomalies through a comprehensive set of stewardship actions.

Ultimately, the choice between these paradigms entails a trade-off between execution speed and the long-term value of data stewardship. While the standard integration flow is optimized for rapid execution through direct database connections, DataHub provides the persistent referential integrity and transparency required in complex domains, where the cost of a data error outweighs the latency introduced by the validation engine.

Metric	Procedural SQL Implementation	DataHub-Led MDM Architecture
Logic Structure	**Procedural**: fragmented validation via manual SQL Lookups per FK dependency	**Declarative**: centralized validation logic defined within Domain Models
Referential Integrity	Relies on manual lookup logic to verify master record existence via database-level queries	Natively enforced via automated Reference Fields and cross-model linkages at ingestion time
Error Management	**Passive**: failures are relegated to static Oracle error tables (`E$`) or addressed through automated recycling loops	**Active**: non-compliant records are routed to the Quarantine area for diagnostic analysis
Remediation	**None**: errors require manual database fixes or redundant, custom-built logic without native stewardship	**Integrated**: supports Active Remediation (Edit and Resubmit) by Data Stewards
Maintenance	**High**: requires manual updates to SQL queries and profiles for schema changes	**Low**: validation rules and constraints are updated centrally within the Hub Models
Execution Paradigm	**Synchronous**: direct JDBC connection to the Oracle source	**Asynchronous**: managed by the cloud-native Hub repository
Observability	**Fragmented**: visibility limited to standard Boomi execution logs and database tables	**Unified**: granular breakdown of batch outcomes (Created, Updated, Quarantined)

Table 8.1: Comparative Analysis of the `OK` Stage: Integration-led SQL Validation vs. MDM-led Governance.

Chapter 9

AI-Driven Development: Generative AI in the Boomi Ecosystem

The latest advancements in the Boomi platform introduce the use of **Autonomous Agents** to transition from manual configuration to intent-based development.
The inclusion of **Generative AI (GenAI)** in this project was motivated by the need to overcome the *Cold-Start* challenges inherent in complex ETL frameworks, where manual profile creation often becomes a bottleneck.

In this project, **Boomi Agents**, configured within **AgentStudio**, function as specialized entities that operate at the design-time phase, serving as a preliminary layer before the formal construction of the ETL process.

These agents represent a functional application of GenAI, where the technology is tasked with executing specific objectives as opposed to simple conversational interactions; in the context of the ETL pipeline, the agent acts as a metadata assistant, responsible for the initial synthesis of technical structures.

The employment of Boomi Agents provides several practical benefits during the initialization of the L0 and L1 layers:

- **Automated Metadata Synthesis:** the agent evaluates the requirements of the Oracle source tables to generate the corresponding profiles, ensuring they are structurally ready for the mapping phase
- **Contextual Profile Generation:** by leveraging GenAI, the agent analyzes the business context of the dataset to propose profiles with accurate data types and constraints, minimizing manual entry errors
- **Lifecycle Acceleration:** delegating the generation of technical artifacts to an autonomous agent streamlines the development workflow by providing the developer with a validated and structured metadata foundation, thereby enabling a more efficient transition toward the implementation of the core integration logic and transformation rules

While these agents facilitate the initialization phase of profile creation, the subsequent governance and complex relational logic remain under the control of the

Boomi Process Canvas, ensuring a balance between AI-driven speed and human architectural oversight.

9.1 Introduction to Generative AI

Recent developments in the field of Artificial Intelligence (AI) have enabled a significant shift in machine processing paradigms, moving from data-driven, discriminative tasks toward sophisticated generative capabilities. While traditional discriminative AI focuses on boundary determination (e.g., classification, regression, clustering) to make decisions based on existing data, Generative AI learns the inherent data structures to produce novel and realistic content. By leveraging deep generative models, this technology can synthesize high-quality outputs, including text, images, and programming code, based on basic user prompts [17].

The technological foundation of GenAI is rooted in Deep Learning (DL), an advanced subset of Machine Learning (ML) that employs artificial neural networks with multiple hidden layers to model complex data representations. Unlike early rule-based AI systems, DL models can autonomously detect patterns and correlations within high-dimensional datasets. These advancements have paved the way for Deep Generative Models (DGMs), which are trained to understand complex data distributions. The training process often utilizes semi-supervised learning, combining small amounts of labeled data with extensive unlabeled datasets to align the model's outputs with human intentions and values [18].

In the context of modern data integration, GenAI provides novel augmentation and automation prospects. As explored in this thesis, experimental implementations using platforms such as Boomi AgentStudio demonstrate how autonomous agents can automate the generation of high-quality, technical data from natural language inputs. This transition from manual configuration to intent-based development optimizes integration lifecycles and significantly reduces time-to-market by bridging the gap between business requirements and technical deployment.

9.2 Boomi AI

Boomi AI represents the native artificial intelligence layer embedded within the Boomi Enterprise Platform, designed to enable agent-driven transformation and understand, reason, and take action in dynamic environments [6].

The architecture of Boomi AI is built on a model of embedded intelligence that automates development through natural language prompts and the orchestration of specialized agents; this approach surpasses the linear limitations of traditional, deterministic automation by managing repetitive processes that require judgment, context-awareness, and adaptation to evolving scenarios.

9.2.1 Boomi Agents

To understand the mechanics of this layer, it is essential to define what constitutes an agent within the Boomi ecosystem.
"AI agents are autonomous or semi-autonomous software systems" that achieve

defined goals through a cycle of observation, reasoning, and action [19]. Every agent relies on several core components that dictate its behavior and reliability:

- **Goal:** a high-level statement that defines the agent's ultimate objective. Unlike traditional programming parameters, the goal provides the semantic context necessary for the agent to autonomously decompose a complex request into smaller, executable sub-tasks, aligning its reasoning with the desired business outcome
- **Model:** the core cognitive engine, typically a Large Language Model (LLM), that facilitates intent detection and natural language understanding. It serves as the reasoning layer that processes input data, maintains conversation state, and predicts the most logical next steps based on the underlying patterns identified during its extensive training on high-dimensional datasets
- **Instructions:** detailed natural language directives (System Prompts) that guide the LLM through complex task execution by supporting conditional reasoning and edge-case handling, effectively acting as a soft-coding layer that prevents unwanted behaviors or off-topic interactions
- **Tools:** functional extensions (e.g., API endpoints, database connectors, or custom scripts) that extend the agent's capabilities. By invoking these tools, the agent transitions from passive text generation to active system interaction, allowing it to retrieve real-time metadata or execute transactions within external enterprise ecosystems
- **Guardrails:** a critical safety and governance layer consisting of predefined filters and validation rules. Guardrails monitor both inbound prompts and outbound responses in real-time to mitigate risks such as *"hallucinations"*, ensure compliance with corporate data policies, and maintain the contextual integrity required for enterprise-grade applications

Agent-driven Workflows vs. Traditional Automation

Unlike deterministic automation based on rigid, syntactically fixed *"if-then"* rules, agent-driven workflows leverage the reasoning capabilities of the LLM to solve complex problems. Boomi AI distinguishes between **Pre-installed Agents** (such as Boomi GPT, DesignGen, and Scribe), which are ready-to-use for platform acceleration, and **Custom Agents** created via AgentStudio to address unique business needs. This duality allows organizations to maintain standardized productivity while fostering tailored innovations, effectively shifting the focus from manual configuration to strategic orchestration.

9.3 Boomi GPT

Boomi GPT represents a significant evolution within the Boomi Enterprise Platform, introducing a Generative AI–driven conversational layer that transforms how integrations, automations, and related tasks can be conceptualized and executed.

At its core, Boomi GPT is a conversational user interface (**CUI**), powered by generative AI, that interprets natural-language instructions to orchestrate AI agents across the full data management lifecycle, from integration development and master data governance to automated documentation and system optimization .

Unlike traditional low-code GUIs or manual configuration, users simply describe needs in everyday language; Boomi GPT translates them into functional platform elements [20].

By reducing the complexities associated with design, implementation, and documentation, this stage significantly accelerates time-to-market and represents a key architectural optimization. This shift aligns with broader industry trends that position GenAI not merely as a supporting technology, but as a procedural accelerator capable of translating business-level intent into executable integration artifacts. In doing so, it bridges the divide between conceptual strategy and technical deployment, minimizing the need for manual configuration.
Technically, Boomi GPT can orchestrate a collection of pre-installed AI agents, each embodying specialized functional capabilities [6]:

- Scribe: a documentation agent that automatically synthesizes technical manuals for both user-defined and AI-generated integration workflows. It is particularly effective for providing immediate structural clarity and maintaining standardized, audit-ready records of the integration logic

- DesignGen: an agent that synthesizes integration workflows by leveraging patterns and best practices derived from Boomi's extensive collection of integration schemas. By processing natural language prompts via Boomi GPT, the agent interprets the user's requirements to systematically generate a functional integration blueprint, which can be directly imported into the process canvas for further refinement

- Answers: a specialized knowledge agent designed to retrieve contextually relevant technical insights from both official product documentation and the Boomi Community. It provides immediate responses to architectural and configuration queries, streamlining the troubleshooting and research phases

- Pathfinder: a predictive tool that suggests the most logical subsequent steps during process design. By analyzing the current state of the integration on the canvas, it recommends optimal shapes or connectors based on historical patterns of successful workflow deployments

- HubGen: an AI assistant focused on accelerating the initial definition of master data models within Boomi DataHub. It interprets business requirements to synthesize draft models and synchronization rules, establishing a baseline for data governance and multi-system alignment

- Integration Advisor Agent: a tool that audits existing integration constructs to identify and recommend structural optimizations. Specifically, it evaluates naming convention compliance, suggests advanced error-handling frameworks, and proposes performance enhancements to ensure the workflow aligns with enterprise-grade best practices

Leveraging Boomi GPT, users interact with the AgentStudio interface within the Boomi Enterprise Platform by entering natural language prompts. These are interpreted to select the appropriate AI agents, producing suitable responses that may include executable artifacts, documentation, or suggested next steps.

Its value lies not only in the content it generates but also in its seamless integration within the platform ecosystem, where AI-driven designs can be instantiated directly within existing governance, lifecycle, and execution frameworks. By leveraging standard constructs (e.g., connectors, process canvases, and API definitions), it operates within architectural constraints while streamlining development processes.
This unified approach enables users, even those with limited integration experience, to create meaningful workflows, bridging the gap between business intent and technical implementation, and reducing the reliance on manual configuration.

9.3.1 Boomi Suggest

While Boomi GPT and custom agents focus on the automated synthesis of artifacts, **Boomi Suggest** represents the platform's predictive intelligence layer. It is an ML-driven feature specifically designed to accelerate the **Data Mapping** phase, which is often the most time-consuming part of integration development.

Boomi Suggest leverages *"crowdsourced intelligence"*, an anonymized, collective knowledge base derived from millions of successful mappings across the Boomi ecosystem. When a developer identifies a source and destination profile, Boomi Suggest analyzes the metadata and recommends the most probable field alignments with a confidence score. This mechanism significantly reduces manual effort by:

- Heuristic Matching: automatically pairing fields with similar semantic meanings, even if their technical names differ
- Function Recommendation: suggesting common transformation functions (like Date Formats or String Concatenations) that are frequently used in similar integration patterns
- Error Mitigation: minimizing the risk of human oversight during the configuration of large, complex profiles with hundreds of attributes

9.4 Motivations behind the AI-driven Optimization

The integration of Generative AI into the ETL framework is not a superficial adoption of emerging technology, but a deliberate architectural response to concrete procedural bottlenecks identified within the Boomi ecosystem. In standard development workflows, profile creation relies on a live connection to a database schema via the Boomi Import Wizard. However, this deterministic mechanism fails in environments where dedicated staging areas are absent or where security protocols restrict direct schema interrogation, a constraint that renders profile creation both time-consuming and prone to inconsistencies.

This scenario demonstrates the *Cold-Start* metadata challenge: the inability to bootstrap the integration lifecycle without a pre-existing, accessible data structure. By delegating profile generation to an autonomous agent, the framework addresses this bottleneck at its root, enabling a transition toward the *intent-based development* paradigm. In this model, high-level business requirements or SQL DDL statements are translated directly into validated technical artifacts, bypassing labor-intensive manual configurations and shifting the developer's focus from repetitive metadata management to strategic orchestration.

The architectural relevance of this paradigm extends beyond the interrogation of inaccessible schemas. It proves particularly valuable in schema-agnostic or API-driven contexts, where no physical data structures exist to import from, and technical artifacts must be synthesized directly from business-level specifications. In such environments, the agent functions as a virtual architectural layer, providing a schema discovery and synthesis foundation that would otherwise require manual, error-prone profile engineering.

By leveraging AI-synthesized profiles, the project introduces two advantages:

- **Significant Time Reduction and Lifecycle Acceleration:** having profiles pre-constructed by an autonomous agent eliminates the repetitive steps of the Import Wizard. This enables a faster time-to-market by allowing for an immediate transition to the core integration logic
- **Mitigation of Human Error and Structural Standardization:** the Agents ensure that technical structures remain standardized and consistent with the provided definitions, reducing the risk of downstream failures in the ETL pipeline stemming from manual oversight

Most importantly, these benefits are achieved without sacrificing systemic reliability. The inherent non-determinism of generative models, namely the risk of incorrect outputs or *hallucinations*, is governed through structured validation and Guardrails, ensuring that the framework remains production-ready and compliant with enterprise-grade architectural standards.

Ultimately, this shift redirects the development effort from manual, configuration-intensive metadata management to high-level architectural oversight. By bridging the gap between business intent and technical deployment, the AI-driven approach

transforms the Boomi Process Canvas into a more agile and metadata-driven environment, capable of evolving alongside complex enterprise requirements.

9.5 Boomi AgentStudio

AgentStudio serves as the comprehensive lifecycle management solution within the Boomi platform, providing the infrastructure necessary to design, govern, and deploy AI agents at scale. By integrating development, deployment, and governance into a single environment, it ensures that AI implementations remain compliant, secure, and consistent with organizational standards through built-in oversight mechanisms. Its structured architecture is organized into three primary functional pillars [21]:

1. **Agent Designer**: a low-code development workspace where both the technical and behavioral logic of an agent can be defined, enabling rapid prototyping and iterative refinement
2. **Agent Garden**: a centralized hub for tracking, deploying, and editing agents, which simplifies management and ensures operational continuity
3. **Agent Control Tower**: the governance and monitoring layer, providing visibility into agent invocations, performance, and outcomes, thereby facilitating compliance, auditing, and optimization

Together, these allow organizations to integrate AI agents seamlessly into their operational workflows, accelerating development, maintaining oversight, and ensuring that AI-driven processes are robust, scalable, and aligned with project objectives.

9.5.1 Agent Designer

Agent Designer serves as the primary Integrated Development Environment (IDE) within AgentStudio, offering a controlled workspace for defining an agent's behavior and iteratively refining its AI-driven logic. Beyond simple prompt engineering, it provides a sandbox where developers can build, test, and tune agents before deployment. The Designer also features a real-time chat interface (**Playground**) for reasoning validation, enabling immediate interrogation of the agent to verify that its outputs align with predefined goals and architectural constraints, while mitigating the risks of non-deterministic behavior.

Before promotion to the Agent Garden, each agent undergoes rigorous stress-testing within this environment. Developers can simulate edge cases, such as malformed data or missing parameters, to confirm that the agent's Guardrails correctly intercept and manage anomalies, ensuring robust and reliable performance

A distinctive feature of the Agent Designer is the ability to interrogate existing agents. This capability facilitates a form of *behavioral benchmarking*, whereby native Boomi agents serve as reference baselines against which custom agent outputs are systematically evaluated and refined, ensuring alignment with established enterprise best practices.

To accommodate varying degrees of complexity, the Designer supports multiple development pathways, allowing users to choose the most efficient entry point for agent creation:

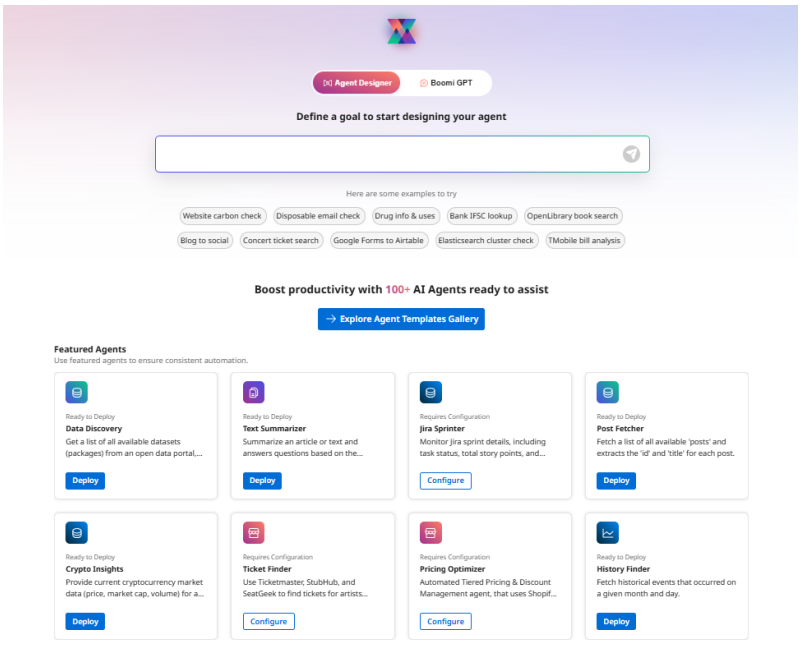


Figure 9.1: General configuration of AgentStudio

- **Starting from a Blank Template:** a granular approach for highly specialized agents, requiring the manual definition of identity and tasks
- **Building with AI:** the methodology selected for this project; it utilizes a *"prompting"* technique where the developer describes the desired agent in natural language, and the platform bootstraps the initial configuration
- **Importing and Templates:** developers can leverage over 100 industry-standard templates or import ready-made agents from the Garden to use as a starting point and structural foundation [21]

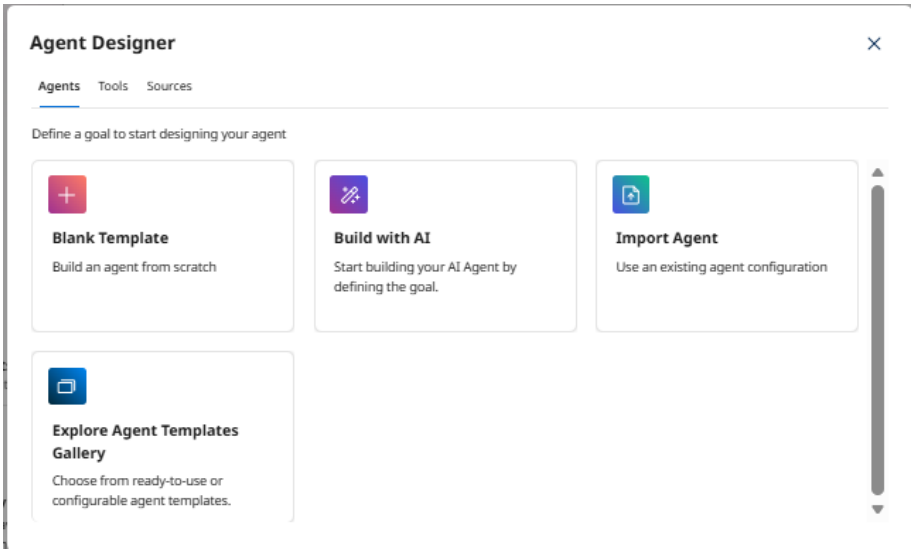


Figure 9.2: The different development paths to choose from

9.5.2 Agent Garden

The **Agent Garden** is the centralized environment where users interact with AI agents, serving as the *consumption layer* of the Boomi AI ecosystem. Here, they are deployed through a CUI powered by Boomi GPT.

In the context of this project, the Garden is the main interface for engaging with the profile-generation agent, allowing for the iterative refinement of technical assets through natural language dialogue.

The Garden acts as the administrative gateway for the agent lifecycle, enabling the exporting and importing of agents between different Boomi environments, ensuring that the logic developed for profile generation is both scalable and transferable. Furthermore, the Garden manages session continuity, allowing users to track multiple conversation threads or clear histories to reset the LLM's context [22].

Finally, once an agent is hosted there, it can be invoked as an **Agent Step** directly within the Process Canvas. This ensures that the agent's generative output, such as a validated JSON or XML profile, is immediately available for use in production-grade ETL processes.

9.5.3 Agent Control Tower

The **Agent Control Tower** represents the governance layer of AgentStudio, providing centralized oversight of the entire AI ecosystem. With the progressive expansion of AI agent adoption across organizational workflows, it serves as a control hub to manage security and privacy risks, ensuring that all interactions remain aligned with corporate compliance requirements and ethical guidelines.

The primary function of this module is to provide a unified interface for monitoring both native Boomi agents and those sourced from external providers. Through near real-time metric tracking, administrators can assess performance, operational health, and usage patterns, verifying that agents operate within acceptable latency thresholds. In the context of this project, the Control Tower proved essential for auditing agent activity, guaranteeing that generative tasks were executed securely and without unauthorized access to sensitive metadata.

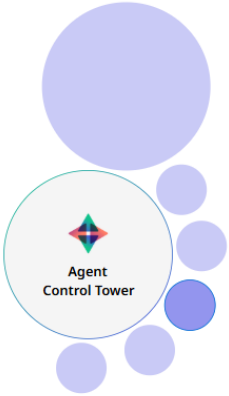


Figure 9.3: Control panel to keep track of all the Agents and their invocations

Beyond monitoring, the Control Tower facilitates "Session Tracing", a critical feature for troubleshooting and transparency. By tracing agent activity, developers can review detailed logs showing how prompts were interpreted and which tools were invoked. These insights feed directly into the refinement process within the Designer, enabling iterative improvements to agent logic and behavior. Furthermore, the Control Tower enforces the **Guardrails** defined during development at runtime, providing a final layer of defense against hallucinations or non-compliant outputs before they can impact the integration canvas.

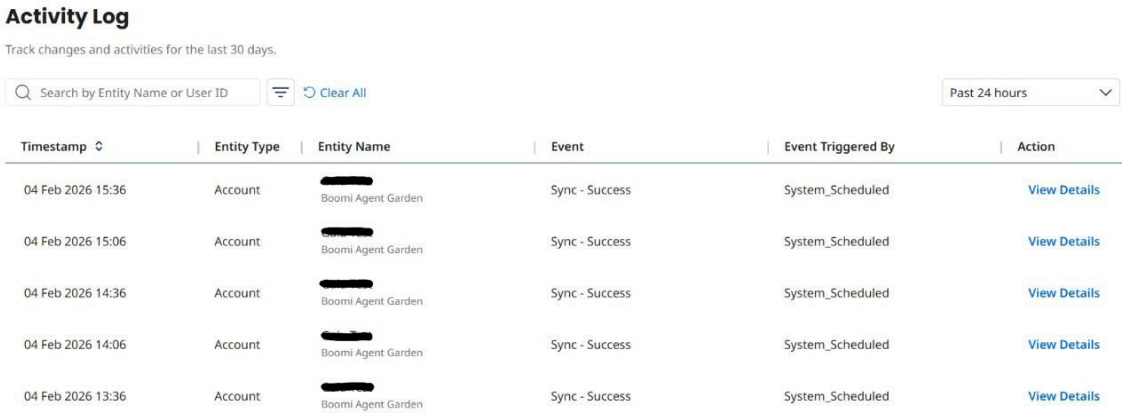


Figure 9.4: Agent Usage Logs with PII Removed

Building on this operational framework, the next chapter translates these capabilities into practice by detailing two agentic tools designed to address the challenges previously identified: a *layer-aware* pipeline for ETL-specific profile synthesis and a *format-aware* converter for rapid baseline generation. These solutions illustrate how the *intent-based development* paradigm can be refined to meet the rigorous demands of enterprise-grade integration workflows.

Chapter 10

AI-Driven Development: Agent Design, Implementation and Deployment

Building upon the theoretical foundations and platform capabilities established in the preceding chapter, this section shifts the focus toward practical implementation, detailing the design, development, and deployment of the two agentic tools, and tracing their architectural evolution. By examining both the *layer-aware* pipeline and the *format-aware* converter, we demonstrate how the *intent-based development* paradigm adapts to diverse and evolving integration requirements within the Boomi ecosystem.

10.1 Building the Agent

AgentStudio provides a versatile development environment offering multiple entry points for agent creation, from blank canvases and extensive libraries of ready-made agents to pre-configured templates. For the scope of this project, two distinct agentic tools were developed to address different profile generation requirements, with the second refined by the operational insights gained during the development of the first: a *layer-aware* pipeline for governed, ETL-specific metadata synthesis, and a *format-aware* converter for rapid, format-agnostic DDL-to-profile conversion. Both were implemented leveraging the *"Build with AI"* approach, which applies Prompt Engineering at the architectural level; by providing a high-level natural language prompt, the platform generates a functional baseline that serves as the foundation for the subsequent customization phase.

This customization spans several dimensions:

- **Core Identity and Mode**: beyond providing basic operational details, the agent's execution mode must be defined. It can operate in a Conversational mode for fluid interactions or a Structured mode, as detailed below [21]
  - **Conversational Mode**: it leverages the inherent fluidity of LLMs to facilitate an exploratory dialogue. It is particularly effective during the requirements gathering and discovery phase, where the agent assists the

developer in clarifying ambiguous field definitions, identifying potential relational constraints, and iteratively refining the business logic of a profile before it is formalized. In this configuration, the agent operates with a higher degree of linguistic freedom, prioritizing human-centric interaction over strict data formatting

  - **Structured Mode** (Schema-Driven): by defining a precise Input and Output Schema, the agent is constrained to produce a *predictable* structure, eliminating the ambiguity of natural language and ensuring that the output can be programmatically parsed and imported into the Boomi Process Canvas, via **Agent Step**

- **Personality and Interaction**: personality traits are configured to govern the tone and verbosity of responses, ensuring that the agent communicates technical outputs in a clear and contextually appropriate manner. Conversation Starters are defined to orient the user toward the agent's primary objectives from the outset, reducing uncertainty and improving the overall interaction efficiency in enterprise deployment scenarios

- **Task Orchestration**: the behavioral core resides in the Tasks definition, a structured set of natural language directives that govern the reasoning process with high technical precision. These instructions decompose the high-level goal into executable sub-tasks, define the functional boundaries, and determine the ability to invoke external **Tools** such as API endpoints or database connectors. Through this mechanism, the system transitions from passive text generation to active interaction, functioning as an orchestrator between the LLM's reasoning capabilities and external data environments

- **Governance and Guardrails**: to ensure safety and reliability, Guardrails are implemented to establish strict operational constraints. These parameters define the limits of the agent's autonomy, preventing unauthorized actions and ensuring that the output adheres to the predefined architectural standards. Beyond the default protections built into every agent, custom guardrails can be configured through denied topic definitions, word filters, and regex patterns to prevent sensitive data or off-topic subjects from being processed or returned, mitigating the risk of *hallucinations* altering the pipeline

Once the main logic has been finalized and validated through rigorous testing within the Designer's Playground, the agent is published to the Agent Garden. From there, it becomes accessible to authorized users across the organization or, as in the case of this project, available for invocation as an **Agent Step** directly within the Boomi Process Canvas at runtime.

10.1.1 Natural Language Interpreter

The first tool is optimized for production-grade ETL deployments requiring simultaneous profile generation across multiple layers (`STG`, `DLT`, `OK`, `ODS`), each enriched with layer-specific metadata and guaranteed structural compliance. While Boomi's Import Wizard offers deterministic profile generation via live database connectivity, it proves impractical in restricted environments or when manual construction across layers becomes the sole viable path due to schema access limitations.

To address this, a generative pipeline was developed that pairs a single agent invocation with a dedicated post-processing script. This architecture facilitates the transition from unstructured human intent to machine-ready data structures through two sequential phases: contextual interpretation and structural refinement.

Contextual Interpretation

The workflow begins with the user providing a `CREATE TABLE` statement alongside the ETL stage for which the profile should be created. In this initial phase, the agent acts as a **Schema Interpreter**, parsing the raw SQL to extract its structural components and synthesizing the specific technical attributes required by the destination.

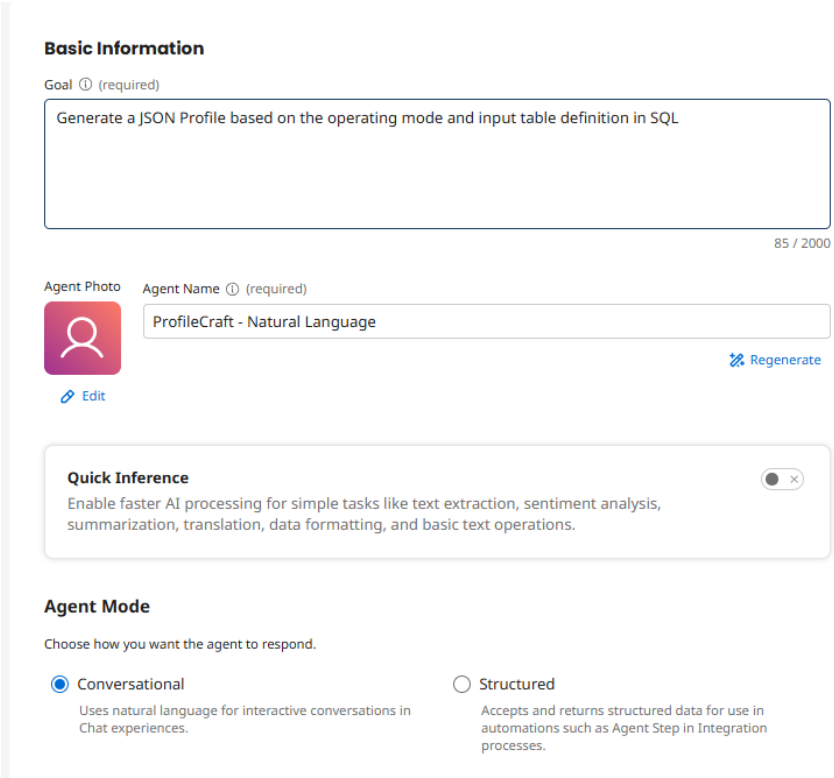


Figure 10.1: Natural language interpretation and goal configuration for the Schema Interpreter

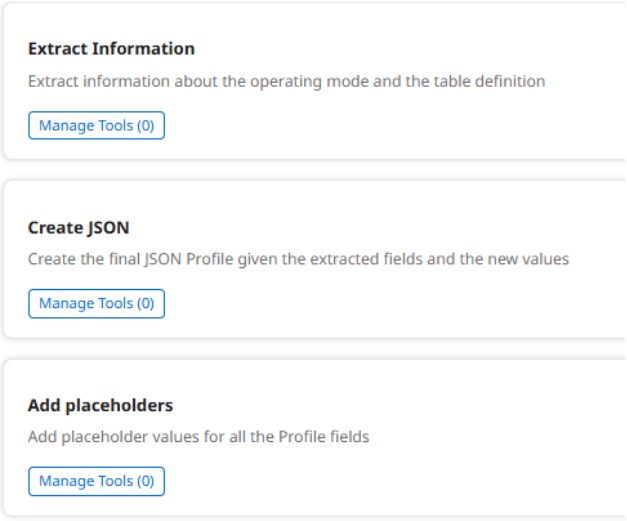


Figure 10.2: Detailed Task definitions for metadata extraction and layer-specific enrichment

Particular emphasis was placed on the **Task Orchestration** within this agent. To achieve successful Prompt Engineering, the instructions were designed with high technical granularity, focusing on three core operational objectives:

- **Information Extraction**: a deep parse of the SQL input is performed to identify column names, primary keys, and data types, ensuring that the relational logic is accurately captured before any transformation occurs
- **Stage-Specific Metadata Enrichment**: based on the target ETL layer, technical metadata is dynamically incorporated to tailor the output according to the business logic of the destination:
  - **STG (Staging)**: `JOBID` and `INS_TIME` are introduced to establish basic lineage for the initial data landing
  - **DLT (Delta)**: in addition to `JOBID` and `INS_TIME`, `FLG_NEG` is incorporated to manage record deletions. Since the `OK` validation stage shares an identical structure with the `DLT` layer, the output can be reused directly without generating a dedicated profile
  - **ODS (Operational Data Store)**: to support historical tracking, the base structure is extended with `JOBID_INS`, `INS_TIME`, `JOBID_UPD`, `UPD_TIME`, and `FLG_NEG`

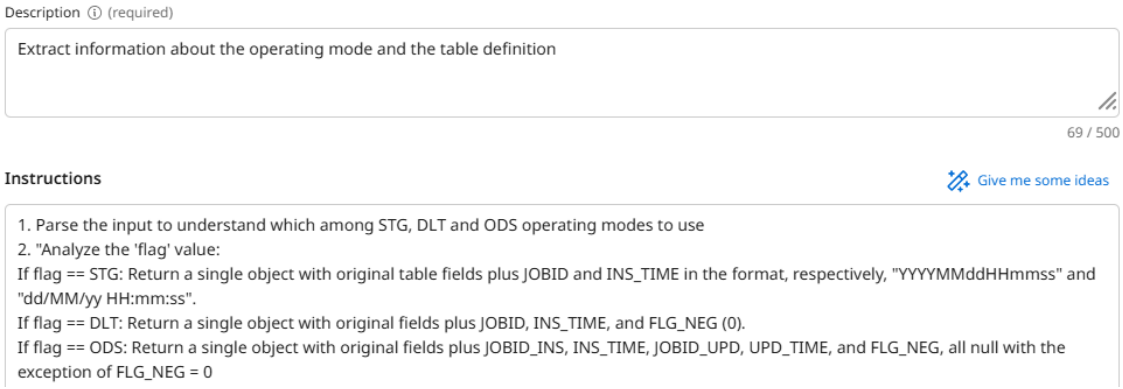


Figure 10.3: Specification of the multiple tasks the agent has to complete

Output Refinement

A significant challenge when utilizing standard LLMs is their inherent tendency to produce *verbose* responses. The core technical payload is often surrounded by natural language fillers or explanatory prose that, while beneficial for human interaction, is fundamentally incompatible with the **Boomi Process Canvas**, where any non-standard character would lead to a parsing failure during the import phase.
To address this, a dedicated post-processing script is employed within the Boomi integration process to sanitize the agent's response. The script strips any conversational noise and Markdown formatting from the raw LLM output, isolating the JSON payload and ensuring its syntactic integrity before it enters the pipeline.

The resulting JSON document is validated against Boomi's structural requirements, ensuring that every generated attribute, data type, and layer-specific metadata field is fully compliant before the profile is made available for deployment on the Process Canvas.

Dual-Agent approach An initial implementation explored a dual-agent architecture, pairing a conversational **Schema Interpreter** with a second agent operating in **Structured Mode**. By enforcing an Output Schema, this second agent was designed to constrain the LLM's response to a deterministic JSON payload, eliminating the ambiguity of natural language and ensuring that the output could be programmatically parsed and imported into the Boomi Process Canvas without manual post-processing.

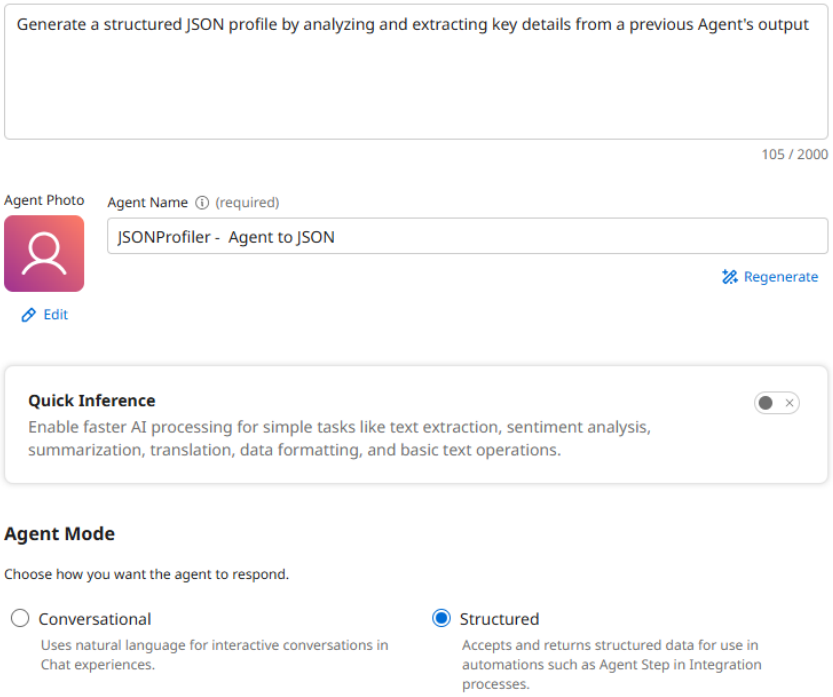


Figure 10.4: Profile Structurer definition and goal configuration

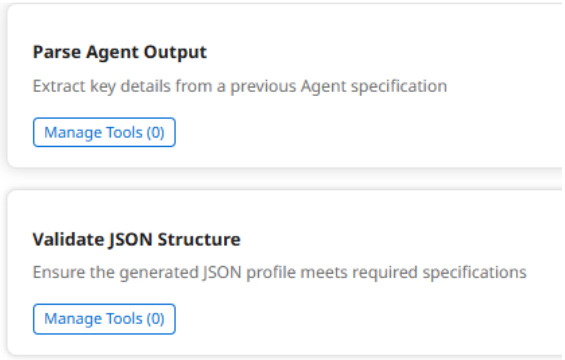


Figure 10.5: Tasks associated with the second agent

Although the input schema was deliberately permissive to accommodate any JSON document produced by the first agent, regardless of its verbosity or carried metadata, the Structured Mode mechanism guaranteed that the agent would return exclusively the JSON payload, without any conversational wrapper or accessory content surrounding it.

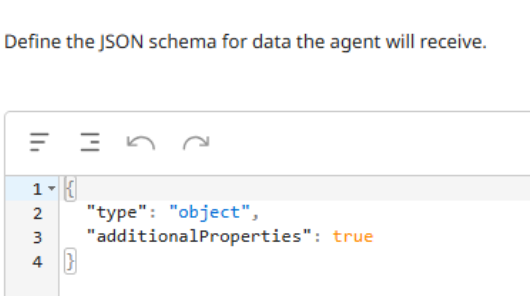


Figure 10.6: Input schema configuration

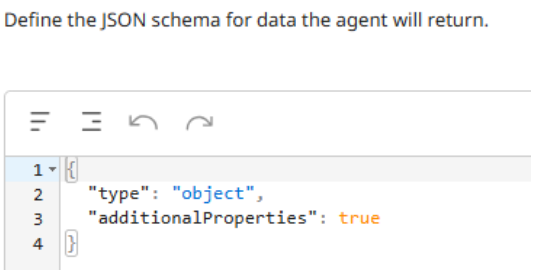


Figure 10.7: Output schema configuration

This dual-agent configuration therefore addressed the verbosity challenge at the architectural level, separating semantic interpretation from schema enforcement into two independent and evolvable layers. However, empirical testing during deployment revealed a concrete operational limitation. The computational overhead of chaining two LLM invocations proved disproportionate to the deterministic nature of the sanitization task performed by the second agent. Since output formatting does not inherently require the reasoning capabilities of an LLM, delegating it to a dedicated post-processing script yielded equivalent structural compliance at a significantly lower execution cost, while also eliminating the latency introduced by a secondary model invocation.

Figure 10.8: Sub-process that handles the logic behind the two Agent calls

The refined single-agent architecture was therefore adopted as the production implementation, retaining the full semantic enrichment logic of the Schema Interpreter while replacing the Profile Structurer with a more efficient deterministic alternative.

10.1.2 Multi-format Profile Generator

The development of the first pipeline revealed a recurring operational pattern where the combination of a single agent invocation with a deterministic refining script consistently produced reliable, production-ready output. This hybrid methodology was subsequently adopted as the architectural baseline for the second tool to ensure consistency across the entire framework.
While the *layer-aware* pipeline remains the appropriate solution for governed ETL deployments, its application to simpler conversion scenarios highlighted a significant contextual limitation. Notably, when layer-specific metadata enrichment is not required, the branching logic and three-fold execution of the generation process introduce an overhead that is disproportionate to the complexity of the request. The second tool addresses this issue by applying the same agent-plus-script principle to a broader and more flexible requirement, facilitating the rapid and *format-agnostic* conversion of any `CREATE TABLE` statement into a clean baseline profile in JSON or XML format.

This agent is therefore designed as a standalone entity that accepts both the SQL

input and the desired output format as parameters, producing a single accurate baseline profile through a *one-to-one* translation approach. By decoupling the generation logic from ETL-stage specificities, the tool eliminates redundant LLM invocations and synthesizes a structurally consistent output that can be immediately ingested into the workflow or manually tailored for specialized use cases.

The post-processing script employed in this pipeline fulfills a purpose comparable to that of the first tool by stripping conversational noise and Markdown formatting from the raw agent response. However, it operates on a broader output surface to handle both JSON and XML structures according to the requested format. This reuse of the structural refinement pattern across both tools confirms its validity as a generalizable architectural component within the *intent-based development* framework.

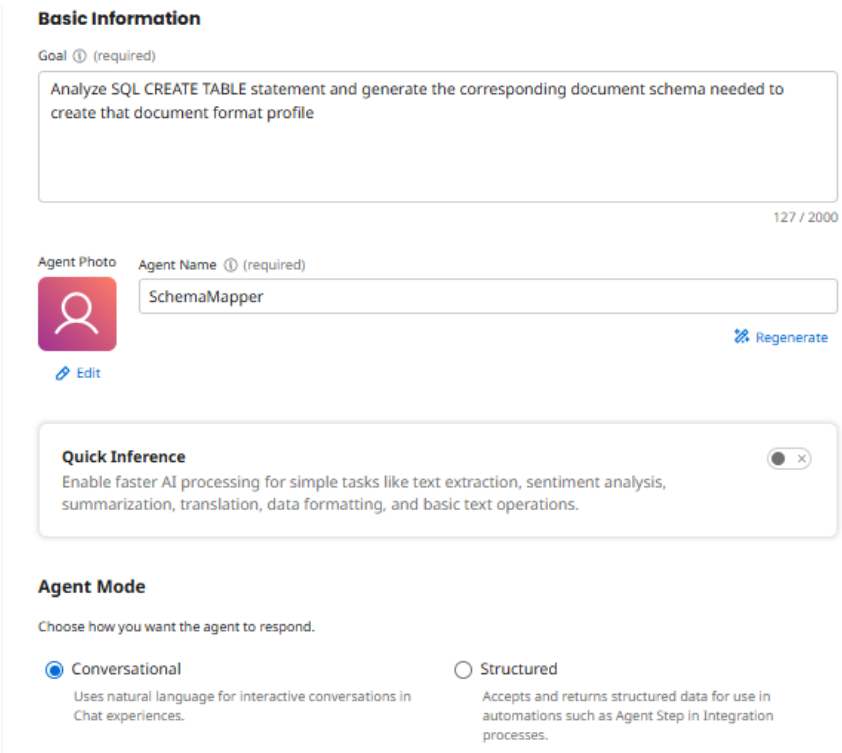


Figure 10.9: Profile Generator definition and goal configuration.

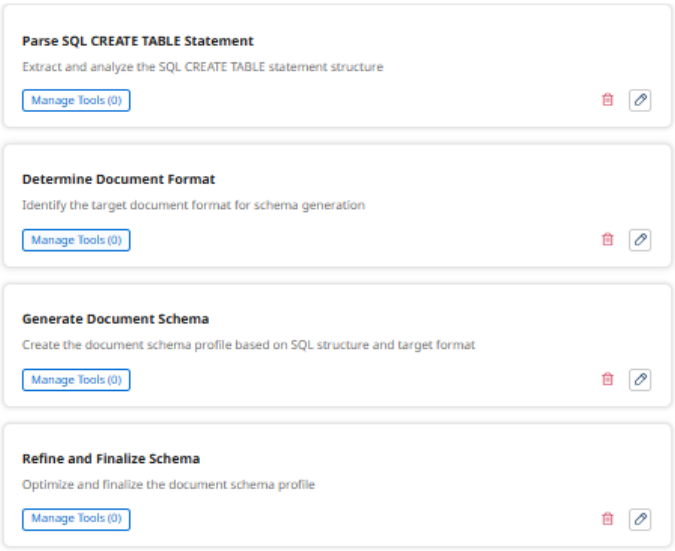


Figure 10.10: Detailed Task definitions for the format-agnostic Profile Generator.

By reducing the generative logic to its essential core, this agent proves particularly suited for rapid prototyping scenarios and ad-hoc integration tasks, where the overhead of a layer-aware pipeline would be disproportionate to the complexity of the request.

10.2 Development and Deployment in Boomi

The transition from agent design to operational deployment involves the creation of specialized orchestration flows within the **Boomi Integration Platform**, each reflecting the architectural principles of the corresponding agentic tool and translating them into governed, production-ready workflows. Critically, these two patterns were not conceived in isolation but developed iteratively, with the second directly informed by the operational constraints identified during the deployment of the first.

10.2.1 Orchestration for ETL Profile Generation

This orchestration pattern is designed for the standardized automation of the ETL layers. The process structures the workflow into three primary phases:

- **Iterative Logic**: a main process receives the `CREATE TABLE` statement and triggers a branch that invokes a sub-process three times, once for each layer (`STG`, `DLT`, and `ODS`)
- **Agent Invocation and Sanitization**: within the sub-process, the Schema Interpreter parses the SQL input and enriches the output with the stage-specific fields (e.g., `JOBID`, `INS_TIME`). The response is then processed by the downstream script, which strips conversational noise and ensures structural compliance before the profile is persisted
- **Collection and Delivery**: the three generated profiles are stored in a **Document Cache**. At the end of the execution, a **Data Process** shape aggregates them into a ZIP archive, which is then dispatched via the **Mail Connector**

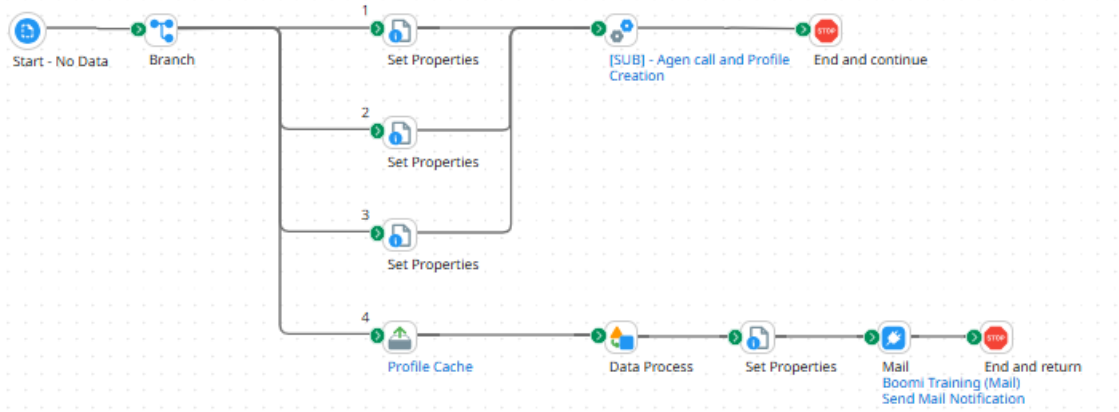


Figure 10.11: Boomi Integration Process for ETL profile generation

Figure 10.12: Sub-process handling agent invocation and sanitization logic

Orchestration Layer

The main process utilizes a branching logic where each path represents a specific ETL stage. To manage this, a series of **Set Properties Shapes** is employed to dynamically assign the `OPERATING_MODE` Process Property to one of the target values.

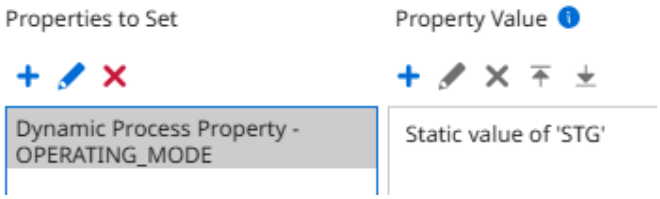


Figure 10.13: Set Properties Shape with the definition of the `OPERATING_MODE`

Once the variable is defined, the main workflow invokes a synchronized sub-process hosting the agent. This *context-aware* invocation allows the Schema Interpreter to retrieve the current value of the property and adjust its Task Orchestration accordingly, incorporating the specific metadata required for that particular layer. This architectural choice promotes high reusability, as the core generative logic remains centralized in a single sub-process while its behavior is driven by the dynamic property of the branch.

After the sub-process completes its execution for each branch, the resulting JSON profiles are persisted within a Document Cache using a shared **Batch Count** identifier. Once all three branches have finished, the final path of the process handles the retrieval and delivery of the technical artifacts. A **Data Process Shape** is utilized to query the cache via the same batch count assigned during storage, retrieving all documents accumulated across the three branches in a single pass for the final ZIP packaging and delivery.

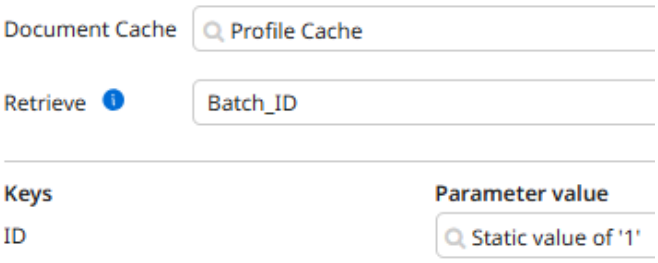


Figure 10.14: Extraction from the Profile Cache of all documents in the same batch

Following the retrieval, a **Set Properties Shape** is used to dynamically assemble the email metadata, mapping the process outputs to the message body and configuring the attachment parameters. The orchestration ends with the **Mail Connector**, which dispatches the notification to the end user. This automated delivery system ensures that the final profiles are received as a ready-to-use archive, effectively transforming the metadata setup from a manual, error-prone task into a governed and efficient AI-augmented workflow.

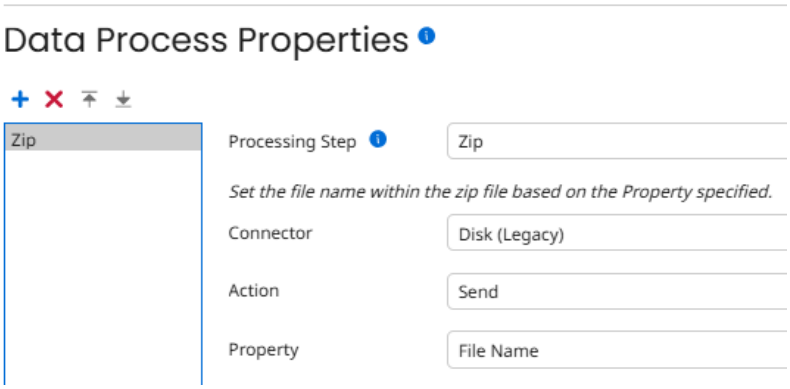


Figure 10.15: Data Process operation that provides a ZIP file with all the created JSON files

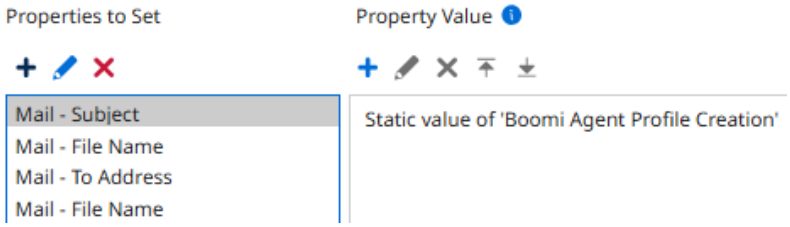


Figure 10.16: Dynamic construction of the email

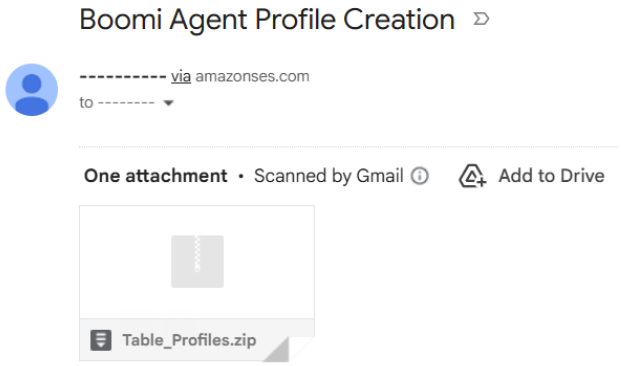


Figure 10.17: Example of resulting email, redacted for privacy purposes

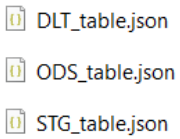


Figure 10.18: The resulting files after extracting them from the ZIP file

While the current orchestration utilizes a Mail Connector for the delivery of the generated profiles, the framework supports a more streamlined deployment pattern through the use of the **Disk Connector**. This approach allows the process to persist the technical artifacts directly into a local or network-mounted directory accessible by the Atom runtime. Bypassing the email-based delivery, the developer is no longer required to manually download, unzip, and import the profiles into the platform; instead, the generated files are immediately available on disk, enabling a direct *file-to-canvas* transition and further accelerating the overall development lifecycle.

Generative AI Implementation and Data Refining

The internal orchestration of the sub-process is designed to transform the raw SQL input into a validated, stage-specific JSON profile through a modular execution chain. The workflow begins by taking the `CREATE TABLE` statement, which remains constant throughout the execution, and utilizing a **Message Shape** to wrap it into a payload suitable for the agent's invocation.

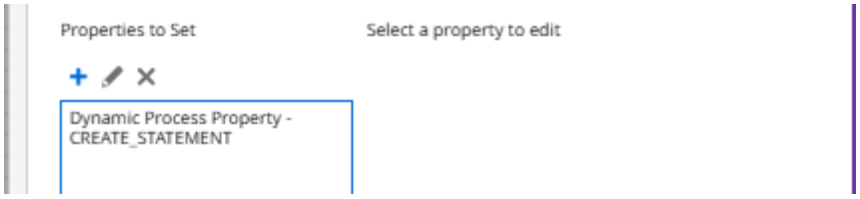


Figure 10.19: Set Properties Shape with the `CREATE TABLE` statement

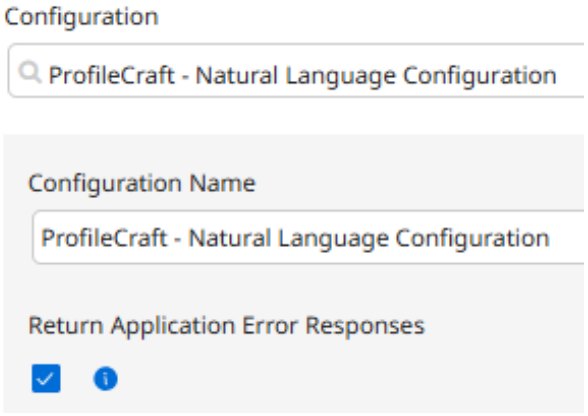


Figure 10.20: Definition of the Schema Interpreter Agent

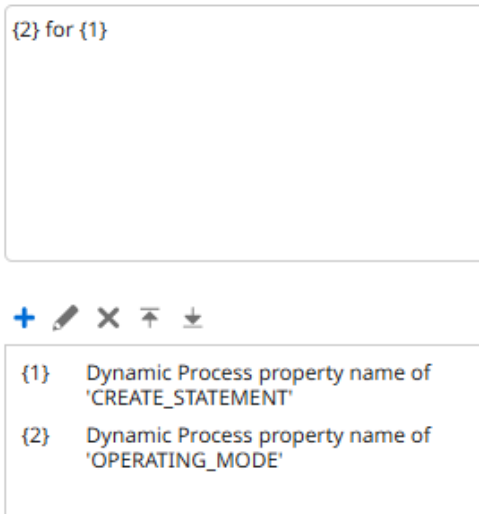


Figure 10.21: Message Step to construct the payload for the agent invocation

Once the agent has interpreted the SQL input and incorporated the context-specific metadata fields, its raw response enters a two-stage refinement pipeline before being persisted.

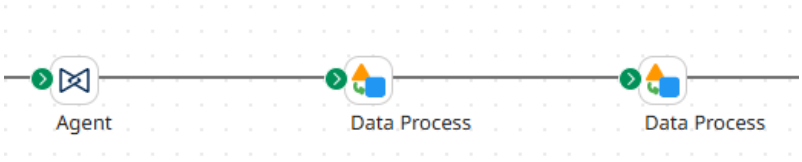


Figure 10.22: Close-up of the Process Canvas: the Agent Step followed by two consecutive Data Process shapes

The first **Data Process Shape** isolates the agent's response content from the accessory metadata returned by the Agent Step invocation, extracting only the core payload before any structural processing occurs. The second **Data Process Shape** then sanitizes the isolated payload by applying a custom Groovy script.
The clean JSON document is then prepared for storage through a **Set Properties Shape**, which leverages the `OPERATING_MODE` Dynamic Process Property to dynamically assign a descriptive file name and the appropriate extension, ensuring that each profile remains distinguishable within the cache.

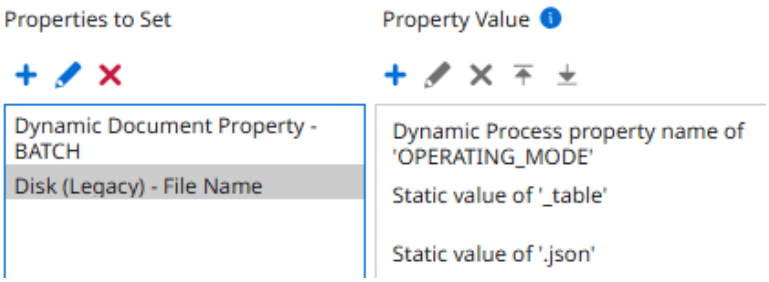


Figure 10.23: Dynamic construction of the document name and extension

The document is then stored in the **Document Cache**, using a Batch Count of 1 as the indexing key. By assigning a consistent batch identifier to each execution, the main process can later perform a comprehensive Retrieve from Cache operation, successfully gathering the `STG`, `DLT`, and `ODS` profiles in a single pass for the final ZIP packaging and delivery.

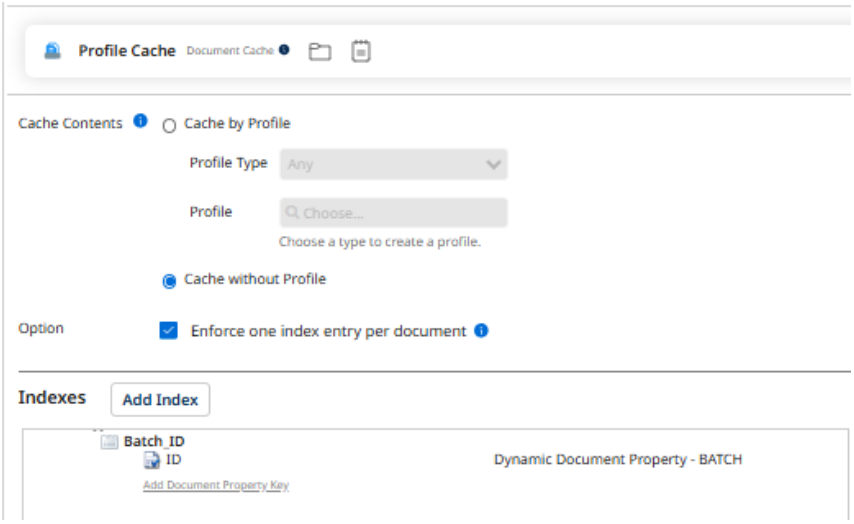


Figure 10.24: Definition of the cache component to store documents after each branch finished processing

Output Refining Script

The sanitization phase is implemented as the second **Data Process Shape** in the refinement pipeline, executed after the content isolation step has extracted the core agent response. Leveraging custom scripting, its purpose is to strip any conversational noise or Markdown formatting from the isolated LLM output, ensuring the JSON payload is syntactically clean before it is passed downstream for storage and delivery.

```
for( int i = 0; i < dataContext.getDataCount(); i++ ) {
    InputStream is = dataContext.getStream(i);
    Properties props = dataContext.getProperties(i);

    String content = is.getText("UTF-8");
    String unescaped = content.replace("\\n", "\n").replace("\\\"",
    "\"");

    Pattern p = Pattern.compile("(?s)(\\{.*\\})");
    Matcher m = p.matcher(unescaped);

    if (m.find()) {
        String finalJson = m.group(1);
        dataContext.storeStream(new ByteArrayInputStream(finalJson.
   getBytes("UTF-8")), props);
    } else {
        dataContext.storeStream(new ByteArrayInputStream(content.
   getBytes("UTF-8")), props);
    }
}
```

10.2.2 Orchestration for Single-Step Universal Conversion

The second orchestration pattern focuses on the **Universal Format Converter**. This flow was optimized for speed and flexibility, implementing a single-step agent invocation followed by a deterministic sanitization script, bypassing the layer-aware branching logic of the first pipeline in favor of a more direct procedural approach.

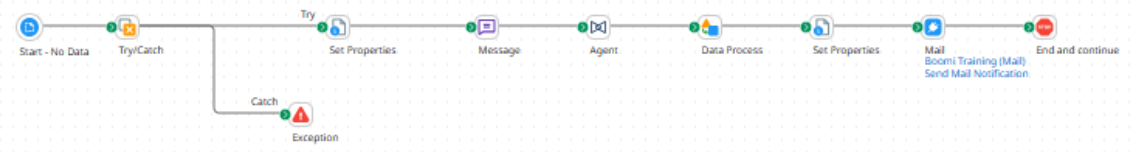


Figure 10.25: Boomi Integration Process flow for the Universal Format Converter

The workflow initiates with a **Set Properties Shape**, which captures the user's `CREATE TABLE` statement and the desired output format. These values are stored as dynamic process properties, ensuring that the configuration remains centralized. A subsequent **Message Shape** is then employed to encapsulate these properties into a structured payload, which serves as the input for the **Agent Step** invocation.

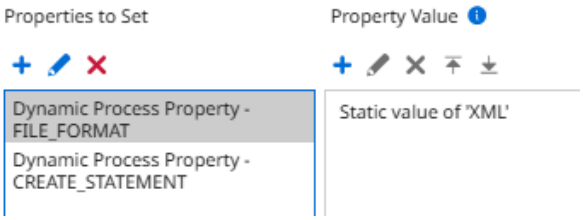


Figure 10.26: File output format and `CREATE TABLE` statement that will be the prompt for the agent

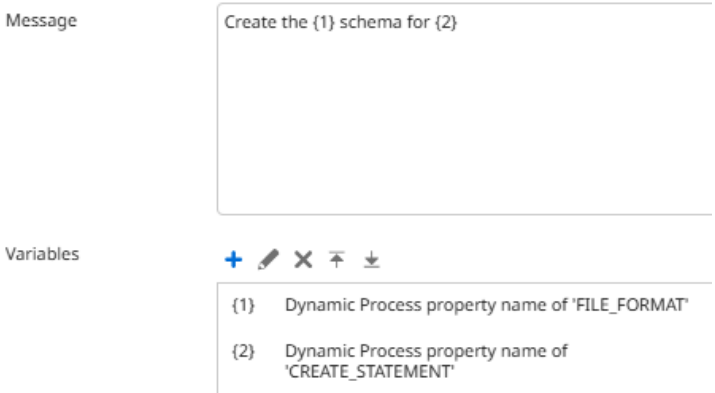


Figure 10.27: Message Shape to construct the actual prompt for the Agent

After the agent generates its raw response, a **Data Process Shape** with custom scripting extracts only the clean technical output, stripping conversational text and Markdown markers. This deterministic approach minimizes compute overhead while ensuring only relevant data flows downstream.

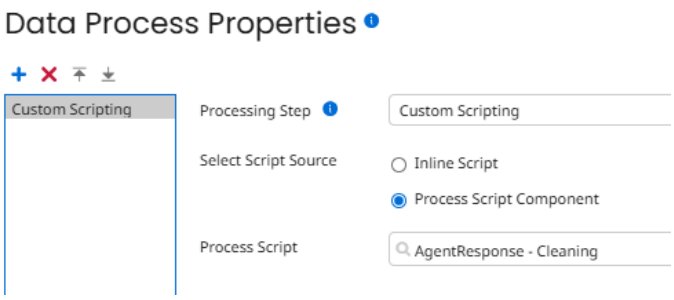


Figure 10.28: Data process step to sanitize the Agent response

The final stage of the process involves a **Set Properties Shape** that dynamically constructs the metadata for the output file. By referencing the format properties defined at the start of the execution, the shape assigns a descriptive **File Name** and the appropriate extension to the document. This prepared artifact is then routed directly to the **Mail Connector** for delivery.

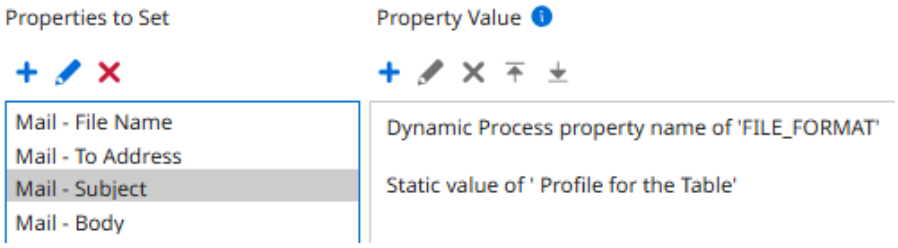


Figure 10.29: Set Properties Step to construct the mail attachment

By decoupling this utility from the specific ETL metadata logic, the orchestration achieves maximum efficiency. The user provides the SQL statement and selects the desired format, receiving a clean, baseline profile in seconds, ensuring a rapid time-to-market for general integration tasks where the complex layering of `STG` through `ODS` is not required.

The two implementations presented in this chapter are complementary tools designed for distinct integration scenarios. Both share the same foundational pattern of a single agent invocation followed by deterministic output sanitization, but diverge in their scope and enrichment logic; the selection between them is therefore dictated by the governance and complexity requirements of the target context.
Together, they demonstrate that *intent-based development* need not follow a single architectural pattern, but can be progressively refined and calibrated to the specific demands of each integration scenario, collectively fulfilling the objective introduced at the outset of this chapter: transforming profile creation from a configuration-intensive bottleneck into a scalable, AI-augmented workflow capable of evolving alongside complex enterprise requirements.

Conclusions

This thesis project, conducted at **Mediamente Consulting**, aimed to modernize complex data integration flows by transitioning from legacy, procedurally-driven Oracle environments to a cloud-native, low-code architecture on the **Boomi** platform. The primary scope was to design a framework capable of handling high-volume Master Data through a resilient `STG+DLT` pattern, ensuring that data synchronization remains scalable and performance-efficient. By standardizing the ETL phases, the project successfully delivered a modular infrastructure that decouples source extraction from business logic, providing the enterprise with an agile foundation for its evolving data ecosystem. The subsequent integration of Generative AI capabilities through **Boomi AgentStudio** extended this foundation beyond traditional integration paradigms, introducing an *intent-based development* layer that addresses structural bottlenecks at their root.

Analysis of Results

The implementation of the **Recycle Error** mechanism and the **STG+DLT** logic represents a significant advancement over traditional linear pipelines. The following key outcomes were achieved:

- **Operational Resilience**: the introduction of self-healing data flows reduced manual intervention by automatically re-evaluating records that failed due to transient referential integrity issues
- **Performance Optimization**: by shifting deduplication and enrichment logic to the database layer via SQL Analytic Functions, the computational load on the Boomi JVM was minimized, ensuring high throughput even during peak transaction volumes
- **Enhanced Governance**: the decoupling of technical system logs from human-readable diagnostic reports significantly shortened the remediation cycle for data stewards
- **AI-Driven Metadata Acceleration**: the deployment of two complementary agentic tools within AgentStudio demonstrated that the *Cold-Start* challenge inherent in ETL profile creation can be effectively addressed through *intent-based development*. The *layer-aware* pipeline reduced the initialization overhead of the `STG`, `DLT`, and `ODS` layers by automating the generation of structurally compliant JSON profiles from SQL DDL statements, while the *format-aware* converter extended this capability to ad-hoc, format-agnostic

conversion scenarios. Both implementations confirmed that deterministic post-processing scripts represent a more efficient sanitization alternative to secondary LLM invocations, establishing a generalizable architectural pattern for GenAI-augmented integration workflows

However, this transition highlighted a fundamental trade-off: the increased sophistication of the process canvas requires higher maintenance standards. The use of advanced SQL within the *Database Legacy Connector* provides higher precision but necessitates coordinated updates between the database schema and Boomi profiles to avoid technical debt. Similarly, the iterative refinement of agentic tools, from the initial dual-agent pipeline to the optimized single-agent architecture, underscores that GenAI-driven components require empirical validation and ongoing calibration to remain aligned with production-grade requirements.

Future Developments

The framework established in this project serves as a baseline for several high-value evolutions, whose relevance and priority are inherently context-dependent. The appropriate direction for future development is shaped by a combination of factors: the complexity and heterogeneity of the source systems involved, the governance requirements of the client organization, the volume and criticality of the data being processed, and the degree of standardization achievable across the integration landscape.

Two of these directions, namely the extension of the **AgentStudio** implementation and the full adoption of **Boomi DataHub**, are grounded in concrete explorations already conducted during this project, which established both the technical feasibility and the architectural compatibility of these evolutions with the existing framework. A third set of directions, outlined separately, identifies further strategic opportunities that emerge naturally from the architecture established here but were not explored within the scope of this work.

AI-Assisted Orchestration with AgentStudio

The experimental phase involving **Boomi AgentStudio** did not remain purely theoretical: two agentic tools were designed, implemented, and validated in a production-oriented context, establishing a concrete architectural pattern that proved generalizable beyond the specific use case of ETL profile generation. This foundation makes the extension of the *intent-based development* paradigm to additional phases of the integration lifecycle a well-grounded and technically viable evolution.

A natural extension of the *layer-aware* pipeline would be the automation of the **Data Mapping** phase, which currently represents one of the most time-consuming steps in the development workflow. While *Boomi Suggest* already provides crowd-sourced field alignment recommendations, an agentic approach could complement this with context-aware generative capabilities, interpreting the business semantics of source and destination profiles to propose complex transformations between heterogeneous systems. By leveraging the same agent-plus-script pattern validated during this project, such a tool could suggest field alignments and transformation

functions directly within the development workflow, providing the developer with a structured starting point that reduces manual effort and minimizes the risk of semantic alignment errors.

A second high-value evolution concerns **Dynamic Profile Maintenance**. As source systems evolve, schema drift represents a recurring source of technical debt, since database schema changes require coordinated updates to Boomi profiles, a process that is currently manual and error-prone. An agent operating continuously in the background could monitor source schema changes, compare them against existing profiles, and automatically propose or apply the required updates, reducing the coordination overhead between database evolution and integration assets.

Finally, the **Predictive Monitoring** of pipeline health represents a strategically significant application of GenAI within the operational phase. Rather than relying on reactive error handling, an anomaly detection agent could analyze execution logs and throughput patterns to identify emerging bottlenecks or data quality drifts before they propagate to target systems, transforming the governance model from corrective to proactive.

Master Data Hub Integration

A structurally significant evolution of the framework involves the full adoption of **Boomi DataHub** as the governing layer of the Master Data lifecycle. Within the scope of this project, DataHub was already explored as a concrete alternative to the procedural validation logic of the `OK` stage. The implementation confirmed that models could be effectively defined to ingest and promote master data to *Golden Record* status while enforcing referential integrity natively at ingestion. Furthermore, the Quarantine area proved to be a governed and scalable replacement for the custom-built Recycle Error mechanism. This initial exploration established the architectural compatibility of DataHub with the existing ETL framework and demonstrated tangible governance advantages over traditional SQL-based validation approaches.

A complete transition would extend this integration to the entire pipeline, transforming the current architecture into a centralized **Master Data Management (MDM)** solution. This evolution becomes particularly strategic in contexts where the client's data originates from multiple heterogeneous sources, where the same master entity may appear across different systems with conflicting or complementary information. In such contexts, a unified MDM layer becomes a functional necessity, other than an architectural improvement, because the complexity of cross-source deduplication and semantic reconciliation often exceeds what can be reliably managed through distributed pipeline logic alone.

DataHub addresses precisely this challenge through the definition of *Golden Records*, a single, authoritative representation of each master entity, reconciled across all source systems and propagated consistently throughout the information ecosystem. Furthermore, DataHub's native support for **match and merge** logic would allow the organization to define declarative data quality policies that operate independently of the underlying integration flows, ensuring that governance evolves alongside the data ecosystem without requiring structural changes to the ETL pipeline.

Combined with the AI-driven profile generation capabilities developed in this project, a DataHub-centered architecture would represent a fully governed, intent-driven Master Data platform capable of addressing the complexity of large-scale enterprise data synchronization.

Further Directions

Beyond the evolutions grounded in the empirical work conducted during this project, the architecture established here naturally opens a set of broader strategic directions. Their feasibility is supported by the modular and extensible nature of the Boomi platform and by the architectural principles that guided the design of the ETL pipeline.

Multi-Source Expansion

Although the framework was designed and validated in the context of Oracle-based source systems, its modular architecture is inherently suited for extension to a broader and more heterogeneous integration landscape. In enterprise environments, data rarely originates from a single, standardized source: ERP systems such as SAP, CRM platforms such as Salesforce, and custom REST or SOAP APIs frequently coexist within the same information ecosystem, each imposing distinct structural and connectivity requirements. The decoupling of extraction logic from transformation and loading phases, a core principle of the `STG+DLT+OK+ODS` pattern, provides a natural foundation for this expansion, as new source connectors can be introduced without altering the downstream pipeline architecture.

Furthermore, the AI-driven profile generation capabilities developed through AgentStudio are particularly valuable in this context: in scenarios where native connectors are unavailable or where source schemas are unstructured, the *intent-based development* paradigm eliminates the dependency on deterministic import mechanisms, allowing the framework to onboard new data sources with significantly reduced manual effort.

Real-Time and Event-Driven Processing

The current pipeline operates on a batch-oriented execution model, where data synchronization is triggered at scheduled intervals. While this approach is well-suited to the high-volume Master Data scenarios addressed in this project, it introduces an inherent latency between the moment a source system records a change and the moment that change becomes available in the target environment.

In use cases where near real-time data availability is a business requirement (e.g., operational reporting, cross-system inventory synchronization, time-sensitive compliance workflows), a transition toward an event-driven architecture would address this limitation by eliminating the staging phase from the critical path. Rather than accumulating records in the `STG` layer before processing, the pipeline would react directly to change events emitted by the source systems, applying transformation and validation logic in-flight and propagating the output to the target environment without intermediate buffering. This shift would substantially reduce end-to-end synchronization latency, bringing the pipeline closer to near real-time behavior. A

post-delivery staging phase could still be retained for logging, auditing, and lineage purposes, preserving the governance properties of the existing framework without reintroducing latency into the critical delivery path.

By integrating **Boomi Event Streams** [23] or an external message broker, this architectural evolution could be pursued without requiring a fundamental redesign of the transformation and enrichment logic already established, but rather a replacement of the ingestion layer with an event-aware trigger mechanism.

Observability and Advanced Monitoring

As the complexity and scope of the integration framework grow, the ability to maintain visibility into pipeline health, data quality trends, and operational performance becomes a critical governance requirement. The current architecture provides transactional logging and error reporting through dedicated diagnostic flows, but lacks a unified observability layer capable of aggregating and contextualizing metrics across the entire integration lifecycle.

A natural evolution would involve the introduction of a structured **monitoring and alerting framework**, where key performance indicators, such as record throughput, error rates, processing latency, and data quality scores, are continuously collected and exposed to operational dashboards. By integrating Boomi's native monitoring capabilities with external Business Intelligence tools or observability platforms, data stewards and integration architects would gain a real-time view of pipeline behavior, enabling proactive identification of bottlenecks and anomalies before they impact downstream systems.
This observability layer would also provide the data foundation for the Predictive Monitoring capabilities outlined in the AgentStudio evolution, where anomaly detection agents rely on historical execution patterns to anticipate failures; together, these patterns would transform the operational governance model from reactive troubleshooting to continuous, intelligence-driven pipeline optimization.

Final Remarks

This project demonstrates that the modernization of enterprise integration is not merely a technical migration, but a strategic repositioning of how data flows are conceived, governed, and evolved. The transition to Boomi delivered a modular, resilient ETL framework capable of handling the complexity of high-volume Master Data synchronization, while the integration of Generative AI capabilities through AgentStudio introduced a new operational layer that transforms manual, configuration-intensive tasks into governed, intent-driven workflows. The progressive nature of this thesis, from the initial pipeline design to the refinement of agentic architectures, reflects a broader principle: in complex integration environments, technical excellence does not emerge from the adoption of a single paradigm, but rather from the disciplined calibration of multiple complementary approaches. The framework delivered here is therefore not a fixed endpoint, but a scalable foundation designed to evolve alongside the enterprise's data ecosystem, capable of absorbing future technological advancements without requiring architectural reinvention.

References

Journal Articles

[17] Stefan Feuerriegel et al. "Generative AI". In: *Business & Information Systems Engineering* (2023).

[18] Leonardo Banh and Gero Strobel. "Generative artificial intelligence". In: *Electronic Markets* (2023).

Books

[8] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. John Wiley & Sons, Inc., 2013.

[9] Luigi D'Ercole and Filippo La Noce. *Data warehousing. Dal dato all'informazione*. Franco Angeli, 2000.

Online Resources

[1] *Mediamente Consulting*
mediamenteconsulting.it
Last visited: 1/1/2026

[2] *VarGroup*
vargroup.com
Last visited: 1/1/2026

[3] *iPaaS Platforms*
ibm.com/it-it/ipaas

[4] *Boomi*
boomi.com
Last visited: 1/2/2026

[5] *Boomi Platform*
help.boomi.com/docs/Atomsphere
Last visited: 1/2/2026

[6] *AI Agents for Boomi*
help.boomi.com/docs/Atomsphere/BoomiAgents
Last visited: 1/2/2026

[7] *Boomi DataHub*
help.boomi.com/docs/Atomsphere/DataHub
Last visited: 1/2/2026

[10] *Different Platforms (iPaaS, SaaS)*
ibm.com/it-it/iaas-paas-saas

[11] *Boomi Documentation*
help.boomi.com
Last visited: 1/2/2026

[12] *Rivery*
rivery.io

[13] *Boomi Oracle DB Connector*
help.boomi.com/docs/Atomsphere/Integration/Connectors

[14] *Process Properties*
help.boomi.com/docs/Atomsphere/Integration/ProcessProperties

[15] *Document Properties*
help.boomi.com/docs/Atomsphere/Integration/DocumentProperties

[16] *Dynamic Process Properties*
help.boomi.com/docs/Atomsphere/Integration/DynamicProcessProperties

[19] *AI*
help.boomi.com/docs/Atomsphere/AIAgents
Last visited: 1/2/2026

[20] *BoomiGPT*
help.boomi.com/docs/Atomsphere/BoomiGPT
Last visited: 1/2/2026

[21] *Boomi AgentStudio Documentation*
help.boomi.com/docs/Atomsphere/AgentStudio
Last visited: 1/2/2026

[22] *AgentGarden*
help.boomi.com/docs/Atomsphere/AgentGarden
Last visited: 1/2/2026

[23] *Boomi Event Streams*
help.boomi.com/docs/Atomsphere/EventStreams
Last visited: 1/3/2026