INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Towards a Reliable Integration Process That Prioritizes Content

PEDRO FILIPE MONTEIRO PACHECO
Outubro de 2021

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Towards a Reliable Integration Process That Prioritizes Content

Pedro Pacheco

A dissertation submitted in partial fulfillment of the requirements for the degree of Master in Computer Engineering, Specialization Area of Software Engineering

Advisor: Alexandre Bragança
Supervisor: Joana Barros

Porto, October 11, 2021

Abstract

The healthcare market increasingly demands Information Technology (IT) systems to reliably integrate information between them. ALERT®, a healthcare IT solution provided by the ALERT company, is a suite of products that integrates the workflow of different healthcare professionals. One of its many functionalities is medication prescription, which obviously requires medication content. This content is provided by external parties and needs to be properly integrated into the ALERT® product. Thus, a solution that appropriately extracts, transforms, and loads such information from multiple sources into its final destination is required.

With that in mind, the current project documents all the required development phases for a proper creation of such solution. This includes background research, analysis, design, implementation, and evaluation.

The main contribution of this dissertation is an approach for the creation of reliable data integration processes in complex systems, where data integrity is of extreme importance. To do so, a general-purpose Extraction-Transformation-Loading (ETL) framework, entitled ETL Framework (ETLF), has been conceived for that effect. Afterwards, with this software framework, a Medication Integration Platform (MIP) for ALERT® was also created. This platform allows a systematic creation of new medication ETL processes, while ensuring data integrity.

In its final appraisal, it was concluded that the developed platform is completely capable of systematically creating new ETL processes that, when avoidable, do not require human interaction. Additionally, it was also concluded that processes built using this platform are sustainable ETL processes that ensure the reliability of data.

Keywords: data, ETL, integration, integrity, reliability, healthcare

Resumo

O mercado da saúde exige cada vez mais que sistemas informáticos integrem informação entre eles de forma fidedigna. O ALERT®, uma solução informática de cuidados de saúde fornecida pela empresa ALERT, é uma suíte de produtos que integra o fluxo de trabalho de diferentes profissionais de saúde. Uma das suas funcionalidades é a prescrição de medicamentos, o que requer obviamente conteúdo de medicamentos. Este conteúdo é fornecido por entidades externas e precisa de ser devidamente integrado no ALERT®. Assim sendo, é necessária uma solução que extraia, transforme, e carregue adequadamente essa informação a partir de fontes múltiplas para o seu destino final.

Tendo isto em consideração, o presente projeto documenta todas as fases de desenvolvimento necessárias para uma criação adequada de tal solução. Isto inclui investigação contextual, análise, design, implementação, e avaliação.

A contribuição principal desta dissertação é uma abordagem para a criação de processos fidedignos de integração de dados em sistemas complexos, onde a integridade destes mesmos dados é de importância extrema. Para tal, foi criada uma framework de Extraction-Transformation-Loading (ETL), intitulada ETL Framework (ETLF). Depois, utilizando esta framework de software, foi também construída uma plataforma de integração de medicamentos para o ALERT®, designada Medication Integration Platform (MIP). Esta plataforma permite a criação sistemática de novos processos de ETL de medicamentos, assegurando em simultâneo a integridade dos dados.

Na sua avaliação final, concluiu-se que a plataforma desenvolvida é completamente capaz de criar sistematicamente novos processos ETL que, quando evitável, não requerem interação humana. Além disso, concluiu-se também que os processos construídos utilizando esta plataforma são processos ETL sustentáveis que garantem a fiabilidade dos dados.

List of Publications

Pedro Pacheco and Alexandre Bragança (2021). "Towards Systematic ETL Processes That Prioritize Content". In: *Simpósio de Engenharia Informática 2021*. Still awaiting submission and approval.

Acknowledgement

This dissertation is the culmination of a cycle that was not undertaken alone.

First, I would like to thank ALERT for the opportunity to conduct this dissertation. Although working remotely, there was always someone one click away and ready to help. A special thanks to Joana Barros, my supervisor, who was always available even when overwhelmed by work.

I would also like to thank the faculty of this master's program and my colleagues, who were instrumental in providing software engineering knowledge. A special thanks to Professor Alexandre Bragança for all the availability, advice, and feedback in this dissertation's course.

Last but not least, I would like to thank all my family and friends who, in one way or another, helped me survive this challenge.

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

ACL	Anti-Corruption Layer.
AHP	Analytic Hierarchy Process.
API	Application Programming Interface.
ATC	Anatomical Therapeutic Chemical.
BC	Bounded Context.
CDA	Clinical Document Architecture.
CI	Consistency Index.
CR	Consistency Ratio.
CRUD	Create, Read, Update, and Delete.
DICOM	Digital Imaging and Communications in Medicine.
DSA	Data Staging Area.
DSR	Design Science Research.
DW	Data Warehouse.
EHR	Electronic Health Record.
EIP	Enterprise Integration Pattern.
EMR	Electronic Medical Record.
ETL	Extraction-Transformation-Loading.
ETLF	ETL Framework.
FA	Functional Analysis.
FAST	Function Analysis System Technique.
FFE	Fuzzy Front End.
FHIR	Fast Healthcare Interoperability Resources.
FR	Functional Requirement.
FURPS+	Functional, Usability, Reliability, Performance, Supportability, and Other.
GAV	Global-as-View.
GUI	Graphical User Interface.
HL7	Health Level 7.
HTTP	Hypertext Transfer Protocol.
I/O	Input/Output.
ICD	International Classification of Diseases.
ICPC	International Classification of Primary Care.

INN	International Non-Proprietary Name.
IT	Information Technology.
JSON	JavaScript Object Notation.
KVDT	Kassenärztliche Vereinigung-Datentransfer.
LAV	Local-as-View.
LOINC	Logical Observation Identifiers Names and Codes.
MDE	Model Driven Engineering.
MIP	Medication Integration Platform.
NCD	New Concept Development.
NPD	New Product Development.
ODBC	Open Database Connectivity.
OO	Object-Oriented.
OOP	Object-Oriented Programming.
PACS	Picture Archiving and Communication Systems.
QEF	Quantitative Evaluation Framework.
QoD	Quality of Data.
QoX	Quality of X.
REST	Representational State Transfer.
RI	Random Consistency Index.
RPC	Remote Procedure Call.
SNOMED CT	SNOMED Clinical Terms.
SOA	Service-Oriented Architecture.
SQL	Structured Query Language.
SWOT	Strengths-Weaknesses-Opportunities-Threats.
TCP	Transmission Control Protocol.
UC	Use Case.
UDP	User Datagram Protocol.
UI	User Interface.
UML	Unified Modeling Language.
URI	Uniform Resource Identifier.
WMS	Workflow Management System.
XML	Extensible Markup Language.

Chapter 1

Introduction

This chapter introduces the reader to the main aspects of the current dissertation. Such dissertation has been developed within the scope of the Thesis/Dissertation/Internship curricular unit as a partial fulfillment of the requirements for the degree of Master in Computer Engineering, Specialization Area of Software Engineering. Firstly, a context is provided to show where the project is emersed within. Then, the problem that led to the emergence of this dissertation is explained, and the objectives to resolve it are defined. Following that, the hypothesis, contributions, and research approach are described. Finally, the structure of the current document is outlined, existent related work is briefly delineated, and, at the very end, the ALERT company is presented.

In healthcare systems there is a need to intertwine all sorts of information: from patient to medication data, from administration to appointment workflows, and from general to individual management. Generally speaking, the more parts a system contains that interact in a non-simple way, the more complex the system will be (Simon 1991). When determining the level of complexity of a physical system, Weng, Bhalla, and Iyengar (1999) refer that "the number of components and the intricacy of the interfaces between them, the number and intricacy of conditional branches, the degree of nesting, and the types of data structures" are all factors to take into consideration. The same can be inferred into any Information Technology (IT) system, where the more evolved and mature its domain is, more components, processes, data types, and relations between them will exist, progressively leading to more and more complex solutions. Healthcare systems can, therefore, be categorized as complex (Kannampallil et al. 2011; Khan et al. 2012).

Healthcare is increasingly changing from isolated treatment episodes to a continuous medical process that spawns across multiple professionals and institutions (Beyer et al. 2004; Kuhn and Giuse 2001), requiring IT systems in this area to integrate information between them. All of this while needing to deal with the fact that a failure or error in a type of system like this can, in some cases, put lives at risk (Maaskant et al. 2018; Priya et al. 2017). All of this convergence of information is "creating opportunities to organize these technologies into service relationships" (Demirkan 2013). Such service relationships use *Web Services*, which are a *de facto* standard for distributed computing communication, enabling for fast and reliable conversations between disparate systems. Because of that, combining data from distinct sources into meaningful information is now more convenient to achieve than it was in the past (Daigneau 2011; Fowler 2011; Hansen, Madnick, and Siegel 2003; Robinson 2011). However, convenience does not always mean less complexity, it can be quite the opposite. Code that combines multiple data models gets tangled because it is mixing two separate vocabularies together (Fowler 2015).

1.1 Context

The current project is being developed at ALERT, a company founded in 1999, dedicated "to the development of healthcare software and research on evolution & cancer" (ALERT n.d.[a]). Please refer to Section 1.9 for more information about the company. ALERT®[1] is the healthcare IT solution provided by ALERT. It is a suite of products that integrates the workflow of different professionals who interact with patients, adapting itself to different healthcare environments, including primary, hospital, and private medicine care. ALERT® provides patient-centered information recording, through a unique form of qualification, taking into account international standards in medicine. ALERT® is also an interoperable tool with several software applications, using Web services and international standards for the communication of medical content.

Figure 1.1 shows an updated view of the ALERT® product lines. The current project is inserted in ALERT® Electronic Medical Record (EMR): a solution that aims at the creation of paper-free clinical environments for hospitals, health centers, and private care centers. The functionality of prescribing medication to a patient is essential in today's healthcare applications, and is transversal to all subsystems of ALERT® EMR. In order to be able to perform such task, medication content needs to be integrated into the ALERT® product. Medication is not considered by itself a product, but rather the content of a product. It can be regarded as another complex system, in which concepts like International Non-Proprietary Name (INN), branded drug, administration route, measure unit, etc., are fundamental to its understanding. Please refer to *Business Context* (Section 3.1) for a more in-depth analysis of medication content and relevant ALERT® use cases.

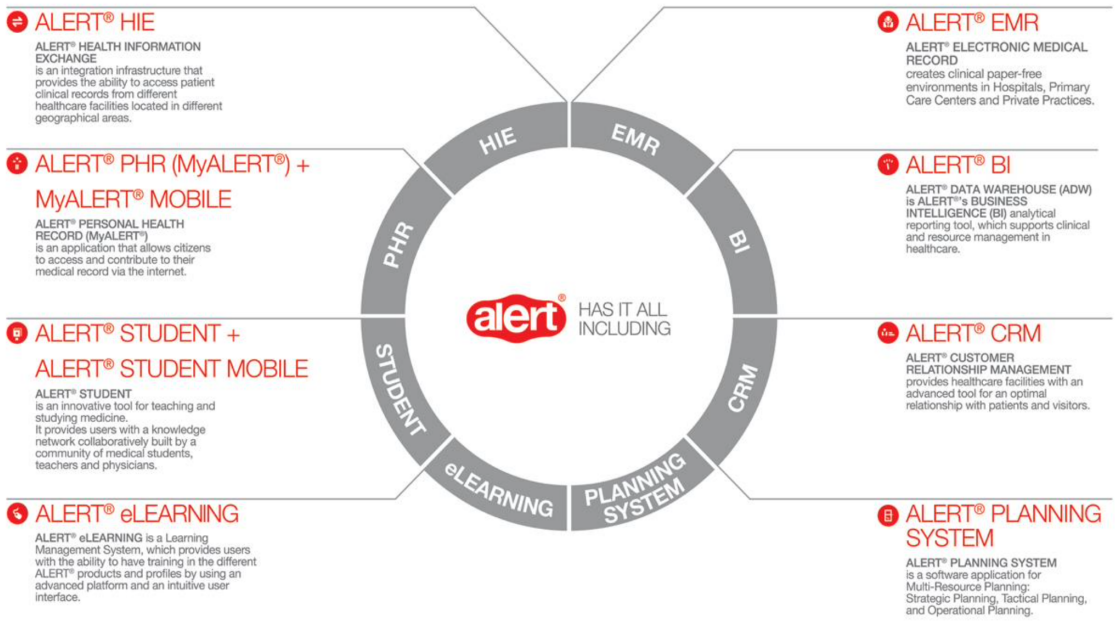

FIGURE 1.1: ALERT® product lines (ALERT 2019).

[1] ALERT and ALERT® do not refer the same concept. ALERT is a reference to the company, while ALERT® refers to the product made available by the company ALERT.

1.2 Problem

Healthcare is an extremely demanding market, where the electronic clinical record of the patient plays a crucial role in the provision of healthcare. MarketsandMarkets™ (2019) estimated that, by the year of 2024, the healthcare IT market will reach USD 390.7 billion, when, in 2019, this value was less than half (USD 187.6 billion), showing that there is an increasing demand for this type of solutions. These expenses are justified by the promise of reduced costs, enhanced workflows, improved efficiency, and error-free processes (Anderson 1997; Beyer et al. 2004; Demirkan 2013). As a side effect, "the convergence of various information and communication technologies [...] is creating opportunities to organize these technologies into service relationships" (Demirkan 2013). This is the main problem that lead to the development of the current dissertation: how should clinical content be updated quickly, automatically, and in accordance with the rules of the markets in which it operates? This specific project arises from the need to regularly extract medication content from different sources and different formats, transform this information into an ALERT®-specific vocabulary, and import it into the various versions of the ALERT® products.

In ALERT®, this medication integration process is divided into two sequential subprocesses: *Extraction-Transformation-Loading (ETL)* and *importation*. The ETL phase is responsible for creating a staging area that uses a data format and vocabulary specific to ALERT® and common to all of its versions. Then, the following importation process is responsible for using the data in this staging area to integrate it into the ALERT® final product. Please refer to Section 5.3.1 for a more in-depth understanding of how ALERT® requires external medication data to be integrated.

ALERT already implemented two solutions, but the ETL process in one of them is error-prone and in the other quite inefficient. The first solution executes the integration process effectively, doing its purpose as it should. It has, however, two main issues: requests for external data fail recurrently, and the source-code has not been developed in a way that protects the process from modifications to the external service. Because of this, the process needs to be re-executed manually until no failures occur, and the extraction of information fails regularly. The other integration solution is even more concerning: whenever the process has to be executed, each step (i.e., backup, extraction, transformation, persistence) needs to be manually triggered. Furthermore, multiple integration processes that aim at the integration of the same information type —even if from different providers— should not have different codebases. If this happens, software evolution and maintenance are not protected, as developers need to keep track of entirely different codebases that should be a single one. Besides these issues, there are other concerning aspects that need to be mentioned:

- New integration processes need to be developed from scratch when a new information service emerges;
- There is a substantial amount of accumulated time wasted fixing errors that should not exist in the first-place (e.g., new irrelevant item attribute);
- Modifying data transformation tasks is not simple because it requires adjusting multiple components in multiple places;

- If the external service Application Programming Interface (API) changes, modifying the data extraction task is not simple because each process is heavily dependent on the extractor;
- The advantages of an automated integration process with its full potential are not being achieved.

A new solution that provides an infrastructure for creating new, fully automated medication integration processes is vital. This will improve the quality of the final product and keep its content the most up to date as possible at all times, raising the client's satisfaction level.

1.3 Objectives

To solve the previously identified problems, a software solution that allows the creation of new integration processes is required. This solution must allow developers to systematically build new medication integration processes, without requiring an entirely new codebase for each one of them. Additionally, the solution must guarantee two other different aspects. The first one is that the integrated information is reliable, as in complete, accurate, and valid at all times. The second one is that processes must be fault-tolerant, as in they try their best not to raise unnecessary errors due the distributed and disparate nature of the systems. Such solution will be from this point forward referred to as *Medication Integration Platform (MIP)*. MIP is a software platform used to build the actual integration processes responsible for integrating medication information into the ALERT® final product. Please refer to *What is the Medication Integration Platform?* (Section 5.1) for a more in-depth description of MIP.

Development efforts in this dissertation are in the most problematic process: medication ETL. This means that MIP must be able to solve all the problems previously described in Section 1.2 and, in the future, be able to be extended with other desired behaviors, such as the final importation process. After building MIP, a specific ETL process will be created in order to provide a proof-of-concept of the developed solution and to also respond to the needs of ALERT. The specific requirements for MIP and concrete ETL processes are identified in Section 3.3.2. The ETL process itself must:

- *Extract* data from a single Web service;
- *Transform* the information, relating and enriching it according to predefined domain and technical rules, while always ensuring that it is valid, complete, and sufficient;
- *Load* the transformed data into a staging area.

To be able to develop an efficient solution capable of performing such tasks, approaches that respect good programming and integration practices have to be identified in order to provide an adequate design and implementation. Moreover, existing technologies that ease the development of ETL processes should be compared and selected. With that in mind, information regarding integration patterns, ETL processes, and Web services must be collected and synthesized. Furthermore, as a process that requires an indefinite execution, solutions for its continuous operation should be also described. Other side objectives for the current dissertation include the identification of existing integration solutions and international norms in the healthcare IT sector.

1.4 Hypothesis

A research hypothesis is a "specific, clear, and testable [...] predictive statement about the possible outcome of a scientific research study", and it is very important for planning that said study (Lavrakas 2008). Thus, to better guide the current dissertation work (i.e., knowledge search, development, evaluation of the final approach/solution), the following hypotheses were elaborated taking into account the problems and objectives described in the previous sections.

H1: MIP is capable of systematically creating new ETL processes that, when avoidable, do not require human interaction.

H2: Integration uses a sustainable ETL process that ensures that external data is valid, complete, and sufficient.

These hypotheses are better detailed and further discussed in Section 7.1.

1.5 Contributions

The current dissertation has two main contributions: *ETL Framework (ETLF)* and *MIP*.

ETLF is a general-purpose software framework that allows the creation of ETL processes. By providing the necessary building blocks, blueprints, and automated operations, ETLF serves as an architecture for systematically building reliable ETL processes. Through a highly configurable approach, the framework allows developers to focus on the content, rather than on how to address the process itself. Please refer to *What is the ETL Framework?* (Section 4.1) for an in-depth explanation of what ETLF is.

On the other hand, MIP is a coding platform used to create medication integration processes specific to ALERT®. It has been built using ETLF, and it allows an easier and faster development of integration processes, which empowers developers to focus on primary development tasks. With it, integration processes can be easily maintained and extended in the future. In a more theoretical side, MIP provides an approach for data integration in *complex systems* where *data integrity* is a crucial factor. Please refer to *What is the Medication Integration Platform?* (Section 5.1) for an in-depth explanation of what MIP is.

1.6 Approach

The current dissertation surfaced in an academical context that requires the creation of a final artifact. This artifact must be inline with the objectives of the company where it is being developed. With that in mind, the Design Science Research (DSR) is the research methodology applied in order to fulfill its objectives. DSR is a "problem-solving paradigm that seeks to enhance human knowledge via the creation of innovative artifacts" (Brocke, Hevner, and Maedche 2020). Through it, technology and science knowledge is strengthened, while also allowing for the creation of "innovative artifacts that solve problems" (Brocke, Hevner, and Maedche 2020). Considering all of this, the following steps represent this said approach:

1. **Interpret the problem to resolve**:
    - Understand the healthcare IT business;

- Analyze the framing of relevant services with the medication integration process;
- Examine the already implemented processes and documentation available from ALERT and service providers;
- Survey the project requirements using Quality of X (QoX)[2] and Quality of Data (QoD) metrics, applying them to the Functional, Usability, Reliability, Performance, Supportability, and Other (FURPS+) classification model;
- Survey other relevant aspects and doubts regarding the integration process.

2. **Synthesize knowledge related to the problem or approaches to its solution**:
   - Identify, describe, and synthesize approaches and technologies for the ETL process, including Web services, resorting to the existing literature;
   - Synthesize knowledge regarding data integration in healthcare systems, including international norms, resorting to the existing literature.
3. **Evaluate different approaches for the problem resolution**:
   - Evaluate discovered approaches and technologies through their adequacy to the project requirements, making use of the Analytic Hierarchy Process (AHP) method when necessary.
4. **Design a solution to the problem**:
   - Taking into account the evaluation carried out previously, design a solution, documenting the entire process via Unified Modeling Language (UML) while applying the C4 and 4+1 view models;
   - Analyze the design created, presenting and discussing alternatives.
5. **Build the solution for the problem**:
   - Implement the solution discussed in the design phase.
6. **Evaluate the designed and implemented solution**:
   - Extract relevant QoX and QoD metrics;
   - Carry out manual tests to assess subjective and qualitative criteria related to MIP (e.g., functionalities, ease of use);
   - Test the implemented process with automatic software tests;
   - Apply a Quantitative Evaluation Framework (QEF) assessment to evaluate the designed and implemented solution.

Throughout this document, the *C4* and *4+1 architectural view* models are used to visually express the software architecture of the solution constructed. *C4* provides a way to "help software development teams describe and communicate software architecture, both during up-front design sessions and when retrospectively documenting an existing codebase", while using "different levels of detail" (Brown n.d.). It is characterized by having four different levels (also known as viewpoints):

[2]The QoX abbreviation is a generalization for terms such as Quality of Service, Quality of Data, etc.

1. *System Context* — shows the big-picture of a software system and its interrelatedness with other ones;
2. *Container* — zooms into the system in hand, decomposing it into a set of containers[3];
3. *Component* — decomposes containers into their particular components;
4. *Code* — shows how each component is implemented as code.

On the other hand, the *4+1 view model* allows one to separately address different concerns regarding the requirements (Kruchten 1995). It has five associated views:

- *Logical View* — primarily describes functional requirements;
- *Process View* — addresses issues of concurrency and distribution, system integrity, and fault-tolerance;
- *Physical View* — primarily supports non-functional requirements where several elements are mapped into different nodes;
- *Development View* — focuses on the modules' organization in a software environment;
- *Scenarios* — all the previous views are shown working together in a set of scenarios or use cases.

1.7 Document Structure

The layout of the current document is as follows:

1. *Introduction* — introduces the reader to the main aspects of this dissertation;
2. *Background* — provides some background information on the main topics that surround this dissertation;
3. *Analysis* — analyzes and details the main aspects of the current project to properly develop a solution;
4. *ETL Framework* — explains and designs ETLF, a major component of the solution;
5. *Medication Integration Platform* — explains and designs MIP, the solution of this dissertation;
6. *Implementation* — describes implementation details of the solution;
7. *Evaluation* — evaluates the designed and implemented solution;
8. *Conclusion* — describes the main conclusions taken from the work accomplished.

In order to provide more exhaustive information about some of the topics and work done, the following appendices are added in the end of this document:

1. *Technologies Selection* — selects which technologies to use through AHP;
2. *Evaluation Criteria* — details QEF's evaluation criteria.

[3]A container is a "separately runnable/deployable unit (e.g., a separate process space) that executes code or stores data" (Brown n.d.).

1.8 Related Work

In the previous *Context*, *Problem*, and *Objectives* sections, the current dissertation is introduced and its main working topics pointed out. Now, a brief state of the art of these topics is presented in order to be able to evaluate what work has already been done in the area and if there are any missing pieces that can be further studied. The complete state of the art gathered for this dissertation can be seen in Section 2.1. Additionally, a technological review on Python tools is conducted in Section 2.4, considering that the current project requires the usage of this programming language (as is it will be stated in *Specification*, Section 3.3.2). Analyzing them both reveals that there is a lot of work done on this dissertation's topics.

Regarding *data integration* and *ETL processes*, the following topics have been extensively dealt with: patterns, quality, complexity, performance, big data, modeling, effort reduction, and assisting tools. *Web services* have also been extensively studied. The discovered topics that are relevant to the current project include: distributed computing related aspects, standards, and programming patterns. Nonetheless, from all the works found, it is possible to notice that the current dissertation's scope has not been yet fully explored top-to-bottom. This means that there is no relevant work discovered in data integration or ETL that proposes an approach on how to systematically create new processes, while guaranteeing total data completeness, accuracy, and validity. Approaches that allow it have been identified and described separately. However, none proposes a full-stack approach that allows the creation of data integration/ETL processes in systems where information cannot be incoherent, inaccurate, or outdated ever.

From the identified technologies, it is possible to see that, as expected, there are already a great deal of Python tools that can handle many of the necessary steps in ETL processes. Relevant identified frameworks operate within the fields of ETL pipelines, data analysis, and workflow management. Again, data completeness, accuracy, and validity are not the main motivating factors for any of them. Moreover, none of them provides developers with an initial structure for building data ETL/integration processes.

1.9 ALERT Life Sciences and Computing

As previously described, ALERT is a company in the healthcare IT sector dedicated to the development, implementation, and maintenance of its ALERT® products. Its main motto is "to improve health and prolong life, achieve profitability to benefit society, and inspire others to excel like we do" (ALERT n.d.[b]). In order to promote innovation, solidify processes, and comply with procedures, ALERT rooted the formula "*double checking while also innovating*" (Figure 1.2) in all of its employees.

FIGURE 1.2: The ALERT success formula (ALERT 2019).

In order to provide some more insight regarding how ALERT operates, its *value chain* is presented in Figure 1.3. *Value* can have many definitions, one of them is that a product or service has good *value* if it has an appropriate *performance-cost* relation (Miles 1972).

A value chain is a tool that allows the examination of all the activities —and their interactions— a firm performs to generate *value* for its customers (Barnes 2001). It separates a firm's global activity into strategically relevant ones. These are a result of the company's history, strategy, implementation of the strategies, and the economics of its activities. Generally, a value chain is divided into two major sections: primary activities and support activities. Primary activities are directly related to the creation, sale, maintenance, and support of a product or service. These correspond to the five sections represented at the base of the value chain in Figure 1.3. On the other hand, supporting activities are the backbone of the company's primary functions, and, therefore, are used to support them. In ALERT's value chain, support activities are represented in the four sections from top to bottom.

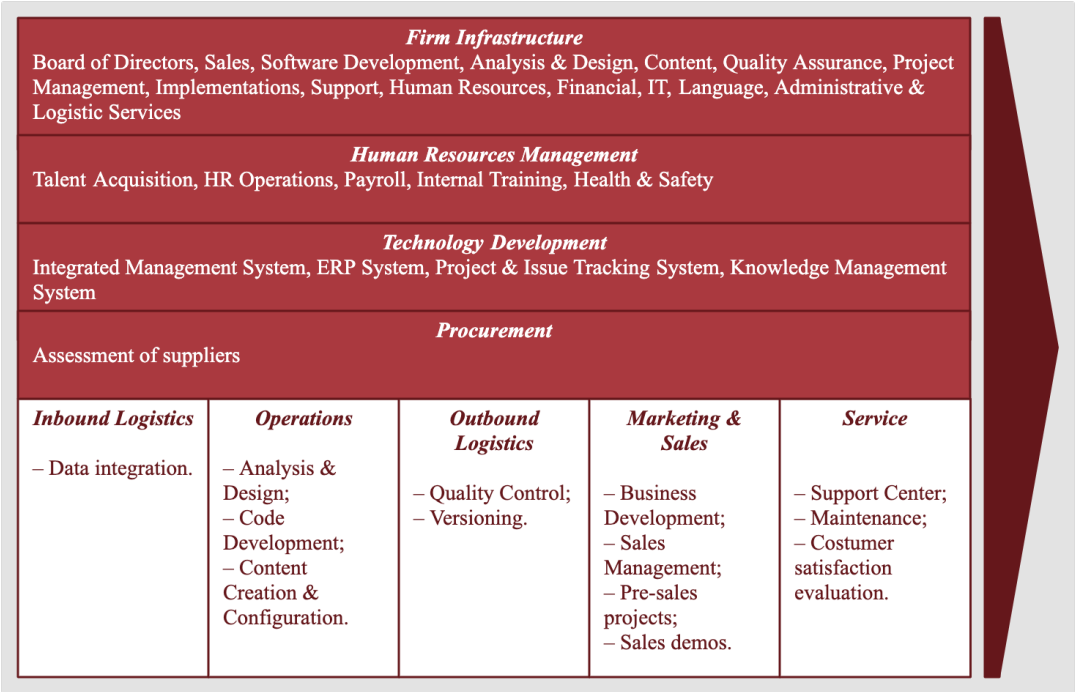

FIGURE 1.3: The value chain of ALERT (template extracted from Cre-ately).

When analyzing ALERT's value chain, it is possible to see that its primary activities are all related to software development. For ALERT, value is generated whenever quality is introduced into its ALERT® product. This might occur when features are added or improved, when new content is included, and many others. In certain cases it is necessary to acquire external information to include in the final product that cannot be constructed by ALERT —might it be because of either time and money, or legal imposition. Such is the case for the current dissertation, where external medication information needs to be integrated into the ALERT® product. ALERT development teams are divided into three major groups: Analysis & Design, Code Development, and Content Creation & Configuration. MIP is being developed by the Content Creation & Configuration team. In order to ensure software quality, there are teams for quality control and versioning. To provide some insight about the capabilities of ALERT® into possible future clients, there is a team specialized in presenting live demonstrations of the product. Last but not least, ALERT also takes special care in maintaining software high-level quality and customer

satisfaction. To do so, it has a specialized support center responsible for answering client needs on demand. This support center also incorporates a Network Operations Center responsible for providing preventive support through a continuous monitoring system based on optimized and automated alarms. This allows the anticipation of potential failures, blockages or problems, and to proceed with their proactive regularization.

Chapter 2

Background

This chapter is provides some background information on the main topics that surround this dissertation. Firstly, a current state of the art is provided. Then, important concepts and patterns that help to characterize and create data integration/ETL processes are presented. In the end, a technological review on existing programming tools that can support the implementation of the solution is provided.

2.1 State of the Art

Regarding data integration, there is already a lot of work done (Table 2.1), and some of it is not specific to *Web services*. Up until Hansen, Madnick, and Siegel (2003) examined opportunities for data integration via Web services, all the contributed work seems to be related to integration via database rather than using communication technologies for distributed computing. After that, the integration sector tends to favor Web services, starting off with a more conceptual effort and then evolving into tackling quality, complexity, and *big data*.

Table 2.1: Data integration related work.

Work Reference	**Description**
B. Czejdo and M. C. Taylor (1992) and B. D. Czejdo and Malcolm C. Taylor (1992)	Defines a system that uses an Object-Oriented (OO) approach "to define partly integrated views of a collection of autonomous information bases."
Klas, Fischer, and Aberer (1994)	Presents an approach to integrate a relational database system into a federated database system.
Keller, Mitterbauer, and Wagner (1998)	Introduces an integration framework related to database technology in an OO context.
Risch and Josifovski (2001)	Provides an overview of a data integration system. This system can process and execute queries over data stored in a local or external data sources. It also contains OO multi-database views "for reconciliation of data and schema heterogeneities among sources with various capabilities."
Ives and Halevy (2002)	Addresses performance needs regarding data integration queries.
Lenzerini (2002)	Presents an overview of data integration from a theoretical perspective.

Continued on next page

Table 2.1 – *Continued from previous page*

Work Reference	**Description**
Hansen, Madnick, and Siegel (2003)	Examines data integration opportunities in the Web services context.
Hohpe and Woolf (2004)	Extensively describes messaging integration patterns.
Akoka et al. (2007)	Presents a framework for quality evaluation in data integration systems.
Russom (2007)	Describes data complexity and its challenges regarding integration.
Jain et al. (2008)	Explores and develops a framework to be used in cause-and-effect relationships between requirements, architecture, and integration processes complexity. It also proposes recommendations based on causality results.
Dayal et al. (2009)	Describes the requirements for next-generation Business Intelligence systems. Its main goal is to "facilitate the design and implementation of optimal flows to meet business requirements."
Salem (2009)	Proposes an architecture for integrating relevant complex data into a repository of active Extensible Markup Language (XML).
Russom (2011)	Reports the evolution of data integration for the following years, helping users make more informed decisions regarding which technologies to use.
Dong and Srivastava (2013)	Explores the progress made in big data integration regarding schema alignment, record linkage, and data fusion. Challenges like volume and number of sources, velocity, variety, and veracity are dealt with.
Araujo (2014)	Proposes "a novel architecture for instance matching that takes into account the particularities of this heterogeneous and distributed setting."
Kadadi et al. (2014)	Focuses on the challenges of data integration in big data.
Dabroek (2016)	Proposes a system that takes advantage of previously provided associations between source schemas.
Gruenheid (2016)	Discusses modifications to existing integration pipelines due to changing datasets and novel ideas on how to interact with such data. It also focuses on designing conceptual elements that enable good online data integration systems.
Nyman (2019)	Presents a solution that can be used to make the data integration process more user-friendly and efficient.

As referred in the previous *Introduction* chapter, this dissertation focuses on data ETL, which is a process that helps to perform data integration. As such, it's relevant for the current dissertation to collect information about it. Such information is presented in Table 2.2. As expected, works related to ETL start off with more conceptual information and then take off to process modeling, effort reduction, programming frameworks, assisting tools, and performance improvements.

Table 2.2: ETL related work.

Work Reference	Description
Vassiliadis, Simitsis, and Skiadopoulos (2002)	Proposes a conceptual model for ETL and provides the foundations for its conceptual representation.
Simitsis (2004)	Proposes a conceptual model for the early stages of a data warehouse. Moreover, it presents a logical model that concentrates on the flow of data from sources to its final destination. It also describes a semi-automatic method for transitioning from the conceptual to the logical model.
Simitsis et al. (2009) and Thomsen and Pedersen (2011)	Presents a novel approach for ETL design that incorporates quality metrics.
Thomsen and Pedersen (2009, 2011)	Provides a Python framework with common ETL development functionalities.
Snehalatha (2010)	Proposes a logical model using XML for ETL processes, and develops an implementation to generate code from the logical model.
El-Sappagh, Hendawi, and Bastawissy (2011)	Investigates the problem of creating a standard for representing a conceptual model of ETL processes, and proposes a new model for the entity map diagram.
Deneke (2012)	Presents the "means to automate the specification of ETL workflows using a domain-specific modeling language."
Liu (2012)	Addresses Data Warehouse (DW) technologies for large-scale and right-time data regarding efficiency.
Liu, Thomsen, and Pedersen (2012)	Demonstrates a Python framework that uses Map-Reduce to achieve scalability.
Xavier (2012)	Develops mechanisms for the creation, control, and monitoring of ETL processes.
Akkaoui et al. (2013)	Presents model-to-text transformations to be used in ETL commercial tools, as well as model-to-model transformations that automatically update the ETL models to provide better maintenance over data source evolution.
Oliveira Martins (2015)	Provides a universal solution for automatic scaling and data freshness in a very large DW.
Silva (2016)	Studies the conditions in which using Big Data technologies is beneficial for information processing.
Triantos (2016)	Designs a new ETL framework for a fully centralized data repository using existing spreadsheets as data sources.
Deepika (2017)	Studies optimization methods of ETL processes in order to minimize execution times. To do so, a comparison of different approaches (i.e., partitioning, parallelization, multi-threading) is given.
Capelo and Belo (2018)	Presents a way to specify and validate ETL processes.
Homayouni, Ghosh, and Ray (2018)	Presents an approach for validating ETL processes using automated balancing tests that check for disparities between source and target data.

Continued on next page

Table 2.2 – *Continued from previous page*

Work Reference	Description
Ojwang' (2018)	Introduces a new methodology to automate the ETL rule and Structured Query Language (SQL) generation phases.
Biswas, Sarkar, and Mondal (2019)	Proposes an efficient incremental loading solution for real-time data integration.

Web services' existing work (Table 2.3) is also quite important to gather as they are extensively used in ETL processes. In the early ages of distributed computing, HTTP was not the preferred communication path due to the fact of it not having a standardized interface. Once such standards were developed, a growth in their usage occurred for data integration. In more recent years, the efforts regarding this technology have been associated with patterns and guides for the implementation of Representational State Transfer (REST) and SOAP services and clients.

Table 2.3: Web services related work.

Work Reference	Description
Waldo et al. (1997)	Describes distributed computing in its early stages.
Parkin, Ingham, and Morgan (2007)	Describes an approach that provides reliable message passing through Web services.
Mulligan and Gračanin (2009)	Describes the conceptual design of a interaction independence middleware and the role of Web services in it.
Daigneau (2011)	Catalogs design solutions for Web services that leverage SOAP or follow REST.
Kalin (2013)	Guide for the development of REST-style and SOAP-based Web services and clients.
Ramalingam and Vaswani (2013)	Studies failures, duplicate messages, and idempotence in distributed services.

As this dissertation is inserted into the healthcare IT sector, it's also important to also collect information relative to it (Table 2.4). Analyzing the initial work, it is possible to verify that healthcare solutions needed to evolve from single episodes to a continuous treatment mindset, and, consequently, standards needed to be developed and worked upon. This indirectly caused an increase of complexity in healthcare IT solutions, as the state of art suggests.

Table 2.4: Healthcare IT related work.

Work Reference	Description
Kuhn and Giuse (2001)	Discusses the evolution of information systems in the healthcare sector.
Beyer et al. (2004)	Analyzes the potential of healthcare information systems as an integrated network.

Continued on next page

Table 2.4 – *Continued from previous page*

Work Reference	**Description**
Bicer et al. (2005)	Provides a framework for semantic interoperability of exchanged messages in the healthcare domain.
Lenz, Beyer, and Kuhn (2007)	Analyzes semantic integration in healthcare networks.
Kannampallil et al. (2011)	Proposes a theoretical approach on how to understand and study complexity in healthcare systems.
Khan et al. (2012)	Proposes a system to enable high-level accuracy in mappings between standards.
Thuy, Y.-K. Lee, and S. Lee (2012)	Proposes a process for measuring structural and semantic similarity in healthcare XML Schemas.
Demirkan (2013)	Proposes a framework that provides opportunities for healthcare organizations to deploy solutions with fewer risks and increased context awareness.
Winter et al. (2018)	Provides insight into architectural design issues of a solution for hospitals that aim at integrating information between them.
Le Sueur et al. (2020) and Zhang et al. (2020)	Discusses challenges regarding data integration in the healthcare area.

2.2 Theoretical Concepts

As described in *Problem* (Section 1.2), data integration is becoming more important every day for today's real-world applications. *Data Integration* can be defined as "the problem of combining data residing at different sources, and providing the user with a unified view of these data" (Lenzerini 2002). In order to develop an efficient solution for a problem, many times often, the development team cannot capture all the needed data, nor develop all the processing required by themselves alone. They might be able to, but the costs to do so could be far superior to those of simply integrating the needed information into their domain. Other times, the problem might even not be related to effort contention, but rather an imposed legal restriction. That being said, one could state that modern day's common enterprise applications are not "islands" and, inevitably, need to communicate with outer systems (Fowler 2011; Robinson 2011). Moreover, there is a recurrent challenge regarding *data integration* related to semantics: the same word in different contexts might have different meanings. With all of this in mind, *data integration* solutions are evolving in terms of complexity, justified by an increase in data complexity. "Success with complex data demands additions to data integration infrastructure, especially support for data quality and data exchange standards" (Russom 2007). Russom (2011) researched on what data integration would evolve into for the next generation of technologies. 40% of respondents refer that a unified platform that supports data integration with data quality, governance, etc., was needed. Furthermore, Russom shows that ETL and batch processing were expected to have a strong commitment due to the community trends, but a low growth in usage. Data governance and data quality were in between the options that show the most growth and commitment potential.

ETL is a paradigm that describes a conceptual framework for relevant business information to be collected and centralized. ETL tools are "pieces of software responsible for the extraction of data from several sources, their cleansing, customization and insertion into

a data warehouse" (Vassiliadis, Simitsis, and Skiadopoulos 2002). Figure 2.1 depicts this high-level framework, where data is (a) extracted from the sources, (b) transformed and cleaned in a Data Staging Area (DSA), and (c) loaded into the DW.

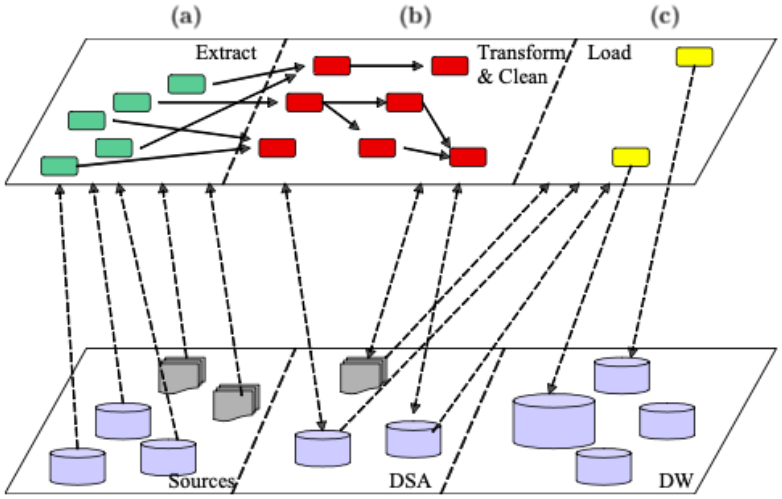

FIGURE 2.1: The environment of ETL processes (Vassiliadis, Simitsis, and Skiadopoulos 2002).

Also related to data integration, another important concept to refer is the *Enterprise Integration Pattern (EIP)*. EIPs are a set of patterns that aid in the development of data integration processes, more concretely in its first two stages: data extraction and transformation. In their book, Hohpe and Woolf (2004) extensively describe all of these EIPs. Although the book is directed to communications via *Messaging APIs*, some of them are also relevant to the other communication styles (see *API Styles* in Section 2.2.6) and can also be applied to them. Section 2.3 will take an in-depth look at the patterns most relevant to the current dissertation.

2.2.1 Data Integration and Related Approaches

According to Lenzerini (2002), *Data Integration* is a process that gathers and fuses information from a set of sources into a global schema. "The sources contain the real data, while the global schema provides a reconciled, integrated, and virtual view of the underlying sources." Lenzerini formalizes a data integration system $\mathcal{I}$ in terms of $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, where:

- $\mathcal{G}$ represents the global schema, expressed in the language $\mathcal{L}_\mathcal{G}$ with alphabet $\mathcal{A}_\mathcal{G}$, which contains a symbol for each element of $\mathcal{G}$;
- $\mathcal{S}$ represents the source schema, expressed in the language $\mathcal{L}_\mathcal{S}$ with alphabet $\mathcal{A}_\mathcal{S}$, which contains a symbol for each element of $\mathcal{S}$;
- $\mathcal{M}$ represents the mapping between $\mathcal{G}$ and $\mathcal{S}$, constituted by assertions of the forms $q_\mathcal{S} \rightsquigarrow q_\mathcal{G}$ and $q_\mathcal{G} \rightsquigarrow q_\mathcal{S}$. $q_\mathcal{S}$ is a query that uses language $\mathcal{L}_{\mathcal{M},\mathcal{S}}$ with alphabet $\mathcal{A}_\mathcal{S}$, while $q_\mathcal{G}$ uses a language $\mathcal{L}_{\mathcal{M},\mathcal{G}}$ with alphabet $\mathcal{A}_\mathcal{G}$. Each assertion specifies that the concept represented by the first query over its schema corresponds to the concept in the other schema represented by the latter query.

Additionally, Lenzerini (2002) also describes two basic approaches that help to model the relation between sources and global schema: *Local-as-View (LAV)* and *Global-as-View (GAV)*.

> *"[...] while in LAV the designer may concentrate on declaratively specifying the content of the source in terms of the global schema, in GAV, one is forced to specify how to get the data of the global schema by means of queries over the sources"* (Lenzerini 2002).

Local-as-View

Using LAV, $\mathcal{M}$ "associates to each element s of the source schema $\mathcal{S}$ a query $q_{\mathcal{G}}$ over $\mathcal{G}$." This means that $\mathcal{L}_{\mathcal{M},\mathcal{S}}$ only allows expressions with one element belonging to $\mathcal{A}_{\mathcal{S}}$. In other words, "the content of each source s should be characterized in terms of a view $q_{\mathcal{G}}$ over the global schema", resulting in the assertion $s \rightsquigarrow q_{\mathcal{G}}$.

The LAV approach is most useful when the integration system is based on an enterprise model or an ontology[1], where the global schema is stable and well-established. A benefit of using this approach is the higher extensibility: when a new source needs to be integrated, the mapping only needs to be complemented with a new assertion. However, it has the downsides of difficulties in processing queries, and inferring how to use the sources.

Global-as-View

On the contrary to LAV, GAV's $\mathcal{M}$ "associates to each element g in $\mathcal{G}$ a query $q_{\mathcal{S}}$ over $\mathcal{S}$." This means that $\mathcal{L}_{\mathcal{M},\mathcal{G}}$ only allows expressions with one element belonging to $\mathcal{A}_{\mathcal{G}}$. In other words, "the content of each element g [...] should be characterized in terms of a view $q_{\mathcal{S}}$ over the sources", resulting in the assertion $g \rightsquigarrow q_{\mathcal{S}}$.

GAV is best used when the set of sources is stable, due to the fact that this approach "tells the system how to use the sources to retrieve data." Furthermore, GAV allows an easier query processing due to the mapping functions directly specifying "which source queries corresponds to the elements of the global schema." The downside is the reduced extensibility, because introducing a new source may have an impact on multiple elements of the global schema.

2.2.2 Bounded Contexts

As implied by the integration approaches, any software solution that integrates data between disparate systems will, sooner or later, be required to transform incoming data formats into their own context. This incoming format represents an external *Bounded Context (BC)*. A BC is an "explicit boundary within which a domain model exists." "Inside the boundary all terms and phrases [...] have specific meaning" (Vernon 2013). In other words, data from one external BC needs to be converted into the integration system's BC. "Code that combines two bounded contexts gets convoluted because it's mixing two separate vocabularies together", thus a clear boundary between them is required (Fowler 2015). This boundary is called of *Anti-Corruption Layer (ACL)*.

[1] An ontology is a shared conceptualization of "an abstract model of how people think about things in the world, usually restricted to a particular subject area" (Gruninger and J. Lee 2002).

2.2.3 Types of Integration Projects

According to Hohpe and Woolf (2004), there are six predominant integration project types:

- *Information Portal* — information is gathered from multiple sources into a single display, so that the user has no need to access multiple systems;
- *Data Replication* — some businesses require the same exact data across multiple systems;
- *Shared Business Function* — some businesses need to execute the same functions in different systems;
- *Service-Oriented Architecture (SOA)* — this project type occurs when a business has (a) a collection of useful services and (b) an interface so that an application can negotiate with those services;
- *Distributed Business Process* — a business transaction might spread across multiple systems;
- *Business-to-Business Integration* — outside parties (e.g., suppliers, partners) make some business functions available to external companies.

2.2.4 Integration Styles

According to Hohpe and Woolf (2004), there are multiple approaches that can be used to provide/consume other applications' services in a data integration context:

- *File Transfer* — one application consumes files produced by other systems;
- *Shared Database* — consumer and producer share a common database;
- *Remote Procedure Invocation* — each application exposes procedures that are invoked remotely;
- *Messaging* — communication is conducted via asynchronous messages, using customizable formats.

2.2.5 Data Integration in Healthcare Systems

Modern solutions in the healthcare business demand a continuous treatment course that involves multiple institutions and professionals in a process that spawns across multiple healthcare systems (Beyer et al. 2004; Kuhn and Giuse 2001; Lenz, Beyer, and Kuhn 2007). This requires system integration and inter-institutional support for healthcare processes. It is not feasible for all health information systems to conform to a single, unified standard, mainly because most part of them are proprietary. However, this problem should be addressed at the semantic level, where "formally defined domain concepts" need be understood by all participating systems (Bicer et al. 2005). It becomes even more important in the healthcare domain to do so, because "healthcare is a complex domain in terms of complex concepts" (Khan et al. 2012). Consequently, "standards based on common ontologies need to be further developed" (Beyer et al. 2004) and applied in order to facilitate and maximize integration opportunities in this complex sector (Le Sueur et al. 2020; Zhang et al. 2020). This is the main reason why the usage of standards in *technical* and *semantic integration* is required (Lenz, Beyer, and Kuhn 2007). *Technical*

integration refers to IT-related technical standards, while *semantic integration* refers to the meaning of information. In order for a process of data integration to be successful, three compatibility types need to be achieved (Beyer et al. 2004):

- *Syntactical* — related to *technical data structures* and *encodings*;
- *Ontological* — related to *type level semantics*;
- *Terminological* — related to *instance level semantics*.

Minor incompatibilities in these items can be handled with a typical ETL process. IT systems should also enable medical knowledge to be modeled by domain experts rather than the IT specialists (Lenz, Beyer, and Kuhn 2007).

Syntactical Compatibility

Lenz, Beyer, and Kuhn (2007) and Thuy, Y.-K. Lee, and S. Lee (2012), argue that the most accepted data format in healthcare IT solutions is XML. The main reasons for this include:

- Easy creation of correct medical documents;
- Easy storage in a distributed system environment;
- Improved syntactical compatibility with other commonly accepted formats.

Ontological Compatibility

> *"In the real healthcare XML Schema documents, many elements in the schemas have an identical semantics, but they may have the different name and structure. On the contrary, many elements have a similar model but are not similar in the meaning. Therefore, one of the main problems with integration of the healthcare data is how to assess the similarity of elements among XML Schema documents"* (Thuy, Y.-K. Lee, and S. Lee 2012).

An ontology can be defined as "a dictionary of terms formulated in a canonical syntax and with commonly accepted definitions designed to yield a lexical or taxonomical framework for knowledge representation which can be shared by different information-systems communities" (Smith 2003). At this level, if both parties are not committed to the same standard, integration might not be possible without modifying at least one schema of one participant. To avoid this, both parties should adhere to some healthcare-specific norm. Some well accepted ones include Health Level 7 (HL7), Digital Imaging and Communications in Medicine (DICOM), Kassenärztliche Vereinigung-Datentransfer (KVDT), and openEHR.

HL7 is a "messaging standard used for communicating medical information between health information systems" (Khan et al. 2012). Its main assumption is that, when a *trigger event* occurs, messages are exchanged between applications (Bicer et al. 2005). Many versions have been developed and one of them is HL7 Clinical Document Architecture (CDA). It is an XML-based standard that provides specifications for various types of clinical documents to be exchanged between providers and patients. It has three levels: level 1 only contains textual information, level 2 contains coded sections, and level 3 contains structured entries (HL7 2019a; Winter et al. 2018). A more recent standard provided by HL7 is Fast Healthcare Interoperability Resources (FHIR). FHIR solutions are built using modular components (i.e., resources) and assembled into working systems (HL7 2019b).

It is used to arrange patient data (e.g., allergies, intolerances, family member histories, medication requests) and provide it to remote application systems through *RESTful Web services*. Several FHIR resources can be combined to provide more complete documents (Winter et al. 2018). FHIR has a medication dedicated module that provides standards for the communication of medication knowledge, request, dispense, administration, etc. Here, a "medication resource represents an actual medication that can be given to a patient, and referenced by the other medication resources" (HL7 2019c). Besides this, it also provides separate standards for immunization resources.

DICOM is a standard that defines formats for medical images and related information, enabling an exchange of data with the necessary quality for clinical use (DICOM n.d.). It is now the basis for Picture Archiving and Communication Systems (PACS), which is helping radiology departments move towards a fully digital environment by providing an interface consumed by external PACS (Winter et al. 2018).

KVDT, also known as xDT, is a family of communication formats for ambulatory information exchange, that is commonly used in Germany. Messages in this standard have several "records" that are composed by several fields. These last ones are also characterized by having a specific sequence that must be respected (European Commission 2013; Kassenärztliche Bundesvereinigung 2020).

OpenEHR is a "patient records related content modeling standard related to management and storage, retrieval and exchange of health data in the form of Electronic Health Record (EHR)" (Khan et al. 2012). It provides open-source specifications, models, and software for developers to create the desired standards (openEHR 2018).

Terminological Compatibility

This terminological type refers to instances of a given element. Controlled terminologies, standard catalogs, or classifications like International Classification of Diseases (ICD), Logical Observation Identifiers Names and Codes (LOINC), International Classification of Primary Care (ICPC), and SNOMED Clinical Terms (SNOMED CT) should be used to avoid incompatibilities.

ICD offers standard terminologies for coded medical data (Winter et al. 2018). It's the foundation for the identification of health trends and statistics globally, and a standard for diagnostic classification. It includes diseases, disorders, injuries, and other health conditions (WHO n.d.[d]). LOINC is another common language that can be used to identify health measurements, observations, and documents, offering standard terminologies for coded medical data (LOINC n.d.; Winter et al. 2018). Another classification system is ICPC, which provides mechanism for applying labels to several stages of primary care. Its nomenclature can be used to describe reasons of encounters and even to provide diagnosis (Bentsen 1986; WHO n.d.[b]). SNOMED CT defines code systems and "relations between subject entities for an unambiguous specification of certain medical information" (Winter et al. 2018). Its main objective is to develop a global language for health, through an "accurate and effective exchange of health information" (SNOMED International n.d.).

Specifically related to medication, there are two important catalogs to refer: INN and Anatomical Therapeutic Chemical (ATC). INNs facilitate the identification of pharmaceutical substances or active pharmaceutical ingredients. Each one is unique and globally identifiable (WHO n.d.[c]), making communications more precise and helping to avoid prescription errors. In the ATC classification system, "active substances are divided into

different groups according to the organ or system on which they act and their therapeutic, pharmacological and chemical properties" (WHO n.d.[a]). Drugs can be classified in groups from five different levels. The first level can be one out of fourteen anatomical or pharmacological groups. The other ones are sub-categories of the previous ones.

2.2.6 Web Services

In his book, Daigneau (2011) explains very well the concepts related to *Web services*, being cited for the majority of this section.

> *"There is so much noise today in the Web services world, it's a delicate and complex endeavor just to identify the specifications and technologies to focus on. The goal remains the same, however —software helps you solve a problem. [...] Many people say Web services is just a new and open way to solve our existing integration problems"* (Crupi 2003).

In modern programming, *Web services* are a standard technology that allows developers to introduce a distributed nature into any system. Web services are a set of software functions that provide access to any task and information, or perform a function in disparate systems through business functions called over Hypertext Transfer Protocol (HTTP). According to Daigneau (2011) and Kalin (2013), the reasons for its generalized usage are:

- *Open infrastructure* — Web services use ubiquitous and interoperable open standards;
- *Platform and language transparency* — Web services use technologically independent open standards;
- *Modular design* — Web services ease reuse and sharing of common logic among diverse clients, and its simplicity facilitates service decomposition of complex systems;
- Services and clients evolve independently (if no breaking-changes occur in the service side);
- Easy learning and implementation.

Nothing comes without its restraints and Web services are no exception. They do, in fact, reduce coupling between client and service provider by allowing the provider to use different technologies than those used by the client. However, this technological and functional decoupling doesn't come alone. Web services require more computing than a simple in-process method call. On one side, the client needs to serialize the response into a stream of bytes and then transmit it across the network. On the other side, the service must deserialize the request and parse it in to its own logic. Additionally, in order to be able to respond appropriately to more complex calls that require more than a simple HTTP status code, the process needs to be executed in reverse: the service serializes a response, sends it over the network, and then the client deserializes it. All this processing, computing, and transmission takes time and, therefore, creates a latency problem. If this is ignored, it might lead to a system with extensive performance issues.

Another major problem is what Waldo et al. (1997) refer to as *partial failure*. In a distributed system there are multiple points of failure: client, service, and network. When one of them fails, the other continue to function properly and must be able to maintain their own consistency. The interfaces used for the communication "must be designed in such a way that it is possible for the objects to react in a consistent way to possible partial

failures" (Waldo et al. 1997). This means that the interfaces for local and remote objects need to have a different implementation from one another. In other words, it is required for the developer to think in a non-traditional way, where all the code and logic is located around a monolithic system.

Taking all of this into account, deciding to use a Web service should be heavily weighted against its pros and cons. Example of scenarios taken from Daigneau (2011) where the usage of "cross-machine" calls are profitable include:

- The client and service belong to different domains, and the service functions are quite complex to import;
- The client is a complex system that incorporates functions (or data) related to other domains managed by different organizations, each evolving at different rates;
- The divide client-server is natural (e.g., mobile and desktop applications that share common functions).

HTTP requires client and server to establish a connection. This takes up some time, and in systems where performance is demanded it can turn out to be an issue. Transmission Control Protocol (TCP) —the protocol used by default— is a connection-oriented protocol that requires a three-way handshake in order to start exchanging data. An alternative to TCP is the User Datagram Protocol (UDP), which is a connectionless protocol that can be a faster option if loss of data, duplication, and order of reception are not a major concern. Moreover, many Web services can be configured to stream data, which minimizes memory usage on both ends. With it, the client doesn't need to wait for all the information to arrive at once, increasing its performance. It is the best option to deliver large quantities of information.

API Styles

The three most common styles of Web service APIs are *Remote Procedure Call (RPC)*, *Message*, and *Resource* (Daigneau 2011). Taking into account the current dissertation's purpose, only the *RPC* and *Resource* styles will be further analyzed, as they are the ones most usable by ETL processes.

With regard to the *RPC API*, one might say that it is an evolution from the old RPC protocol, being the usage of HTTP to communicate between machines their main difference. As shown in Figure 2.2, in this kind of API, the client sends a request with the name of the procedure to execute alongside its arguments. On the other end, the service will receive it, execute the desired function with the given parameters, and return the result to the client.

On the other hand, the *Resource API* leverages the HTTP methods (i.e., GET, POST, PUT, and DELETE), circumventing the need to use a domain-specific interface from the client's Create, Read, Update, and Delete (CRUD) operations. As Figure 2.3 demonstrates, each request should specify the HTTP method, the resource's Uniform Resource Identifier (URI), and the media to be sent. The service, after receiving this specific request, processes it, and responds with a status code together with a content message. *Resource APIs* are characterized by providing a gateway into a service's resources through representations that "capture the current or intended state of a resource." It was from this that REST —one of the most widely known architectural styles for Web services— was devised by Fielding (2000).

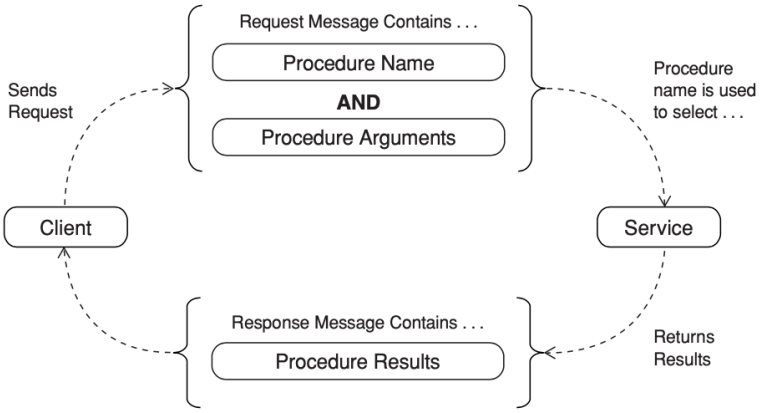

FIGURE 2.2: *RPC API* request process (Daigneau 2011).

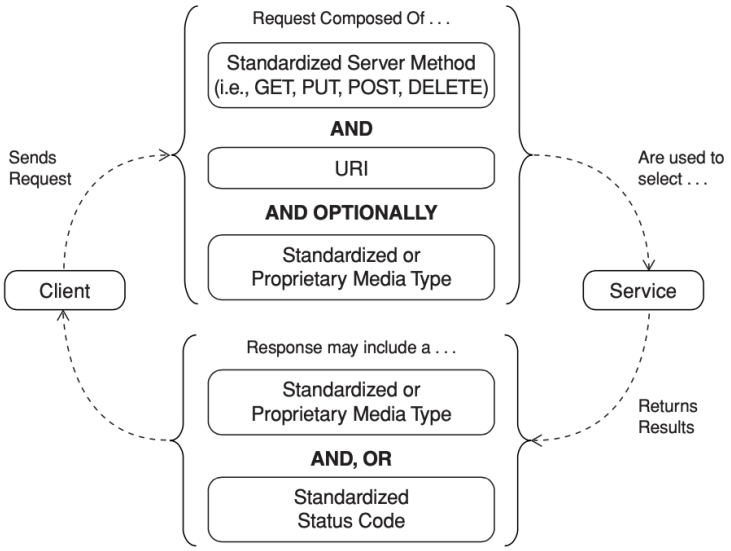

FIGURE 2.3: *Resource API* request process (Daigneau 2011).

Breaking Changes

In his book, Daigneau (2011) describes how both clients and services should evolve into their different versions without introducing the so called *breaking changes*. A breaking change is "any change that forces client developers to update their code or change configurations." If the client fails to do so, they may experience runtime exceptions. Some examples of breaking changes include changing the service domain, URI patterns, media types, or message structures.

Communication Standards

As described earlier in this section, *Web services* make use of ubiquitous and interoperable open standards. These standards can come in three different flavors: SOAP, REST, and GraphQL. While SOAP and REST are still dominant, GraphQL is not as standardized as the others are yet. Because of this, and added to the fact that currently no relevant medication information service provider uses this style, it will be no longer addressed in the current dissertation.

Using the words of Mulligan and Gračanin (2009), SOAP and REST can be compared in the following simple and concise way:

> *"While SOAP adheres very closely to the* RPC *model, REST revolves around the concept of* resources *and focuses on using the inherent power of HTTP to retrieve representations of these resources in varying states. [...] REST request and response messages add zero overhead to the messages being transmitted apart from the standard HTML headers which are used to route the packets through the network. SOAP, on the other hand, encloses each message payload within an additional SOAP 'envelope' set of XML tags and adds a few SOAP-related headers to the outbound HTTP packet. This, and this alone, contributes to the added bloated in SOAP packets. What's more, REST is able to take advantage of simplistic CRUD situations and execute them much more efficiently than the SOAP implementation."*

2.3 Patterns

This section aims to demonstrate a core set of patterns that can be applied in implementations of ETL processes. The works of Hohpe and Woolf (2004) and Daigneau (2011) are extensively used as they contain the most essential patterns considering the current dissertation's purpose.

2.3.1 Workflow Patterns

This section describes patterns that allow one to control the overall workflow of data processing with different approaches. These patterns are *Pipes and Filters*, *Routing Slip*, and the *Process Manager*. Table 2.5 presents a summarized side-by-side comparison between the three.

Table 2.5: Workflow patterns comparison (Hohpe and Woolf 2004).

Pipes and Filters	**Routing Slip**	**Process Manager**
Supports complex message flow.	Supports only simple, linear flow.	Supports complex message flow.
Difficult to change flow.	Easy to change flow.	Easy to change flow.
No central point of failure.	Potential point of failure (compute routing table).	Potential point of failure (*Process Manager*).
Efficient distributed runtime architecture.	Mostly distributed.	Hub-and-spoke architecture may lead to bottleneck.
No central point of administration and reporting.	Central point of administration, but not reporting.	Central point of administration and reporting.

Pipes and Filters

Pipes and Filters allows for complex processing to be done on a message while maintaining independence and flexibility. It's an architectural style that divides a large processing task into a "sequence of smaller, independent processing steps (Filters) that are connected by channels (Pipes)", as Figure 2.4 demonstrates. Each filter receives messages in the inbound pipe, processes it, and then transmits it to the outbound pipe. *Pipes and Filters* also allow for steps processing to be easily employed.

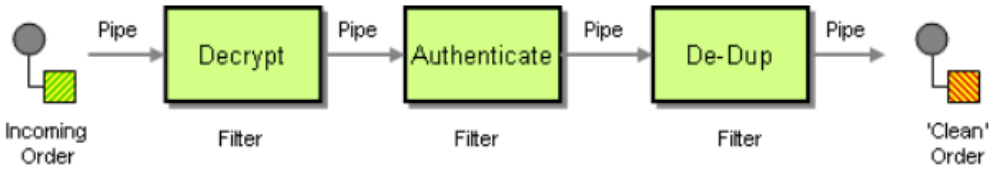

FIGURE 2.4: Behavior example of *Pipes and Filters* (Hohpe and Woolf 2004).

Routing Slip

A *Routing Slip* is a *Composed Router* (see Section 2.3.2) that allows one to control a message's path from a central point. As Figure 2.5 shows, a list of processing steps is attached to the message. After successful processing, each process passes the message to the next step. "This routing logic is built into the processing component itself." This router is most useful for binary validating steps, *stateless* transformations, and gathering data while making no decisions.

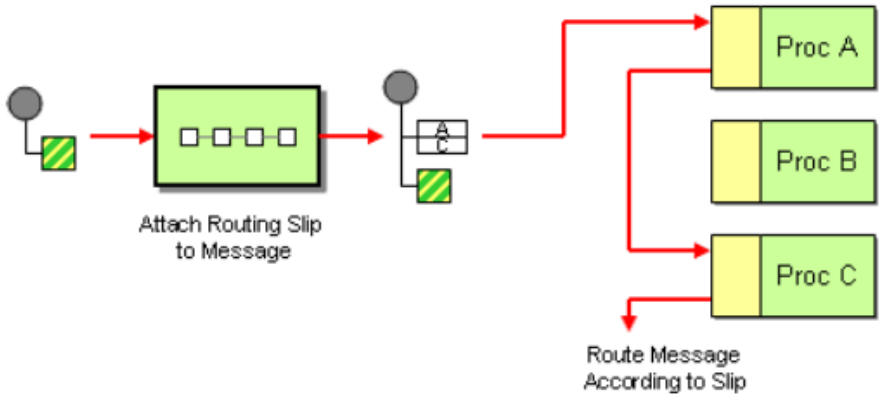

FIGURE 2.5: Behavior of a *Routing Slip* (Hohpe and Woolf 2004).

Process Manager

The main difference of the *Process Manager* is that it is more flexible compared to the other ones. However, it "requires the message to return to a central component after each function," just as Figure 2.6 illustrates. The next step to be carried out is evaluated dynamically, based on intermediate results.

2.3.2 Routing Patterns

This kind of patterns are routers that consume a message and resend it to a different channel according to specific conditions. Routing a message allows to decouple its source from its ultimate destination. *Message Routers* described in this document can be divided into two categories.

- *Simple*: routes messages from one channel to one or more outbound channels;
- *Composed*: combined from multiple *Simple Routers*;

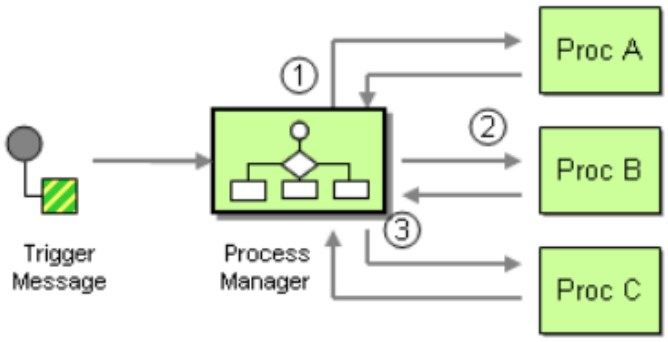

FIGURE 2.6: Behavior of a *Process Manager* (Hohpe and Woolf 2004).

Content-Based Routing

The *Content-Based Routing* is a *Simple Router* that inspects the content of a message and routes it to the proper destination. Figure 2.7 shows an example of this router's behavior. When a new order message is sent, its content is dynamically analyzed and redirected to the desired service.

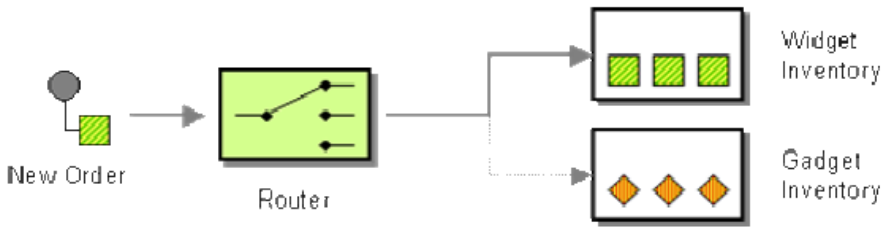

FIGURE 2.7: Behavior example of a *Content-Based Router* (Hohpe and Woolf 2004).

Splitter

The *Splitter* is a *Simple Router* that can be used to split a message into individual, smaller ones to be processed as desired. Figure 2.8 shows how this router allows the separate processing of each part of a message composed by multiple elements.

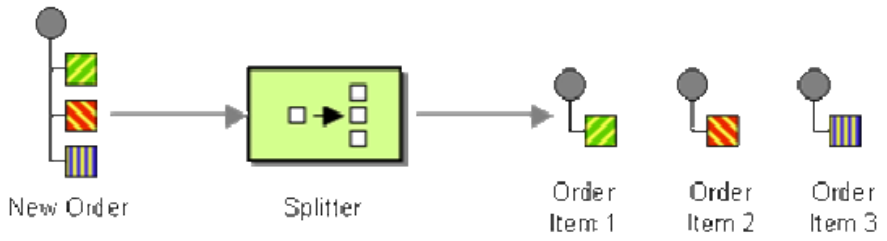

FIGURE 2.8: Behavior example of a *Splitter* (Hohpe and Woolf 2004).

Aggregator

The *Aggregator* is a *Simple Router* that recombines a stream of messages by grouping the related ones and combining them into a specific message. It can make decisions of what information is relevant to aggregate or not, or simply join all messages into a single

one. When designing it, it's necessary to specify (a) what messages are correlated, (b) when it's ready to aggregate all messages, and (c) how will the messages be aggregated. It is, therefore, a *stateful* router because it needs to keep information between messages, only being allowed to proceed when all correlated messages have been received. Figure 2.9 shows the behavior described above.

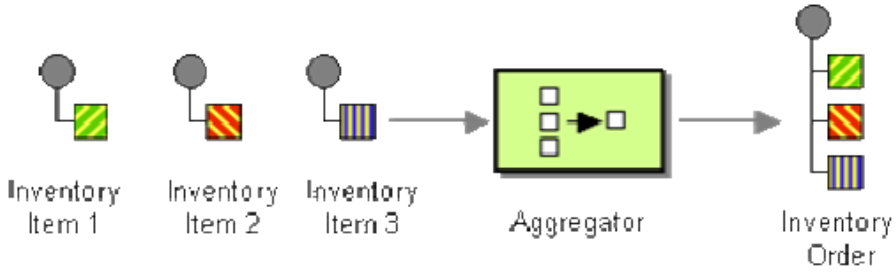

FIGURE 2.9: Behavior example of an *Aggregator* (Hohpe and Woolf 2004).

There are multiple strategies that can be applied in this aggregation process.

- "*Wait For All*": waits for all information to be received. It's the slowest option, but the best at making decisions;
- "*Time Out*": waits a certain time before making a decision with all messages received up until then;
- "*First Bet*": decides upon the first received message, and all others are discarded;
- "*Time Out With Override*": waits a certain time or until a message with certain presets has been delivered.

Resequencer

The *Resequencer* is a *Simple Router* that "puts out-of-sequence messages back into sequence", just as Figure 2.10 shows. Like the previous router, this one is also *stateful*.

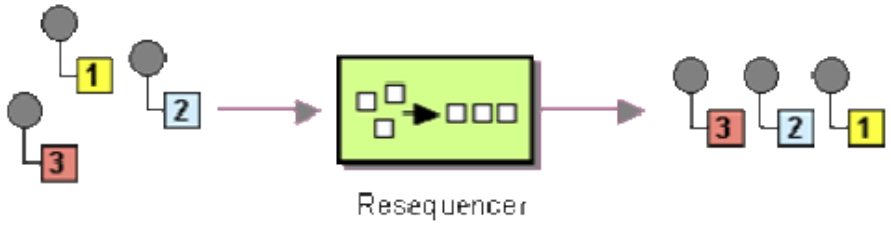

FIGURE 2.10: Behavior of a *Resequencer* (Hohpe and Woolf 2004).

Composed Message Processor

The *Composed Message Processor* is a *Composed Router* that combines multiple router variants to allow information retrieval from multiple sources and recombining it into a single message, while maintaining control over the sources inspected. Figure 2.11 shows the behavior of the components of this router. First a *Splitter* separates the different elements of a message, then a *Content-Based Router* routes them for proper processing, and finally an *Aggregator* joins the results.

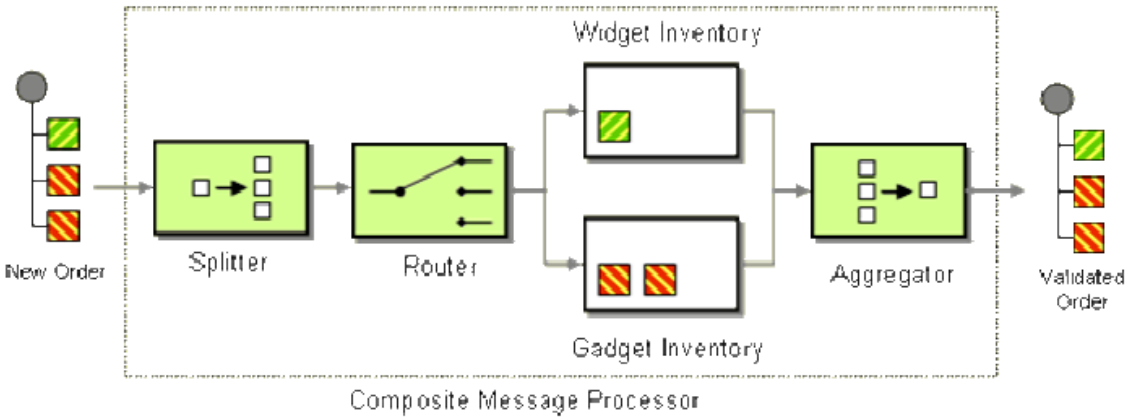

FIGURE 2.11: Behavior example of a *Composed Message Processor* (Hohpe and Woolf 2004).

Scatter-Gather

The *Scatter-Gather* is a *Composed Router* very similar to a *Composed Message Processor*, but, instead of using a *Splitter*, it broadcasts a message to multiple recipients and then aggregates them into a single one. Figure 2.12 demonstrates this described behavior.

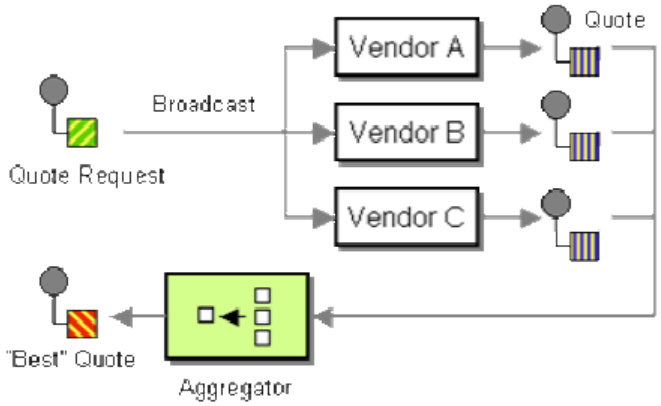

FIGURE 2.12: Behavior example of a *Scatter-Gather* (Hohpe and Woolf 2004).

2.3.3 Transformation Patterns

The patterns described in this section offer solutions to accommodate different data formats into a single, comprehensible one for the underlying system. The most important and relevant pattern here is the *Message Translator*, which is quite abstract. All other patterns are just variant implementations of the aforementioned one.

Message Translator

The *Message Translator* is a filter that allows different systems that make use of different data formats to communicate between them. This allows those systems to maintain a *loose coupling* between each other. Figure 2.13 shows the basic behavior of a message translator.

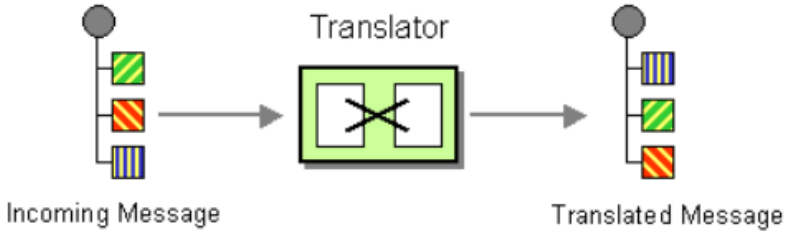

FIGURE 2.13: Behavior of a *Message Translator* (Hohpe and Woolf 2004).

Table 2.6 shows the layers at which a message translator can operate. The *Data Structures* layer acts alongside the domain model. It's responsible for the translation of business related entities and their relations. The *Data Types* layer deals with the metamodel of the business domain, and is responsible for the conversion of fields' metadata. The *Data Representation* layer is necessary in order to translate character or byte streams into character strings. The *Transport* layer is "responsible for a complete and reliable data transfer across different network segments and deals with lost data packets and other network errors."

TABLE 2.6: Action levels of a *Message Translator* (Hohpe and Woolf 2004).

Layer	**Deals With**
Data Structures	Entities, associations, cardinality.
Data Types	Field names, value domains, constraints, code values.
Data Representation	Data formats, character sets, encryption, compression.
Transport	Communications protocols.

Envelop Wrapper

The *Envelop Wrapper* wraps message payload data to be compliant with the service requirements. It allows the client to add header information (e.g., authentication, encryption) to the ongoing message (Microsoft 2009). Furthermore, multiple layers can be applied if necessary, just like an onion. Figure 2.14 illustrates this pattern behavior. This wrapper is directly related to the *Service Connector* pattern (see Section 2.3.4).

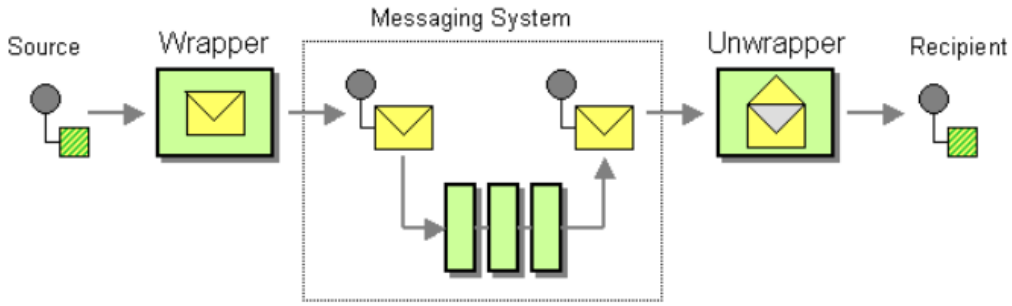

FIGURE 2.14: Behavior of an *Envelop Wrapper* (Hohpe and Woolf 2004).

Content Enricher

The *Content Enricher* is used when the system needs more data than that which is provided by a service. It can look up for missing information internally and externally, or

compute it from available data. Figure 2.15 shows the basic functioning of this transformer component.

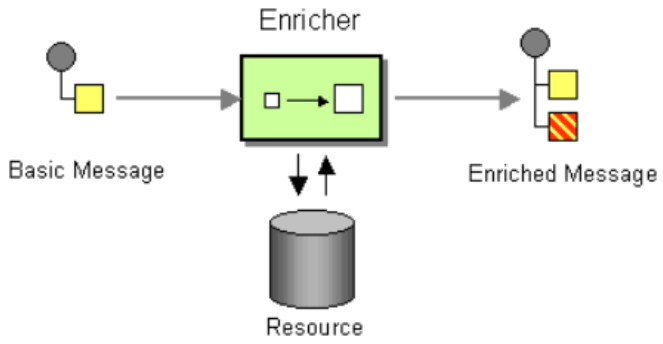

FIGURE 2.15: Behavior of a *Content Enricher* (Hohpe and Woolf 2004).

Content Filter

The *Content Filter* is the opposite to *Content Enricher*: it removes information from a message. Its purposes include removing unimportant information and simplifying the message structure. Figure 2.16 shows how removing data using this filter works.

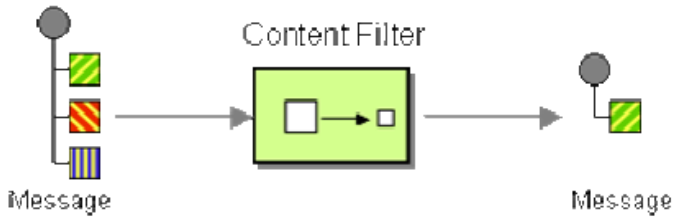

FIGURE 2.16: Behavior of a *Content Filter* (Hohpe and Woolf 2004).

Claim Check

The *Claim Check* acts like a *Content Filter*, but instead of throwing away the filtered out information, it stores it to be used later on. The persistence is made using a unique key that can be business related, message related, or self-generated. It allows to reduce the volume of data sent across the system without sacrificing content, and, less commonly, hide sensitive information. Figure 2.17 illustrates an example of how it works.

Normalizer

The *Normalizer* is a *Composed Transformer*, responsible for translating messages with different formats into a common one. It allows organizations to use a "common interchange format" when the incoming one is oblivious to the one used internally (Microsoft 2009). As depicted in Figure 2.18, this component uses a *Message Router* to route each message to the correct *Message Translator*.

Canonical Data Model

The *Canonical Data Model* is an independent model that provides an additional level of indirection between the formats used by all exchanging parties. It "defines message

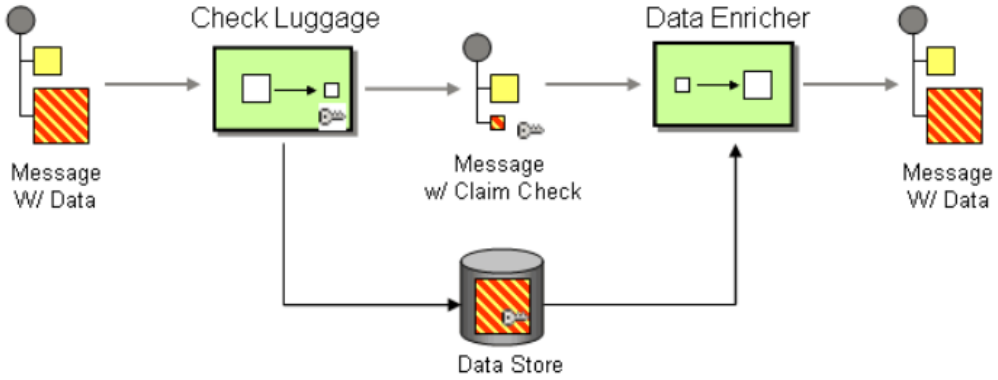

FIGURE 2.17: Behavior example of a *Claim Check* (Hohpe and Woolf 2004).

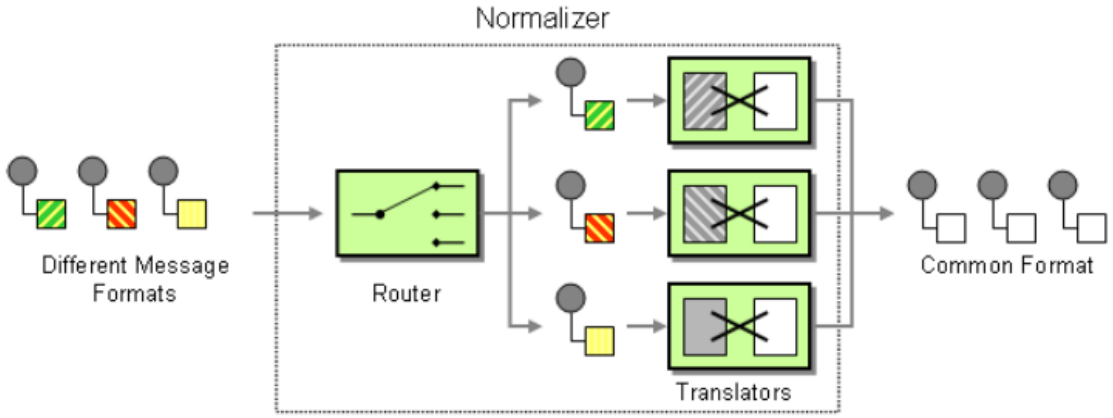

FIGURE 2.18: Behavior of a *Normalizer* (Hohpe and Woolf 2004).

formats that are independent from any specific application." The more applications participating in the integration process, the more the usage of this pattern pays off. Figure 2.19 shows the basic concept of this model and helps to understand the inter-relatedness it provides between the integrator system and providers (identified with letters from A to F). Using this model implies the usage of *Message Translators* to convert from one format to another, as Figure 2.20 demonstrates.

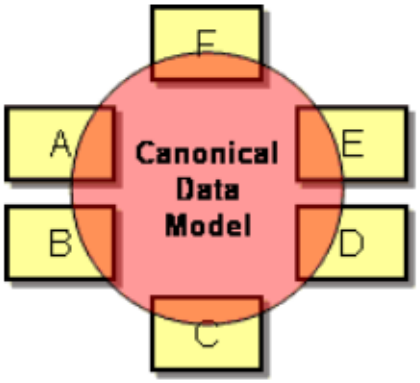

FIGURE 2.19: *Canonical Data Model* concept (Hohpe and Woolf 2004).

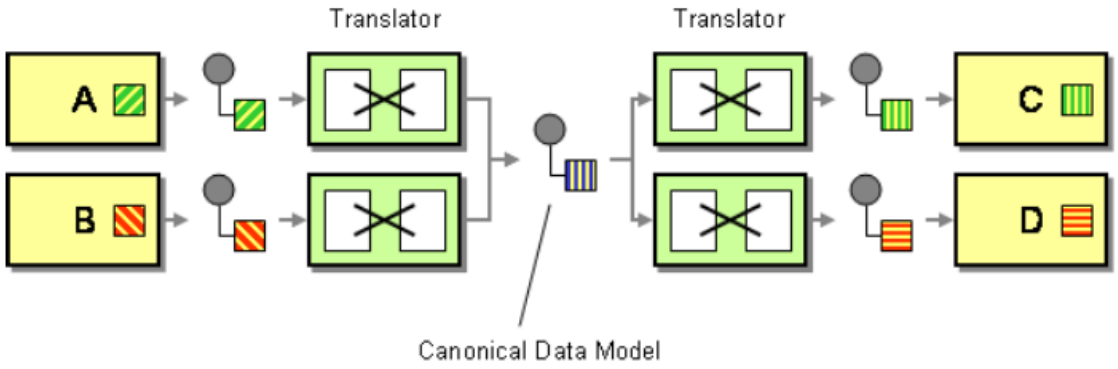

FIGURE 2.20: Behavior example of how a *Canonical Data Model* is used (Hohpe and Woolf 2004).

2.3.4 Endpoint Patterns

It's because of *Message Endpoints* that applications can communicate effectively with external systems. The *Service Connector* is the most crucial pattern here, which all ETL processes are required to have.

Service Connector

Service Connectors allow the client to "be insulated from the intricacies of communications logic" (Hohpe and Woolf 2004). This is achieved by encapsulating generic functions and including additional logic specific to each service. The client will always need to know how to access each and every service it requires information from, and using this pattern helps to isolate these types of concerns. Therefore, it is a "thin layer of code" that attaches the application to the external service, as demonstrated in Figure 2.21.

FIGURE 2.21: *Service Connector* component diagram (Daigneau 2011).

This component provides the following regulatory functions:

- Service location and connection management — the connector tasks include address discovery, connection establishment, and connection exception handling;
- Request dispatch — connectors must serialize requests into a stream bytes and then transmit it over the network using the appropriate HTTP method;
- Response receipt — the connector must also deserialize the response and capture its status code. It can implement a *Synchronous* or *Asynchronous Response Handler* (see Section 2.3.4).

Optional functionalities include:

- Retrieve or construct service URIs;
- Select and issue proper server method;
- Implement a *Message Translator* to convert data into the required format or into client domain structures;

- Implement an *Idempotent Retry* (see Section 2.3.4).

Service Connectors can be of two types: *Proxy* and *Gateway*. A *Proxy* provides "a surrogate or placeholder for another object to control access to it" (Gamma et al. 1995). *Service Proxies* encapsulate the connection logic, using higher level methods which call the desired service, making it look like a local function call when, in reality, a request is being sent to an external machine. They are commonly used with *RPC* and *Message APIs*. On the other hand, *Service Gateways* are typically created manually for *Resource APIs*. Here, the client is able to call services by executing methods on the gateway component. In addition, responses can be parsed to discover *linked services* and, with them, desired URIs constructed. Multiple *Gateways* can be chained together to reuse common behaviors (Hohpe and Woolf 2004).

Idempotent Retry

Clients should make every effort to guarantee that all requests are delivered and, therefore, should handle many types of connectivity problems. Moreover, if a service (a) might have down times, (b) might involve the use of alternative or intermediary services, or (c) makes use of message queues that work independently to the service itself, the client-service call itself must be protected:

> *"Networks are inherently unreliable. Connections will occasionally time out or be dropped. Problems will arise for innumerable reasons. Servers will be overloaded from time to time, and as a result, they may not be able to receive or process all requests. If a client can't connect to a service or loses a connection, or if the server reports that it is busy, sometimes the best solution is to simply try again"* (Daigneau 2011).

As demonstrated in the activity diagram from Figure 2.22, *Idempotent Retry* solves this problem by preparing calls to catch several connectivity errors. Whenever an error occurs, the client should retry the failed call up to a threshold that, when reached, further connection attempts must then be aborted. Idempotence can be defined as "a correctness criterion that requires the system to tolerate duplicate requests" (Ramalingam and Vaswani 2013). *Read*, *Update*, and *Delete* operations are considered to be idempotent, while *Creates* are not. This means that a client should not simply retry a *Create* operation upon network failure because it might end up with multiple objects that were supposed to be just the one.

Tolerant Readers

This pattern addresses the problem of the existence of unknown content or media types in messages. How can clients anticipate changes in service responses? The answer to this problem is to "design the client [...] to extract only what is needed, ignore unknown content, and expect variant data structures." To make this happen, declarative (e.g., XPath, JSONPath) or imperative (i.e., static or dynamic code) approaches can be used to only extract the desired data. Furthermore, *Tolerant Readers* should attempt to continue parsing a message even when violations are detected. With that in mind, exceptions should only be raised when (a) there a total impossibility to read the structure of a message, or (b) violations regarding business logic are detected.

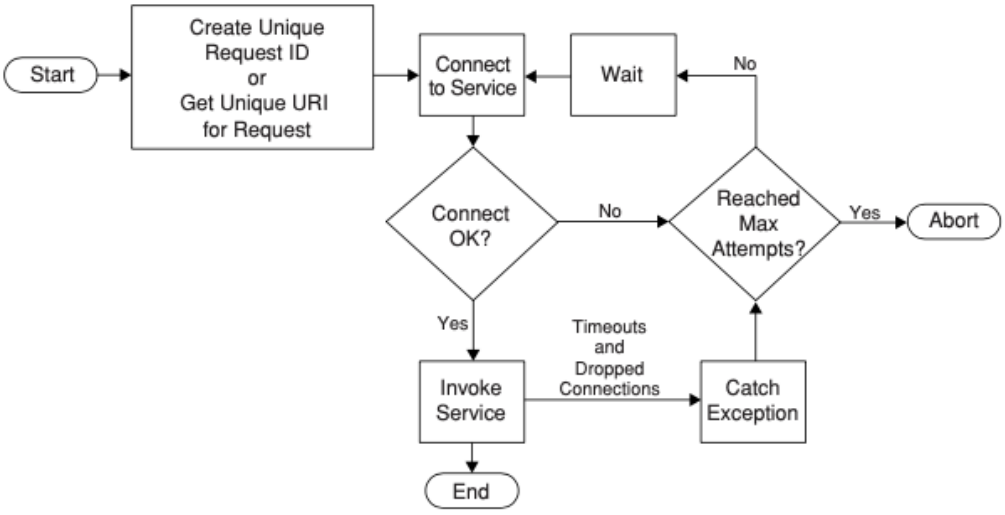

FIGURE 2.22: *Idempotent Retry* activity diagram (Daigneau 2011).

Request/Response

This is the simplest way to request and receive data from a service. This pattern should be used whenever the client requires immediate data processing, as demonstrated in Figure 2.23. Here, clients expect a response from each message sent to the service (Daigneau 2011; Microsoft 2009). As a side effect, this pattern introduces a temporal coupling together with the implications that (a) the service is obligated to be operational at all times and (b) a service might, at some point, be unreachable due to high capacity load.

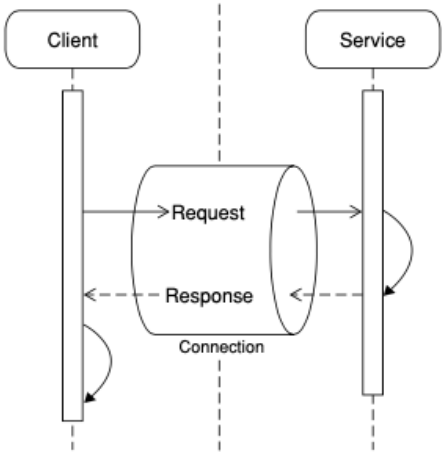

FIGURE 2.23: *Request/Response* sequence diagram (Daigneau 2011).

Linked Services

The easiest way for a client to acquire URIs is by using configuration files where all relevant addresses are stored. This, however, means that if a service changes its resources' locations, the client also needs to change. This can be avoided by querying a "central repository that stores metadata for service-related artifacts" (also known as *Service Registry*) to acquire root service addresses at the expense of decreased performance due to

the additional calls. Added to that, *Linked Services* can also be used for clients to discover "properly formatted URIs" that are adjacent to the most recent request specific to a resource. This will bring multiple benefits:

- All addresses are up-to-date and correctly formatted;
- Each request contains only relevant addresses;
- Clients are protected from server-side changes;
- Services can be discontinued more easily.

Asynchronous Response Handler

If the chosen approach to connect to a Web service is synchronous, the thread responsible for making the call blocks while waiting for a response. This waiting time is considered as a waste, and, when appropriate, it should be used for other purposes. As shown in Figure 2.24, the *Asynchronous Response Handler* enables the client to perform other tasks while it also waits for a response. However, one has to consider that this approach should not be used in operations that (a) are long-running, (b) might timeout, or (c) their response may be lost.

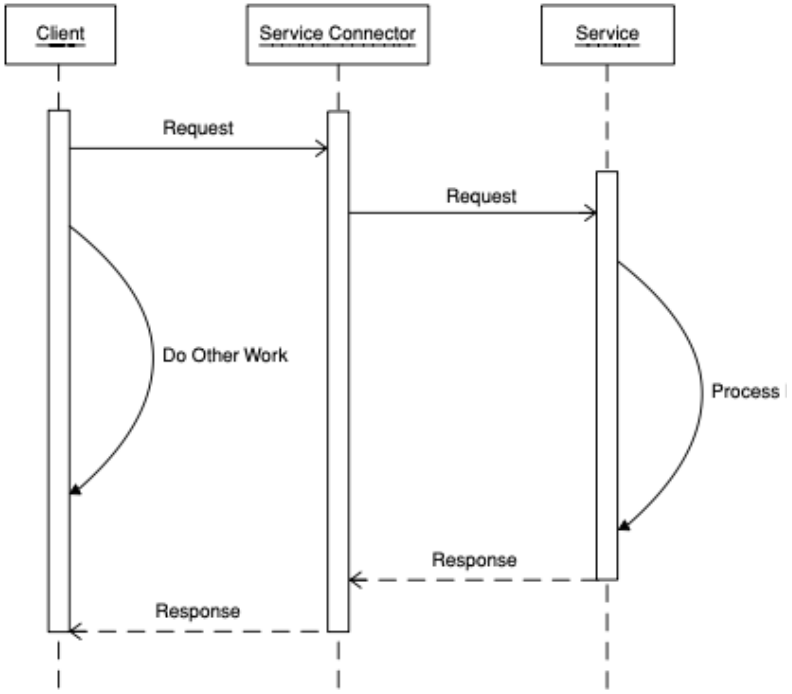

FIGURE 2.24: *Asynchronous Response Handler* sequence diagram (Daigneau 2011).

Messaging Mapper

A *Messaging Mapper* allows the conversion of communication formats into domain objects and vice versa, while keeping them independent from each other (Figure 2.25).

> *"Since the* Messaging Mapper *is implemented as a separate class that references the domain object(s) and the messaging layer, neither layer is aware of the other. The layers don't even know about the* Messaging Mapper*. [...] In*

most cases, the Messaging Mapper is invoked through events triggered either by the messaging infrastructure or the application. [...] If we have to invoke the Messaging Mapper *directly from the application, we should define a* Messaging Mapper *interface so that the application does at least not depend on the* Messaging Mapper *implementation"* (Hohpe and Woolf 2004).

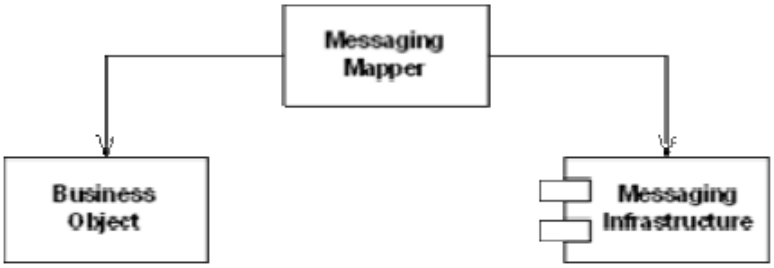

FIGURE 2.25: *Messaging Mapper* component diagram (Hohpe and Woolf 2004).

2.3.5 System Management Patterns

Integration solutions might be long-running processes triggered from time to time. Some patterns can be implemented in order to monitor, control, analyze, test, and debug it. This section describes such patterns.

Control Bus

The *Control Bus* connects components of an application to a central management console. Figure 2.26 illustrates the interface between a *Control Bus* and a core processing unit.

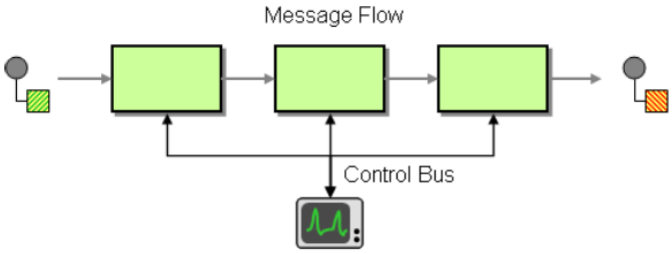

FIGURE 2.26: *Control Bus* (at bottom) interfacing with a core processing unit (on top) (Hohpe and Woolf 2004).

This component allows dealing with aspects related to:

- Configuration — rather than using local files to store configuration parameters, a *Control Bus* can be used as a "central point of configuration and [...] reconfiguration of the integration solution at run-time";
- Logging — a *Control Bus* can be used to store various kinds of logs (e.g., exceptions, number and types of messages processed, average processing time) for debugging purposes. Severe exceptions can also trigger a direct notification to the process operator;
- Monitoring — the *Control Bus* can also be used to display the described information in a central console. From here, operators can perform debugging.

Wire Tap

The *Wire Tap* allows one to inspect the contents of individual messages. Each incoming message is forwarded to a new recipient, just as Figure 2.27 shows. The *Control Bus* can be used to trigger the *Wire Tap* state, being turned on only when necessary (i.e., testing, debugging).

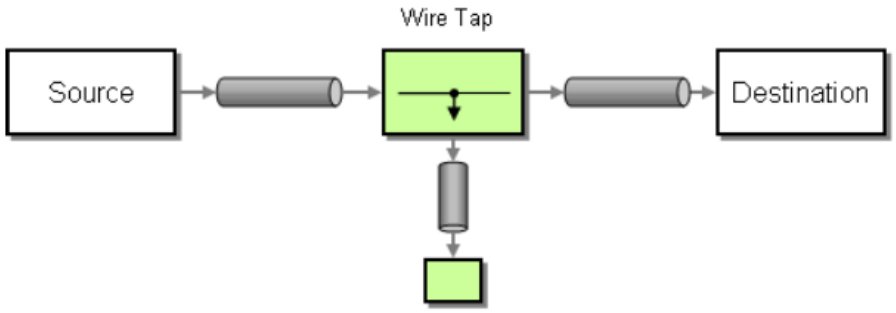

FIGURE 2.27: Behavior of a *Wire Tap* (Hohpe and Woolf 2004).

Detour

A *Detour* component provides the ability to route messages to additional, optional steps that are switched on and off due to performance-related issues. It's an alternative to the *Wire Tap* for when messages need to be rerouted or modified instead of just being inspected. As Figure 2.28 shows, the *Detour* component uses a *Context-Based Router* to route the traffic through a different channel with additional components. A *Control Bus* can be used to switch the traffic to the *Detour* and to specify with steps it should run.

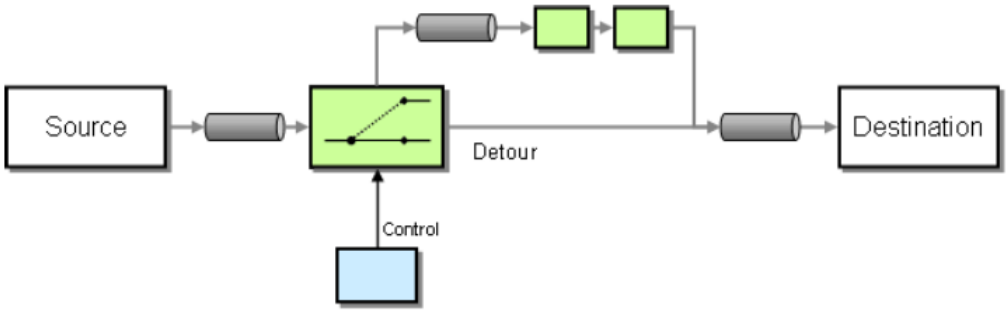

FIGURE 2.28: Behavior of a *Detour* (Hohpe and Woolf 2004).

Message History

Message History allows one to debug individual messages without introducing dependencies between components. It's a list of all the components a message passed through since its origination. As Figure 2.29 shows, each component tags the message with its identification. This pattern is most useful when messages flow through a series of filters in order to accomplish specific business functions. If a *Process Manager* is used, the need for *Message History* is mitigated.

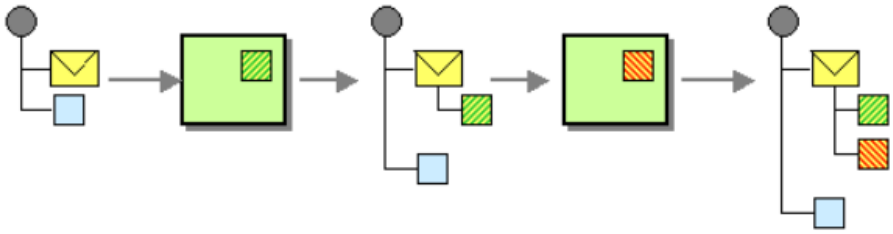

FIGURE 2.29: *Message History* pattern exemplification (Hohpe and Woolf 2004).

Message Store

A *Message Store* enables one to debug all messages that traveled through the system, capturing them in a central location. It allows to perform an effective reporting of the data that flows through the application. As Figure 2.30, each message is stored when it is created. This can be achieved by the component itself or by a *Wire Tap*.

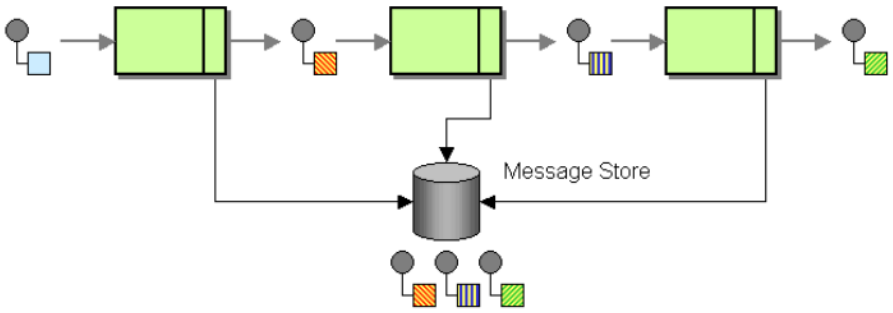

FIGURE 2.30: Behavior of how a *Message Store* component is to be used (Hohpe and Woolf 2004).

2.4 Technological Review

This section looks at the technologies available for use in the current project, considering that it requires the usage of Python (as is it will be stated in *Specification*, Section 3.3.2). First, it is explained why to use Python as a programming language and some concerns regarding it are described. Then, the existing, available and still maintained frameworks that can be used in ETL processes are explained.

Section 2.4.2 provides framework options that can be used in any phase of ETL processes, while tools in Sections 2.4.3 and 2.4.4 can be combined in order to provide a more customized ETL solution. The use of one doesn't imply that others cannot be used. It will all depend on the tools needed to achieve certain requirement goals. There are other frameworks that can be used (e.g., odo, Riko, circuitbreaker, requests), however their purpose is very fine-grained and, at this point, it's too early to make a decision if their usage will be profitable. Moreover, frameworks no longer maintained (e.g., mETL, Bubbles, Carry, Pinball) were completely disregarded due to the project's *robustness* and *integrity* requirements. If they existed, tools with a Graphical User Interface (GUI) would be also disregarded because of the difficulty and time-consumption in designing specific/complex ETL scenarios with the available graphical functionalities (Biswas, Sarkar, and Mondal

2019; Thomsen and Pedersen 2009). Furthermore, "any GUI-based tool cannot assure more productivity compared to code-based approach" (Biswas, Sarkar, and Mondal 2019).

2.4.1 Why Python?

In his book, Lutz (2001) describes Python as:

> *"[...] a general-purpose open source computer programming language, optimized for quality, productivity, portability, and integration. [...] Among other things, Python sports Object-Oriented Programming (OOP); a remarkable, simple, readable, and maintainable syntax; [...] and a vast collection of precoded interfaces and utilities. [...] Perhaps Python's best asset is simply that it makes software development more rapid and enjoyable."*

As described, its main strengths include efficient high-level structures, and a simple and effective approach to OOP. Furthermore, its "elegant syntax and dynamic typing, together with its interpreted nature," make Python an ideal language for scripting and rapid application development (G. V. Rossum and Python Development Team 2020). Having this in mind, one could say that Python enables better maintainability and faster implementation of desired solutions due to its inherent nature.

However, it comes with some downsides. One of them are the performance penalties due to the boxed representation of numbers as heap objects, and the late binding of function and method calls (Barany 2014; Drozd, Gladkova, and Matsuoka 2015). Moreover, Python provides a concurrent execution model based on multithreading in shared memory, enabling multiple computations to run on multiple threads. Yet, executing these threads simultaneously is not at all a standard feature of current Python implementations —contrary to the majority of mainstream languages— (Meier and Gross 2019). Because of this, it is expected that performance will take a hit while parallel processing. This is especially bad because ETL technologies need to integrate large amounts of data, which makes this process "extremely time-consuming," and parallel processing is key to solve the aforementioned issue (Liu, Thomsen, and Pedersen 2012).

So, why and why not Python in a nutshell:

+ Sports at OOP;
+ Simple, readable, and maintainable syntax;
+ Optimized for quality and productivity;
+ Vast collection of first- and third-party libraries/frameworks;
+ Automation;
+ Matured community;
− Performance penalties;
− Parallel processing is not that great.

2.4.2 General ETL Frameworks

Three frameworks were found that can be used in every step of ETL processes: *Bonobo*, *Mara*, and *Pygrametl*. Table 2.7 shows the main differences between all of these in a nutshell. It is possible to see that all of them are targeted at *simple* data integration, not

offering out-of-the-box features to validate, transform, and handle complex transformation processes.

Table 2.7: Features comparison of Python complete ETL frameworks.

Frameworks / Features	Bonobo	Mara	Pygrametl
License	Apache-2.0	MIT	BSD-2-Clause
Open-source	X	X	X
SOA-enabled	X		
SQL integration	X	X	X
Scheduling			
Complex transformations			
Data validations			
Good throughput handling			
Environment variables support	X		
Docker support	X		
Stable release		X	X
All major operating systems support	X		X
Web UI		X	
ETL jobs automatic documentation	X		
Concise documentation			X

Bonobo

Bonobo is an open-source framework that contains all the logic to "execute efficiently" an ETL process. It is recommended to be used with "small scale" data (as in "not big data"), and allows the usage of common file formats and basic services (Bonobo Development Team 2018a). "Developers can focus on writing simple and atomic operations, that are easy to unit-test by-design, while the focus of the framework is to apply them concurrently to rows of data" (Bonobo Development Team 2018b). In other words, "[i]t allows the user to focus on the content of the transformations, rather than worrying about optimized blocking, long operations, threads, or subprocesses" (Bonobo Development Team 2018c). Being still under heavy development, it cannot be considered stable until version 1.0 is released (currently at v0.7 since July 20, 2019).

Mara

Mara is another ETL framework, considered to be "lightweight", "opinionated", and "halfway between plain scripts and Apache Airflow" (Mara Development Team 2020). Its focus is in the reduction of transparency and complexity. Pipelines, tasks, and commands are created using declarative Python code. It can be extended with a Web User Interface (UI) tool to inspect, run, and debug pipelines.

Pygrametl

Pygrametl provides "commonly used functionality for the development of ETL processes," and is commonly used in production systems in different sectors such as healthcare, finance, and transport (Thomsen and Jensen 2021). The developer can focus on operations on the input data where "the power and expressiveness of a real programming language" can be used to "achieve high productivity" (Thomsen and Pedersen 2009). When convenient, some processes can be run in parallel with others (Thomsen and Pedersen 2011). It does not contain JavaScript Object Notation (JSON) or XML data sources as is, but it can be easily extended (Biswas, Sarkar, and Mondal 2019).

2.4.3 Workflow Management System Frameworks

Workflow Management System (WMS) tools enable developers to organize and monitor repetitive tasks. Tools like these are used to set up and run ETL workflows. Table 2.8 shows the main differences between two found WMS frameworks: *Apache Airflow* and *Luigi*. Although *Airflow* is more powerful and complex, *Luigi* also offers the minimum feature set required for its purposes while being simpler to use.

Table 2.8: Features comparison of Python WMS frameworks.

Frameworks / **Features**	**Apache Airflow**	**Luigi**
License	Apache-2.0	Apache-2.0
Open-source	X	X
Scheduling	X	
Highly configurable	X	X
Logging	X	X
Recurring failure handling	X	X
Out-of-the-box email notification	X	X
Docker image	X	X
Stable release	X	X
All major operating systems support	X	X
Web UI	X	X
Concise documentation	X	X
Simple to use and set up		X

Apache Airflow

Apache Airflow "is a platform to programmatically author, schedule and monitor workflows" (Apache Airflow Development Team 2020). It has three main components: Web Server, Scheduler, and Workers. The Scheduler executes tasks on an array of Workers for maximum efficiency, while following the specified dependencies. In addition, it also contain a UI in order to visualize pipelines running in production, monitor progress, and troubleshoot issues. It is widely used by hundreds of other organizations, therefore one can say that it is properly tested and stable. It is quite complex to use and commonly provides way more resources than those that are actually needed. If time and money are a restraint, it should be avoided.

Luigi

Luigi is a Python package that helps developers build complex pipelines of batch jobs. With it, it is possible to handle dependency resolution, workflow management, visualization, handling failures, command line integration, and many more (Luigi Development Team 2020). Its main purpose is to "address all the plumbing typically associated with long-running batch processes," by helping developers stitch many tasks together. At a concept level, Luigi is similar to GNU Make, where tasks are executed alongside their dependency tree. Just like *Airflow*, *Luigi* is widely used by all sorts of companies. *Luigi*, however, does not come with built-in triggering, and it requires some external tool to trigger workflows periodically.

2.4.4 Data Processing Frameworks

Although Python comes with a large set of tools out-of-the-box, there are some frameworks that were specifically designed for data processing. These include: *Apache Spark*, *Pandas*, and *Petl*. Table 2.9 summarizes the found features of the described frameworks. It's possible to see that all of them provide similar features regarding data transformation, although *Spark* makes integrating multiple sources difficult and *Pandas* has lower performance compared to the other two.

Table 2.9: Features comparison of Python data processing frameworks.

Frameworks / Features	Apache Spark	Pandas	Petl
License	Apache-2.0	BSD-3-Clause	MIT
Open-source	X	X	X
Data extraction from common formats	X	X	X
Multiple data types integration		X	X
SQL integration	X	X	X
Complex transformations	X	X	X
Data validations		X	X
Good throughput handling	X		X
Good performance	X		X
Stable release	X	X	X
All major operating systems support	X	X	X
Concise documentation	X	X	X
Simple to use and set up		X	X
Lightweight		X	X

Apache Spark

Apache Spark is an "unified analytics engine for large-scale data processing" (Spark Development Team 2020). *PySpark* provides a Python API to handle *Apache Spark* jobs. It has been developed in order to provide speed, ease of use, generality, and large amounts of data processing. Analyzing the documentation revealed that there is no clear way to integrate and transform multiple sources into a database. Furthermore, it revealed that using this tool might be over-reaching as it provides functionalities related to machine learning and data parallelism across multiple worker nodes in a cluster.

Pandas

Pandas is a "Python package that provides fast, flexible, and expressive data structures designed to make working with 'relational' or 'labeled' data both easy and intuitive" (Pandas Development Team 2020). Its most relevant features include: handling missing data and efficient `group-by`, `slicing`, `indexing`, `merging`, `joining`, and `reshaping` functions. It was developed to be primarily used as a data analysis tool, and, therefore, it's not built having performance in mind.

Petl

Petl is a "general purpose Python package for extracting, transforming and loading tables of data" (Petl Development Team 2019). It provides support for both OO and functional programming, and can handle a wide range of data source formats. The data flow needs to be synchronized with a pipeline, and does not support parallelism by itself (Biswas, Sarkar, and Mondal 2019).

Chapter 3

Analysis

This chapter analyses and details the main aspects of the current project so that a proper solution can be developed. Firstly, a business context is provided for the reader to gain some more insight about where the current project is inserted within ALERT®. Then, a value analysis of MIP is carried out, and, thereafter, its requirements detailed according to the FURPS+ classification model. Following that, an analysis of the to-be-developed solution is executed to detail the system context of MIP, the main logical components of the integration system, and what are some crucial functionalities MIP must exhibit.

3.1 Business Context

As previously described in Section 1.1, ALERT® is a complex healthcare product that integrates medication content into its services. This content is managed by external organizations, and evolves at an independent rate than that at which ALERT operates. This means that the current project is inserted in the *Business-to-Business* and *Data Replication* project types (described in Section 2.2.3). That's because data from external medication information suppliers needs to be integrated into the ALERT® product, and, in some cases, replicated across different systems. These suppliers provide and organize their data on their own way, while ALERT® has its own medication context approach. In order to properly integrate it, extraction, transformation, and loading tasks need to be successfully achieved. Extraction is achieved using Web services, which gives the current project a *Remote Procedure Invocation* integration style (described in Section 2.2.4). After this, the external information in the external format/vocabulary is converted into the ALERT® context, just as Figure 3.1 illustrates, and, in the end, loaded into the primary database of ALERT®.

3.1.1 Medication Ontology

As already explained in *Ontological Compatibility* (Section 2.2.5), ontologies are used to define "a dictionary of terms formulated in a canonical syntax" (Smith 2003). It is with that in mind that the domain model in Figure 3.2 has been created: it illustrates a core set of medication-related concepts in ALERT®. Its main purpose is to introduce the reader to the world of medication and is in no way exhaustive enough to depict all concepts inherent to medication. Some elements are based on the FHIR standard (described in Section 2.2.5).

Medication is a drug used to treat, cure, and prevent diseases. It is always associated with four main concepts: INN, Route, Pharmaceutical Form, and Ingredient. INN is one of the most important concepts here. As already referred in *Terminological Compatibility*

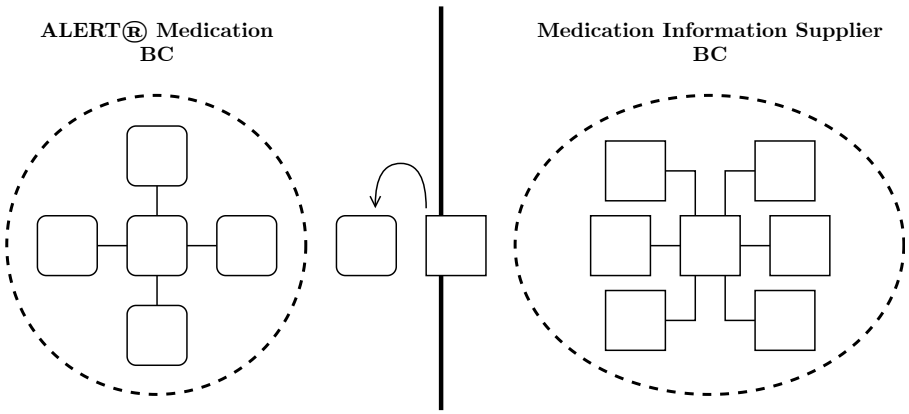

FIGURE 3.1: Bounded contexts of ALERT® medication.

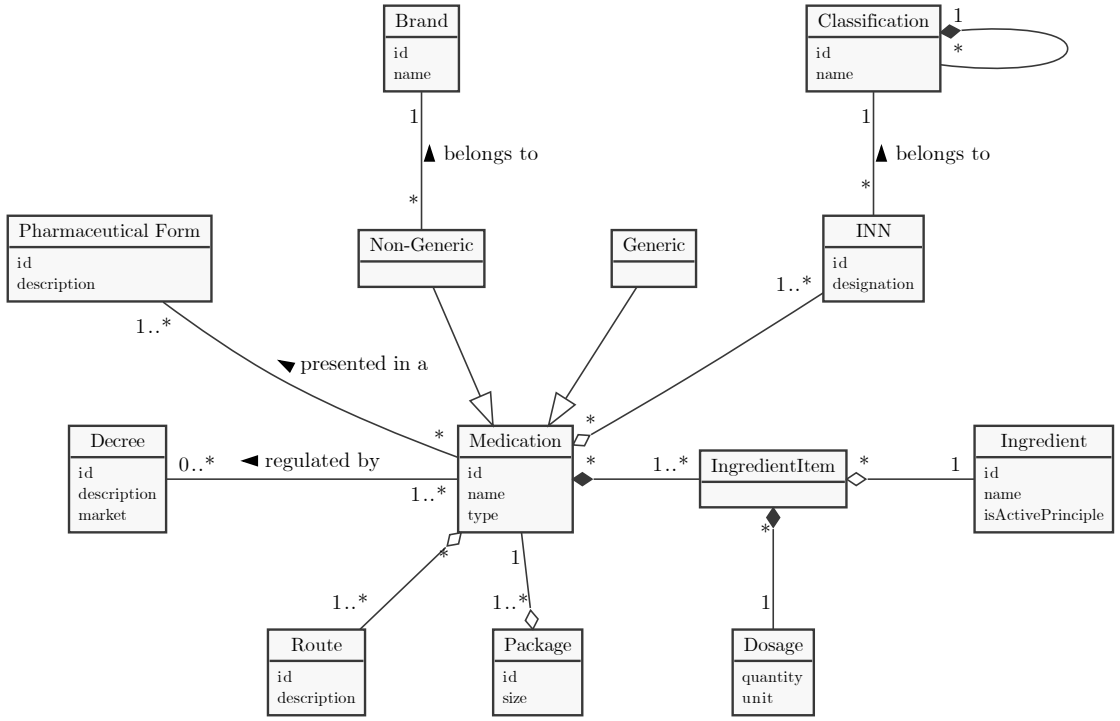

FIGURE 3.2: Medication domain model.

(Section 2.2.5), it allows the identification of pharmaceutical substances with a unique and globally identifiable designation. This means that medication can be prescribed in its generic or non-generic form. While generic medication is labeled with the respective INN, non-generic ones are always associated with a specific brand. Furthermore, based on the INN, one can classify to what classification group a medication is part of (e.g., anti-infectives, vitamins). Routes are the way medication can be administered (e.g., oral, intravenous). Pharmaceutical forms are the way drugs are presented and marketed for use (e.g., pill, cream). Ingredients are the chemical compounds of Medication drugs, which can vary from one to another in their dosage. The Dosage refers to the quantities and respective measure units in which a drug is available (e.g., 30 mg, 1 mg/ml). Besides that, Medication drugs can be grouped in packages that vary in size and regulated by specific decrees to control the consumption of certain substances.

3.1.2 Use Cases

In this section, every relevant Use Case (UC) of ALERT® that requires medication content is outlined. Figure 3.3 depicts them alongside their main actors. The four main use cases are to visualize (UC1), prescribe (UC2), administer (UC3), and dispense (UC4) medication. All of them require, in one way or the other, the visualization of the medication information imported into the system. UC4 includes validating prescribed medications (UC5) and, whenever necessary, replacing prescribed medications with valid alternatives (UC6).

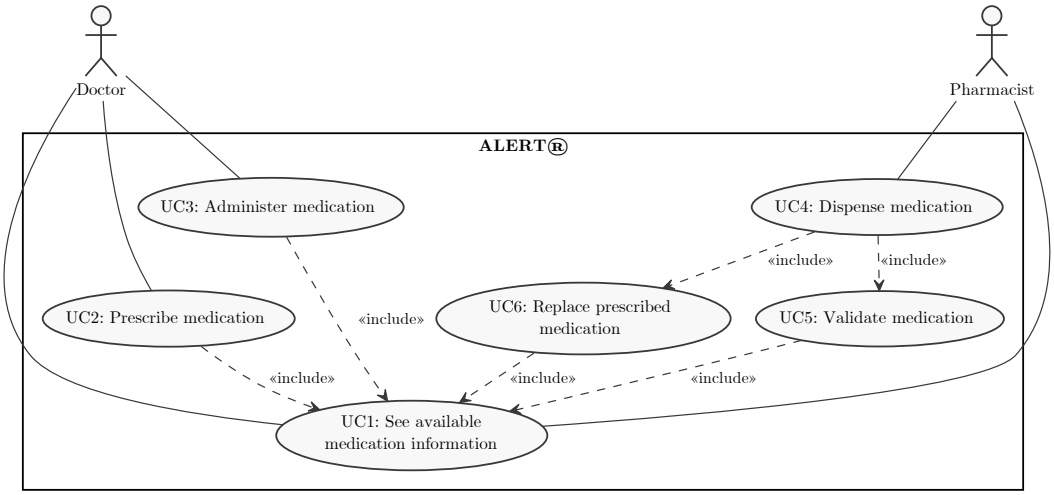

FIGURE 3.3: Medication-related use cases in ALERT®.

As one can infer, each use case requires different kinds of medication knowledge.

UC1: Lists vital medication information.

UC2: Uses information from UC1 to create a prescription.

UC3: Requires information of administration routes from UC1, and stores information about drug doses —relevant for both the doctor and the pharmacist in order to control the dosage administered.

UC4: Presents the necessary information so that the pharmacist knows exactly which drug to dispense.

UC5: Requires information from UC1 so that the pharmacist is able to validate that the prescribed drug is appropriate for the patient.

UC6: Requires information from UC1 so that the pharmacist is able to infer appropriate medication alternatives.

3.2 Value Analysis

The concept of *value* has been already touched in Section 1.9, where a *value chain* for the ALERT organization is provided in order to give the reader some insight of how the company is organized to generate *value*. The current section, on the other hand, focuses on the solution to be created in the current dissertation.

Whenever developing a new product, service, or process, a series of operations that transform *fuzzy* ideas into viable and ready to sell products are performed. This process of transforming an idea into a ready to sell product can be divided into three areas: the *Fuzzy Front End (FFE)*, the *New Product Development (NPD) process*, and *commercialization*. FFE is "generally regarded as one of the greatest opportunities for improvement of the overall innovation process" (Koen et al. 2004). It's here that the process of *Value Analysis* occurs:

> *"It is a disciplined action system, attuned to one specific need: to accomplishing the functions that the customer needs and wants [...]. In its disciplined thinking, value analysis is comprised of specific mind-setting, problem-setting, and problem-solving systems. These systems will assist anyone who has the task of providing more of what the customer wants for less cost"* (Miles 1972).

Over time, FFE has been receiving more and more attention in order to increase the *value*, *amount*, and *success* probability before concepts enter the development phase (i.e., NPD). In order to acquire some insight and common terminology in FFE, the New Concept Development (NCD) rationale can be used (Koen et al. 2004). The next 3.2.1 and 3.2.2 sections correspond to two (out of five) elements belonging to NCD:

- *Opportunity Identification* — enables organizations to identify opportunities they might want to seek;
- *Opportunity Analysis* — allows organizations to evaluate if the identified opportunity is worth pursuing.

3.2.1 Opportunity Identification

As previously discussed in *Problem* (Section 1.2), the healthcare IT market —where ALERT is inserted— is growing every year (MarketsandMarkets™ 2019). "Healthcare spending increases reflect longer life expectancies, advances in developing countries' standard of living and the corresponding ability to afford high-quality medical treatments, and technological advances that create new possibilities for curing diseases and delivering services" (Demirkan 2013). These healthcare expenses come with the promise of generally reduced healthcare costs, better workflows, and more efficient, error-free processes (Anderson 1997; Demirkan 2013). Furthermore, software technologies and frameworks are continuously evolving. Developers need to keep-up the pace with this evolution in order to provide relevant and exquisite products/services (Rajlich 2014). Web services together with ETL processes allow for information to be fluently transferred between distributed systems. This will concede any system from any business domain to integrate data into its own context.

ALERT, being a company dedicated to the provision of healthcare-related services, provides many functionalities in its product that are related to medication (as described in Section 3.1.2). In order for it to function as intended, it is necessary to integrate data from medication information suppliers. As a direct consequence of this, many items need to be imported into the system from time to time. Added to that, this data is not provided by a single supplier, which means that the extraction and transformation of information needs to be achieved for all relevant ones. Manually integrating the needed content in a trustworthy and efficient way is impossible. Thus, MIP, a software platform used to build integration processes responsible for integrating medication information into ALERT®, is

required. This will provide an increased value for the customer due to medication information being constantly complete, accurate, valid, and the most up-to-date as possible, which decreases the probability of the occurrence of human-related errors (e.g., wrong drug prescription, prescription of non-existing medication).

3.2.2 Opportunity Analysis

In order to assess the identified opportunity, the Strengths-Weaknesses-Opportunities-Threats (SWOT) strategic management tool will be used. SWOT is a key decision tool for "addressing complex strategic situations by reducing the quantity of information to improve decision-making" (Helms and Nixon 2010). As the name suggests, the SWOT analysis has four main elements: *Strengths* and *Weaknesses* evaluate the good and bad aspects regarding the project itself, and *Opportunities* and *Threats* evaluate the good and bad external factors regarding the market, the company, competitors, suppliers, and many others. Figure 3.4 shows this assessment for the to-be-developed solution: MIP.

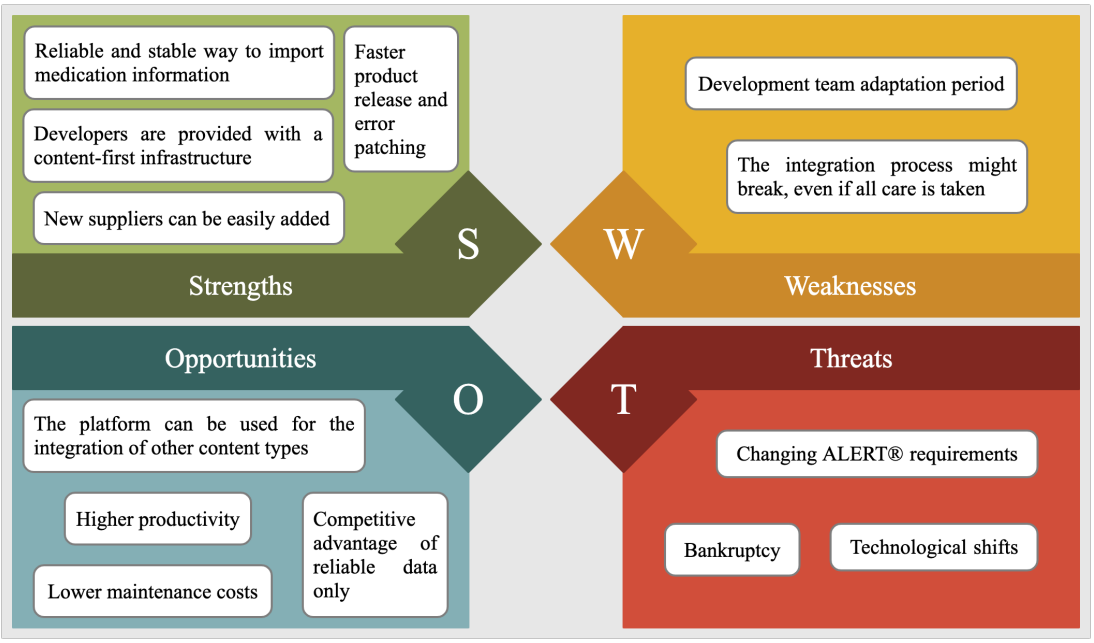

FIGURE 3.4: SWOT Analysis of MIP (template extracted from Creately).

From an internal point of view, MIP's strengths are the ones evidenced in the opportunity identification. Its main benefit is that it will provide a *reliable* —as in trustworthy—, and *stable* way to import medication information into the final product. With a *content-first* integration approach that focuses on what content needs to be integrated, rather than prioritizing how the process should be done, new processes can be built in a systematic fashion. Furthermore, if built appropriately and following good development practices, software *evolution* and *maintenance* will be protected. This means that a common highly maintainable code platform, from where all the suppliers integration processes are created, will (a) reduce the initial development effort whenever the need to integrate for a new one arises, and (b) allow for a faster detection of unforeseen errors. This will cause a faster *product release* and a faster *error patching*, which further increases the client's satisfaction level. A side effect of this is that the development team will need some *adaptation period* to be familiarized with the platform, but this is a low-level weakness as it occurs with almost

every new programming tool. Another disadvantage is inevitably inherent to distributed computing. As ALERT is not in control of the external services, if suppliers introduce *breaking changes* into their services, the integration process might *break* (although the strength of *fast error patching* helps to alleviate this point).

From an external point of view, the platform can be used for the integration of *any content type* if appropriately abstracted. This means that if the platform uses high-level concepts regarding ETL processes and Web services, it will be also capable of integrating other types of content (e.g., diseases, healthcare-related classifications). In addition, a completely automated process leads to *lower maintenance costs* and *higher productivity*, as developers spend less time worrying about how and when to trigger integration processes, and can spend that time in new and more prolific tasks. With this integration approach, new medication services are integrated faster and only reliable —as in complete, accurate, and valid— data is imported into the final product, which can turn out to be a *competitive selling point* for ALERT. Some minor threats were identified that have a low-chance of occurrence. If MIP's abstracted framework was turned into an open-source tool (and properly maintained thereafter), *bankruptcy* and *changing requirements* could be withdrawn as threats. *Technological shifts* are related to the evolution of new software. Many years ago, Web services were not used for distributed programming and today they are a standard for it. If in some years in the future a new technology surfaces that alienates the information extraction and transformation process, MIP could very well become legacy software.

3.2.3 Value for Developers and Customers

In the above *Opportunity Analysis* section, one can infer that value is perceived in two different ways between developers and customers, which goes according to the work of Lindgreen and Wynstra (2005) that refers that "value to the producer means something different from value to the user." In the current dissertation, this is even more emphasized considering that MIP is a tool for developers that will help them to provide a better service to the ultimate customer. To better explain and demonstrate this, *value-based drivers* are next described for both developers and customers. These value-based drivers are variables that affect the value of a product, service or company, and can be perceived upon three scopes (Lapierre 2000): *product*, *service*, and *relationship*. Each scope contains a set of inherent drivers that increase or decrease their perceived value.

In this project, developers best regard value within the *service* and *product* scopes, and see it from a technical perspective. Here, MIP's main benefits and sacrifices are the ones evidenced in the SWOT analysis and can be resumed to:

+ Increased data veracity and accuracy;
+ Stable integration process;
+ Integration processes are built *systematically*;
+ Integration processes are built with a *content-first mindset*;
+ Faster product release and error patching;
+ Improved productivity;
− Adaptation period;
− No possible protection from the so called *breaking changes*.

For the customers that will benefit from this project, value is regarded in all scopes and seen from a business perspective. Using the key drivers of customer-perceived value identified by Lapierre (2000), the following list identifies the main benefits of MIP in the eyes of the costumer:

+ Increased *quality of the product* by guaranteeing that medication-related modules contain only viable information;
+ Increased *responsiveness* of the service provided, considering that MIP enables for a better maintenance of integration processes, enabling for errors to be fixed more rapidly;
+ Increased *technical competence* by using/creating technologies that improve the business;
+ All of this also contributes to increasing the company's *image* and *confidence* level.

A common sacrifice, however, emerges for both developers and customers alike: there is a great deal of initial *effort* and *time* spent to develop MIP. But once this is overcome, its benefits can be fully exploited.

3.2.4 Value Proposition

A value proposition "defines the specific strategy to compete for new customers or increased share of existing customer businesses" (Jalili and Rezaie 2010). Figure 3.5 illustrates the following value proposition statement of MIP:

> MIP is a content-first platform that allows developers to systematically create integration processes and, therefore, incorporate medication content into their own application context, through a highly automated, reliable, and maintainable process. This ensures data integrity, while reducing the initial development effort and the occurrence of failures. As a result, it allows greater productivity, faster product releases, higher content quality, and more consistent information across the market, thereby rising the client's level of satisfaction.

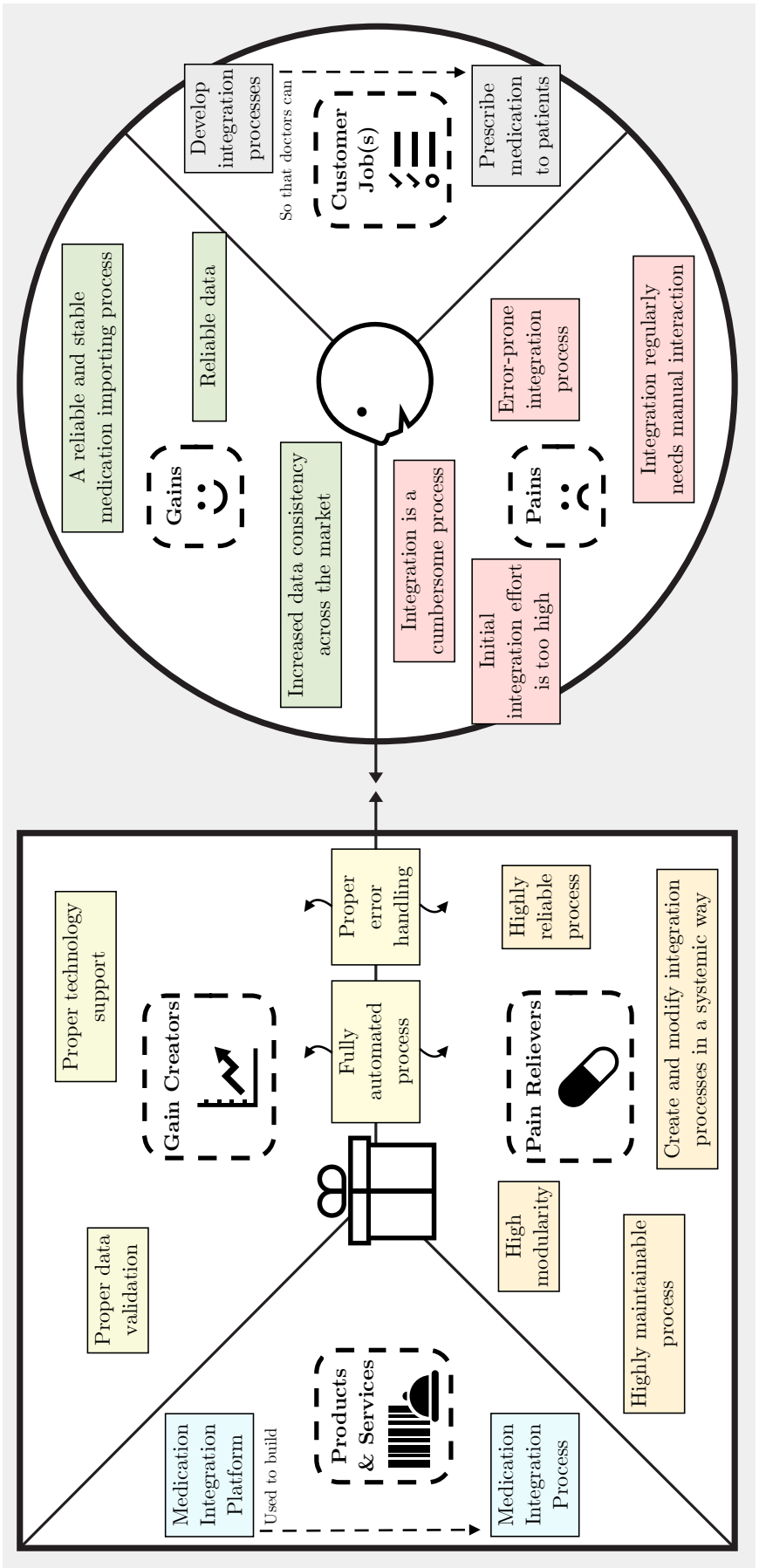

FIGURE 3.5: Value proposition of MIP (template extracted from Strategyzer).

3.3 Requirements

Initially, the requirements of this project were somewhat vague. Thus, there was the need to detail these further in order to be able to properly construct the desired solution. In an initial phase, an elicitation process was conducted. After that, the requirements were specified, detailed, and validated.

3.3.1 Elicitation

The stakeholder for MIP is any healthcare provider that uses the ALERT® EMR product. Having an automated medication integration process that updates the medication contents in a trustworthy and almost risk-free way, will improve the quality of the services provided to the client. The need for this specific integration process surfaced a long time ago, and, therefore, it has already been studied by the content team. Because of that, there is no need to meet directly with the client. Instead, the Business Expert, which is the Content Team Leader, was used to acquire the requirements of this specific project. In order to detail them further and to gather all the needed information, various methods were applied:

- Meetings with the Content Team Leader;
- Documentation analysis (from the medication information supplier and ALERT);
- Analysis of ALERT®;
- Analysis of the external service API and respective medication data;
- Functional analysis of the existing integration processes.

The analysis of ALERT® consisted in analyzing the use cases of the final product (outlined in Section 3.1.2) that are directly related to medication content. Additionally, the medication content approach of ALERT® has been also reviewed, which includes dissecting database schemas, packages, procedures, and so on. The external service API and respective content was also thoroughly analyzed. This includes the actual data itself in the external format provided (i.e., XML), and how to retrieve such data (i.e., HTML requests). Regarding existing integration processes, these have been attentively elucidated by meetings with the Content Team Leader, by the documentation, and, subsequently, by its code analysis.

Additionally, special care was taken in order to know what requirements were important for ETL processes. For this purpose, the QoX metric suite was used:

> *"The purpose of the QoX metric suite is to capture all relevant quality metrics for an ETL engagement and incorporate these into an optimized design and implementation. By incorporating these metrics at all levels in the design process, the resulting implementation will better match the customer's expectations and objectives. This will reduce the time and cost of ETL projects"* (Simitsis et al. 2009).

In other words QoX uses a layered approach, that proceeds in "successive, stepwise refinements from high-level business requirements, through several levels of more detailed specifications, down to execution models" (Simitsis et al. 2009). This means that the initial set of requirements might not be the final ones. According to Dayal et al. (2009) and Simitsis et al. (2009), the central set of metrics it uses for ETL processes includes:

- *Affordability* — "ability to maintain or scale the cost of an ETL process";
- *Auditability* — ability to "provide data and business rule transparency";
- *Availability* — capability of the process to be operational whenever required;
- *Flexibility* — ability to accommodate new or changing requirements;
- *Freshness* — capability of the system to provide a desired latency to its data;
- *Maintainability* — "ability of an ETL process to be operated at the design cost and in accordance with service level agreements";
- *Performance* — time taken to execute an ETL workflow, which can be affected by the memory made available;
- *Recoverability* — capability to resume after interruption and restore the state of the process;
- *Reliability* — ability to perform the intended operation, during a specified time period under certain conditions;
- *Robustness* — "ability of an ETL process to continue operating well or with minimal harm, despite abnormalities";
- *Scalability* — "ability of an ETL process to handle higher volumes of data";
- *Traceability* — "ability of an ETL process to track the lineage (provenance) of data and data changes".

In addition to QoX, it is also important to know how to infer the quality of the integrated data. For this purpose, the dimensions of QoD were gathered from the work of Akoka et al. (2007):

- *Data accuracy* — correctness and precision with the real data;
- *Data completeness* — degree of which all relevant data is present in a dataset;
- *Data consistency* — degree of which a dataset satisfies a set of integrity constraints;
- *Data freshness* — degree of how old a dataset is;
- *Data uniqueness* — degree of which two or more values do not conflict with each other within a dataset.

All the information and requirements gathered from this elicitation process are described in the next *Specification* section.

3.3.2 Specification

As already pointed out in *Problem* (Section 1.2) and *Objectives* (Section 1.3), integration processes for ALERT® medication are divided in two sequential processes: medication ETL and importation. MIP, as a healthcare IT solution for reliable medication integration, must be developed in such a way that supports both of them. Please refer to *System Context* (Section 3.4.1) and *What is the Medication Integration Platform?* (Section 5.1) to better understand the MIP system and what it actually is. To better distinguish what MIP needs to deal with, next follows a list of its every Functional Requirement (FR) in sequential order regarding overall integration processes according to the ALERT® requirements:

1. Extract, transform, and load data from external services into a staging area (*FRETL*);
   (a) *FRE* — Extract medication content from a single external service;
   (b) *FRT* — Transform the external data according to predefined domain rules;
   (c) *FRL* — Load this data into a local staging area;
2. Import the resulting data from the staging area into the primary database (*FRI*).

However, as already described in *Objectives* (Section 1.3), this dissertation's main efforts are aimed at the first subprocess (i.e., *FRETL*). With this in mind, the requirements of the current project can be divided into two working sets: MIP and actual ETL processes, where ETL processes are built using MIP. It should also be duly noted that MIP's requirements must respect and enable the requirements of full integration processes to be properly implemented. This means that, for the current project, MIP must only allow for the importation process to be easily refactored into the platform in the future.

Besides the functional requirements described above, a few other specifications were gathered. Tables 3.1 and 3.2 present these supplementary specifications using FURPS+. Successfully answering all specifications will also allow to successfully solve all issues described in *Problem* (Section 1.2). Later on, in the *Evaluation Approach* (Section 7.2), criteria for the evaluation of these requirements will be thoroughly detailed.

Regarding the platform itself and taking into account that it is a software development tool, one can see that its supplementary specifications have a higher focus on systematically and easily allowing the creation and modification of integration processes. With that in mind, MIP should be highly extensible (*MDRE*) and modifiable (*MDRM*) for it to be *flexible*, which allows (a) a better interfacing accommodation with external systems, and (b) a better reaction to software evolution. This way, developers will be able to (a) better respond to client-side modifications, (b) introduce new information suppliers easily, and (c) insert/modify desired/existing behaviors in integration processes. As a tool that will be used by many people —not all of them software developers—, MIP should facilitate its usage (*MUEU*). This means that it should allow for one to easily develop new software structures and leave no place for misinterpretation. Integration processes will be executed in the clients' machines, which means that it needs to be able to run in all relevant environments, namely Unix and Windows systems (*MSA*). Another relevant aspect is that MIP needs to be highly configurable (*MSC*), considering that (a) some variables will change from time to time or from client to client, and (b) some other variables contain very specific logic that must be set via configurations for easy access. Because of that, MIP must contain four different groups of configurations: core, process, service, and email. In order to improve maintainability and facilitate extensibility and modifiability, Python is the demanded programming language (*MImpRPL*). Added to that, MIP should provide a way for processes to be triggered either manually or by schedule (*MFO*).

With regard to concrete ETL processes, these should try their best to be executed without triggering critical errors (*PRR*), which includes properly handling timeouts, HTTP error codes, and many other runtime exceptions. However, not all failures can be prevented — such as when breaking changes are introduced into the external service. When such critical failures occur, an email notification (*PFN*) should be triggered and the next scheduled run should still be carried out (*PRSur*). In order to efficiently solve this kind of problems related to the described errors, logs should be kept to easily pinpoint where the error occurred and what caused it, consequently increasing the maintainability of the platform

Table 3.1: MIP supplementary specification.

FURPS+ Category	**Requirement**	**Description**
F	*MFO* – Operation	Processes can be scheduled or run on demand.
U	*MUEU* – Ease of use	As a platform from which integration processes are built, MIP should facilitate its use.
S	*MSA* – Adaptability	Each process should be able to run in different environments (i.e., Unix, Windows).
	MSC – Configurability	MIP must contain configuration modules for core components, for each process, for each service, and for email notification.
+ (DR)	*MDRE* – Extensibility	MIP should be easily extensible to new information suppliers, without modification of any other already developed component.
	MDRM – Modifiability	MIP and its processes should be able to be modified efficiently.
+ (ImpR)	*MImpRPL* – Programming Language	Python.

Legend:
(DR) – Design Restrictions
(ImpR) – Implementation Restrictions

(*PSM*). Besides errors, logs should also be kept with all additional relevant information (*PFA*), which includes execution times, types of information fetched, and invalid data discovered. Moreover, the ETL process must guarantee that only valid information (*PRI*) is integrated into the staging area. Besides that, the process will need to handle large amounts of data (*PPT*), but not as much as in big data. A rough estimation is that it will need to process one hundred thousand entries per execution in each instance. The memory used by the process, however, is not a major drawback because as long as ETL processes do not require *extremely* large amounts of memory, the memory made available can be very easily expanded. Furthermore, it would be good to not take over thirty minutes of execution time (*PPF*). Regarding the staging database to be used, SQLite (*PImpRDB*) was the demanded engine for data to be stored in a canonical internal format. Additionally, accesses to this local database should be controlled (*PRSec*). Lastly, in this particular case, the communication with the service provider will be via REST using an XML format (*PIntAPI*). The service also provides a SOAP API, however it was decided to use REST because it adheres very closely to the concept of resources and is able to take advantage of simplistic CRUD operations through HTTP (as described in *Communication Standards* of Section 2.2.6).

3.4 Solution Analysis

This section is dedicated to the analysis of the to-be-developed solution. Firstly, a context regarding MIP is provided in order to analyze the solution in more depth. This is done by conceptually describing MIP. After that, the solution is described in a more software-specialized fashion. Finally, a functional analysis is conducted in order to detail what are all the required functionalities of both MIP and ETL processes.

Table 3.2: ETL processes supplementary specification.

FURPS+ Category	Requirement	Description
F	*PFA* – Auditability	Store relevant information about carried out processes.
	PFN – Notification	Notify on critical failure.
R	*PRI* – Integrity	Extracted data should be complete, unique, consistent, and accurate at all times. Invalid data should be persisted with an `INVALID` state.
	PRR – Robustness	Processes should try their best to perform its functions without errors/failures.
	PRSec – Security	Database access should be controlled.
	PRSur – Survivability	If a failure occurs, the next scheduled run should still be performed.
P	*PPF* – Freshness	Ideally, ETL processes should take less than 30 minutes. More than one hour is too much.
	PPT – Throughput	The average number of entries that need to be extracted per process is around one hundred thousand.
P	*PSM* – Maintainability	Sources of runtime errors should be easily pinpointed and fixed using logs.
+ (ImpR)	*PImpRDB* – Database	SQLite.
+ (IntR)	*PIntRAPI* – API	REST/XML.

Legend:
(ImpR) – Implementation Restrictions
(IntR) – Interface Restrictions

3.4.1 System Context

Here, the *System Context* viewpoint from the *C4* architectural view model is illustrated and detailed so that the reader can properly comprehend the solution from a coarse-grained level.

Figure 3.6 depicts a conceptual model of MIP. It can be seen that MIP requires the participation of both Medication Information Suppliers and ALERT® in order to properly carry out its integration tasks. On one side, the Supplier provides the required external information that ALERT itself cannot create. This data will be then transformed and loaded into the other participating system: ALERT®. Here, the extracted and transformed information must be guaranteed to be valid, complete, and accurate at all times. With that in mind, it should be duly noted that MIP uses the *LAV* integration approach, where the content of each data source is specified in terms of the internal schema. Please refer to *What is the Medication Integration Platform?* (Section 5.1) for a more in-depth description of MIP.

Logical View

Until here, *System Context* detailed and explained the concept of MIP. Now, attentions are shifted towards a more technical perspective where the solution is described with a logical view of MIP. With that in mind, Figure 3.7 depicts a component diagram of the system where MIP is inserted.

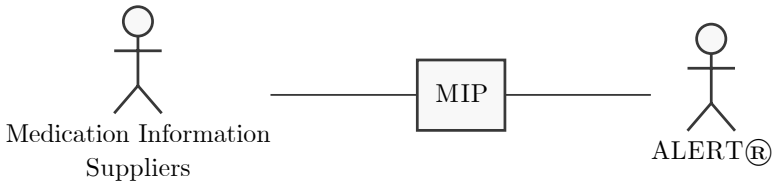

FIGURE 3.6: High-level model of MIP (Level 1 of C4).

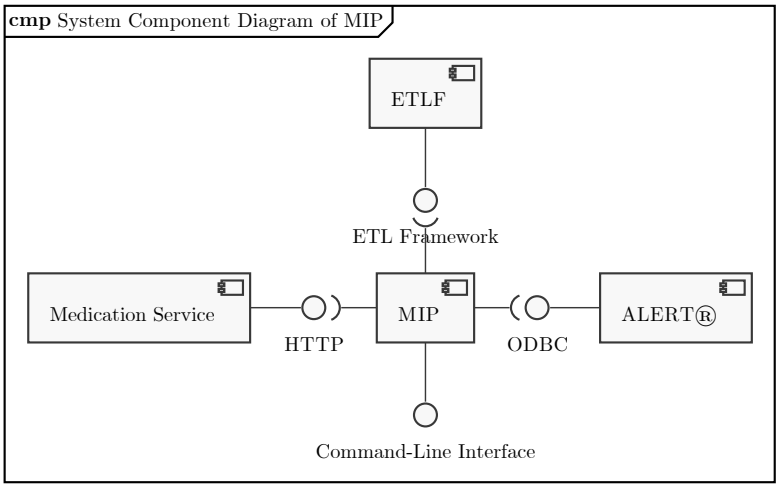

FIGURE 3.7: MIP system component diagram (Level 1 of C4).

As one can see, MIP provides an interface for triggering integration processes via command-line. Besides that, it interacts with the medication service via HTTP and with ALERT® via Open Database Connectivity (ODBC) to, respectively, extract and persist the external content. Additionally, it also requires *ETLF* —another major asset of this dissertation. Chapters 4 and 5 will further explain ETLF and MIP, respectively. For now, what's important to refer is that ETLF is an ETL-specific software framework that allows developers to systematically create sustainable and reliable ETL processes.

The most defying problem for MIP to resolve is the creation of a sustainable software platform that aims at the integration of data from an external format into an internal one. This is specifically daring in ALERT® because "[c]ode that combines two bounded contexts gets convoluted because it's mixing two separate vocabularies together" (Fowler 2015) and *any* unsound data can put lives at risk. Besides this, there are other finer-grained problems that need to be resolved. One of them is how to generate IDs that are unique across all ALERT® systems deployed on all customer servers and that can be reused across all of them, without communicating with a central database system. Another anticipated problem is that data must be properly handled while it is in memory. This is because, if data structures are not properly managed, they can quickly grow to unbearable sizes considering the ETL nature of the process.

3.4.2 Functional Analysis

This section provides a Functional Analysis (FA) of the ETL solution to be developed. It describes the functionality of the system at a finer granularity level compared to the requirements' specification of Section 3.3.2. To accomplish this type of analysis, a Function

Analysis System Technique (FAST) diagram has been produce and depicted in Figure 3.8. FAST diagrams are tools that "provide a graphical representation of how functions are linked or work together in a system (product, or process) to deliver the intended goods or services" (Borza 2011). Applying FAST to a problem helps one to focus on what is truly important, leading to a clearer path to achieving a solution for the given problem. In other words, FAST helps to guarantee that the final solution achieves all of its needs and there are no unnecessary, duplicated, or missing functions. This means that all illustrated functionalities must be present —or allowed to be present— in the final solution.

When designing the solution, some important supplementary specifications must be taken into account. *Robustness* —as in "no uncaught errors"— is needed in order to *fully automate the process*, which also needs to be carefully designed taking into account that it needs to *handle many entries* and be somewhat *efficient*. Furthermore, in order to ease *software evolution*, it needs to be easily extensible and modifiable. To do so, creating *modular components* will help immensely to achieve such goal. To ensure maintainability, all steps need to be *logged* and *errors handled properly*. If a critical error occurs, someone should be notified immediately. Moreover and considering the nature of the current project, *data integrity* must be safeguarded at all times. This can mean many things, but, ultimately, the integration system cannot compromise the final state of neither the staging area nor the database.

Regarding key functions, it's possible to identify four basic ones: *Background Execution*, *Receive*, *Convert* and *Persist Medication Data*. *Executing the Process in Background* results from the need to continuously integrate medication content from a provider that changes its content from time to time. Therefore, the process should be triggered according to some kind of property defined in a configurations file or executed manually if needed. Python scripts allow this task to be achieved quite easily.

The other three basic functions (i.e., *Receive*, *Convert* and *Persist Medication Data*) are all related to data manipulation in a common ETL process. Although the current process makes use of a REST service, MIP must easily allow for either a REST or SOAP API to be used. Therefore, distinct HTTP connectors must be developed in order to access such services. These external calls have to be protected from any kind of error related to distributed computing and from any kind of server-side failure. After receiving the external data, the process must then *convert* it into an internal format. This means that both contexts need to be mapped and information parsed. While parsing, it is crucial to (a) filter the necessary information out, (b) ensure the incoming relevant information is coherent, and (c) enrich the data with supplementary information. Lastly, the converted information needs to be *persisted* into a local staging database.

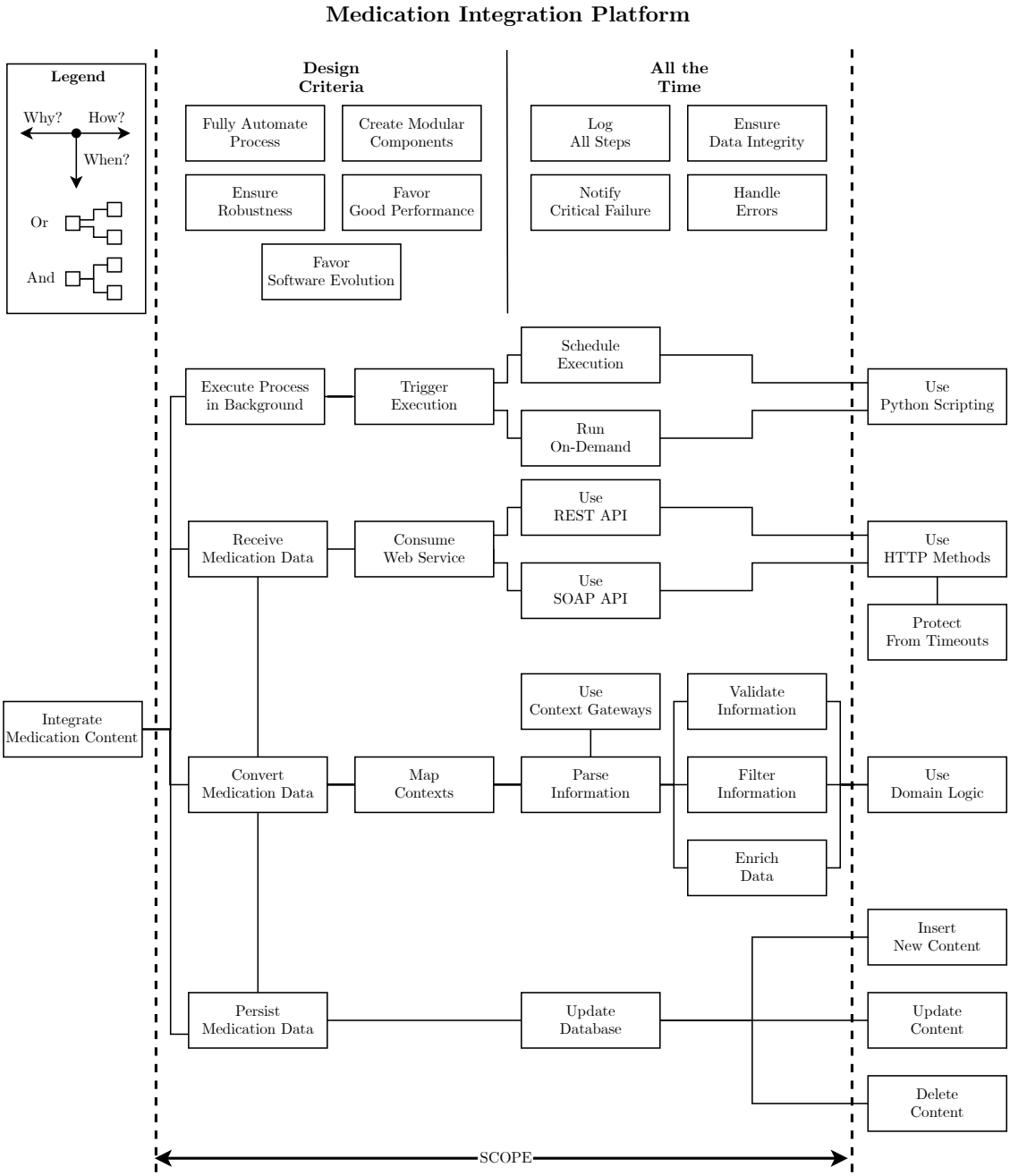

FIGURE 3.8: FAST diagram of MIP.

Chapter 4

ETL Framework

In *Analysis* (Chapter 3), the current project has been thoroughly analyzed and detailed. Section 3.4.1 provided a *System Context* with a *Logical View* that introduced the reader to the envisioned MIP software system. Now, this chapter further explains (Section 4.1) and designs (Sections 4.2 and 4.3) the first major component of this dissertation: ETLF.

It should be duly noted that the Container level of C4 is not present for ETLF because, as a framework used for creating new ETL processes, it has no containers itself. Additionally, some relevant and valid alternatives are provided after the described and accepted solution. Moreover, whenever a design decision affects a requirement, the acronym of that requirement (identified in Section 3.3.2) is laid out right next to it between parentheses. When it is positively affected, a positive sign is associated to it. When negatively affected, a negative sign is displayed.

4.1 What is the ETL Framework?

To define ETLF, a concrete definition of *framework* must be first provided in order to be able to fully comprehend it. With that in mind, the following description by Riehle (2000) resumes pretty well the software framework concept:

> *"Frameworks model a specific domain or an important aspect thereof. They represent the domain as an abstract design, consisting of abstract classes (or interfaces). The abstract design is more than a set of classes, because it defines how instances of the classes are allowed to collaborate with each other at runtime. Effectively, it acts as a skeleton, or a scaffolding, that determines how framework objects relate to each other.*
>
> *[...]*
>
> *An application framework is a framework that ties together a set of existing frameworks to cover most aspects of a certain type of application. [...]*
>
> *In software development based on application frameworks, applications (or systems) become extensions of the application framework. They reuse the software architecture of this particular type of application as defined by the application framework and its domain frameworks."*

All things considered, ETLF can now be defined as a software framework for creating data ETL processes, which can be constructed very differently from one another. This means that a general-purpose, reliable ETL framework cannot be built in a way that restraints developers to use it in a specific manner. For instance, ETL processes can be constructed

so that target databases are either constantly updated or created from scratch. The process might even not directly persist information in a database and, instead, use a Web service to load the desired information on to it. The number of options are numerous. Once this is understood, the framework itself can also be understood. By providing the necessary building blocks, blueprints, and automated operations, ETLF acts as a skeleton and scaffolding for systematically building ETL processes (*+MUEU; +MDRE*). Some classes are to be implemented, others extended, and other ones used out-of-the-box. Through a highly configurable approach, the framework allows developers to focus on the content, rather than on how to address the process itself.

From another perspective, ETLF is like a jigsaw puzzle box with blueprints, instructions, and a set of tools inside it. The blueprints are used to create new puzzle pieces, and the instructions demonstrate how to assemble the puzzle itself. It is then up to the players to manufacture their own pieces and mount them, using the tools provided by ETLF. The end result will be a highly original, personalized puzzle.

Advantages

In its review over framework terminology, Riehle (2000) proceeds in describing the advantages of using application frameworks:

> *"There are a number of key advantages to be gained from using application frameworks. The primary technical advantage is that they provide design and code reuse. The larger and better an application framework, the more design and code reuse becomes possible. Also, systems based on frameworks are easier to maintain, because most key design and implementation decisions are localized in one place, the framework.*
>
> *The primary business advantages of design and code reuse are higher developer productivity and shorter time-to-market of new applications. In addition, applications tend to be less buggy (they are reusing mature implementations), and tend to look more homogeneous (a suite of applications built from the same framework has the same architecture and similar implementations)."*

As one can see, the usage of a framework for the construction of ETL processes contributes considerably to the achievement of the following requirements: *MUEU, PRR* and *PSM*. What should also be emphasized are the advantages of higher developer productivity, shorter time-to-market, and the homogeneity introduced into all ETL processes, which are in line with the *Opportunity Analysis* (Section 3.2.2).

4.2 Components

In this section, ETLF's theoretical components will be detailed for a better engagement with the framework, and, thus, understand it better. To do so, a logical view is provided.

4.2.1 Logical View

Figure 4.1 depicts a theoretical component diagram regarding ETLF and outside components. ETLF assumes that these extrinsic components exist and that they are somehow reachable through any kind of interface, might it be HTTP, ODBC, or even object-oriented APIs. ETLF itself does not directly use such components, only actual implementations do so.

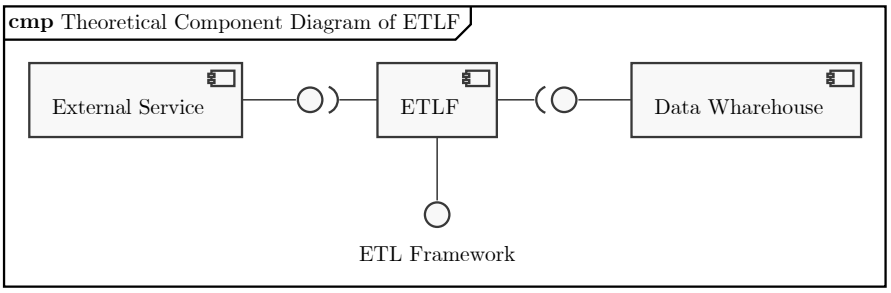

FIGURE 4.1: Theoretical component diagram of ETLF (Level 3 of C4).

4.3 Code

In this section, ETLF's code will be described first without and then with detail. To do so, logical and process views are presented.

ETLF focuses primarily in providing a base architecture for the creation data ETL processes in a systematic and reliable fashion. As it is, performance and memory allocation for processes built using it can be two major drawbacks (*-PPF; -PPT*). However, the framework can be extended in the future to support this kind of requirements.

4.3.1 Logical View Without Detail

To better explain how the framework is organized, a class diagram without fine-grained details is depicted in Figure 4.2. Each one of them has a specific purpose:

- *Builder* — responsible for the execution of some assembling operations required by the process, which includes creating all required objects, injecting them with dependencies, and loading configurations;
- *Controller* — responsible for orchestrating the workflow of the ETL process;
- *Mapper* — responsible for the transformation of external data into business *Entities*;
- *Entity* — contains business data of a specific type, while incorporating insertion and validation logic;
- *Connector* — responsible for connecting to external services;
- *Parser* — responsible for parsing external data into a Python object;
- *Loader* — responsible for loading information into its final destination.

The *Connector*, *Mapper*, and *Loader* arose directly from the intrinsic sub-procedures of ETL (*FRETL*). The *Builder* and *Controller* appeared from the need to construct and control complex processes —a side effect of *Create Modular Components* and *Favor Software Evolution* from FA. The *Parser* emerged from the need to analyze and/or transform external data formats.

As already explained, ETLF provides blueprints for the creation of ETL components. These blueprints are represented as either interface or abstract classes. The main difference between them is that while interface classes only stipulate a contract that needs to be fulfilled by implementations, base classes support this functionality while allowing one to define default behaviors to all subclasses. As an example lets look at the *Connector* and

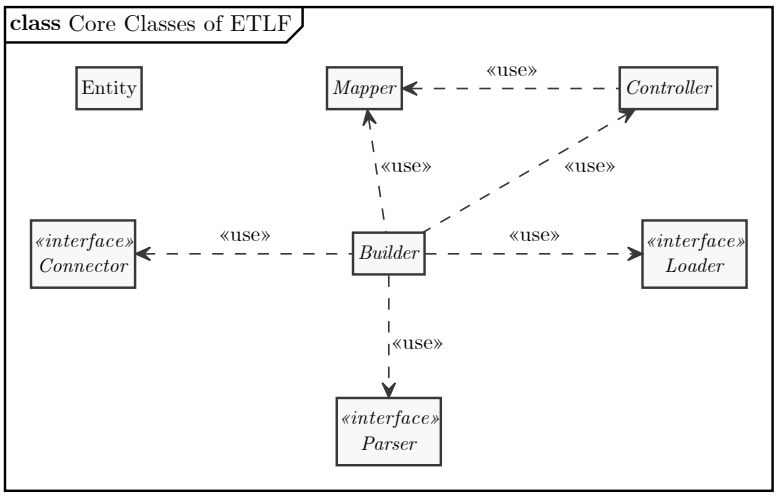

FIGURE 4.2: Class diagram of ETLF without fine-grained details (Level 3 of C4).

Controller components. The *Connector* component defines a predefined set of methods that need to be implemented in order for it to respect the rules of the framework itself. All implementations of the *Connector* interface must oblige to this contract. On the other hand, the *Controller*, in addition to defining a predefined set of abstract methods, it also has its own functionality of triggering the transformation of *root Mappers*[1]. With that in mind, the *Builder*, *Controller* and *Mapper* represent abstract classes, and the *Connector*, *Parser* and *Loader* components represent interface classes.

As one can see, the *Builder* uses almost all classes so that these can be appropriately assembled and injected as dependencies in a systematic fashion (*+MUEU*). The *Controller* makes use of *Mapper* classes in order to systematically trigger the transformation of information (*+MUEU; +MDRE; +MDRM*). *Mappers* are responsible for, firstly, setting off the transformation of their own required *Mapper* dependencies, and then triggering its own transformation (*+MUEU; +MDRE; +MDRM; +PRI*). As it is possible to see, in ETLF alone, the *Mapper*, *Connector*, *Parser*, and *Loader* classes do not make use of one another. This is because only implementations of these classes will contain the required logic to do so. However, depending on the implementation, the *Connector* and *Loader* classes should be called from either a *Mapper* or a *Controller*, while the *Parser* should be called from a *Mapper* or a *Connector*.

Apart from blueprints, ETLF also provides some operations reusable across business *Entities*. *Entities* are to ETLF what tables are to SQL, though they have a closer proximity to ETL and data integrity. In other words, *Entities* contain a collection of columns that represent data, while providing functionalities to insert and evaluate its own items systematically (*+MUEU; +MDRE; +MDRM; +PRI*).

4.3.2 Process View Without Detail

Figure 4.3 shows the very basic workflow for ETL processes built using ETLF. Processes start by first building all major components intrinsic to ETLF (i.e., *Controller*, *Mappers*, *Connectors*, *Parsers*, *Loaders*). Then, the *Controller* must be called to trigger the ETL

[1] *Root Mappers* are *Mappers* that are not dependencies of any other *Mapper*.

process. It uses a list of *root Mappers* to start their transformation in a sequential and systematic way (*-PPF; +PPT*). Thereupon, each *Mapper* triggers the transformation of *non-root Mappers* that it depends on. Then, when all dependencies have been satisfied, the *Mapper* triggers its own transformation (*+PRI*). The process is completed when all *root Mappers* have finished their job.

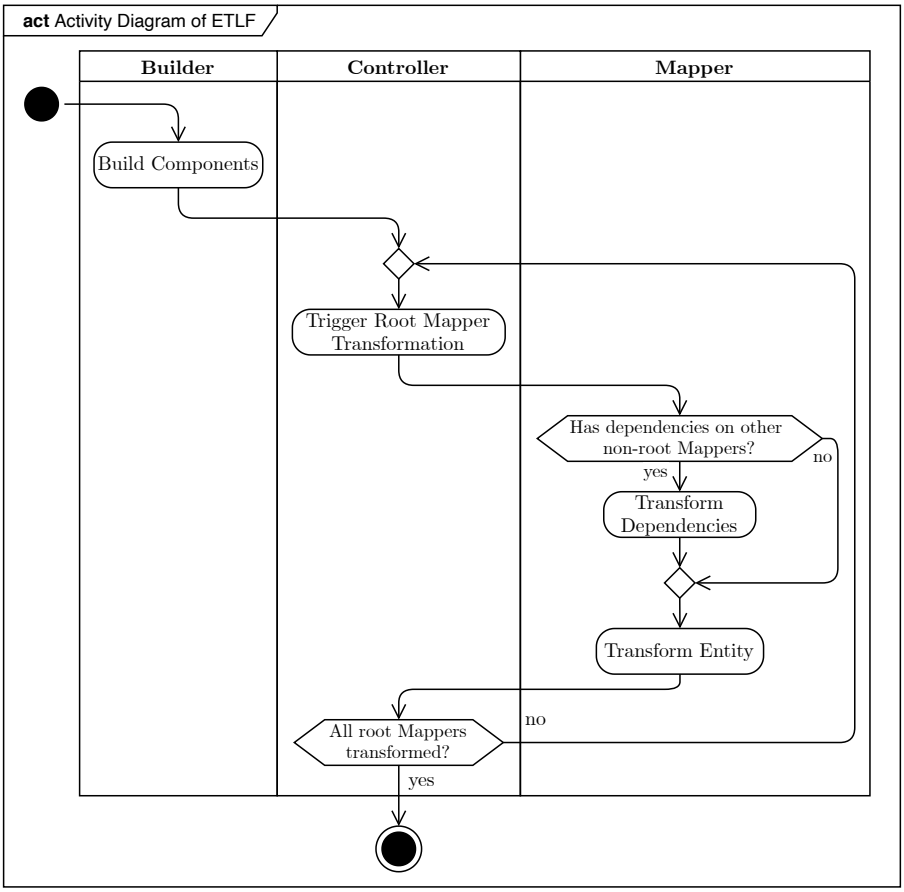

FIGURE 4.3: Activity diagram of ETLF (Level 3 of C4).

4.3.3 Logical View With Detail

Delving further into the framework's *Logical View*, Figure 4.4 depicts a diagram of classes, highlighting their details and the dependencies between them. The evidenced modularity allows for a greater customization of ETL processes, better and easier testing of its components (*+PRI; +PRR*), and better software evolution and maintenance (*+MUEU; +MSC; +MDRE; +MDRM*). The framework was designed having the *Process Manager* architecture in mind. This architecture supports complex and easily changeable message flows, and provides a central point for administration and reporting (*+MFO; +PFA; +PFN; +PRSur; +PSM*). It can, however, lead to bottlenecks and substantially decrease the performance of the process at hand (*-PPF; -PPT*). If this is to be avoided, *Mappers* could make use of pipelines and thereby implement a *Pipes and Filters* architecture for data routing/processing (*+PPF; +PPT*).

The `Builder` is an abstract class, responsible for building all the necessary components. The `build()` operation calls all other abstract `building` methods for automation purposes (*+MUEU; +MDRE; +MDRM*). These `building` functions need to be properly

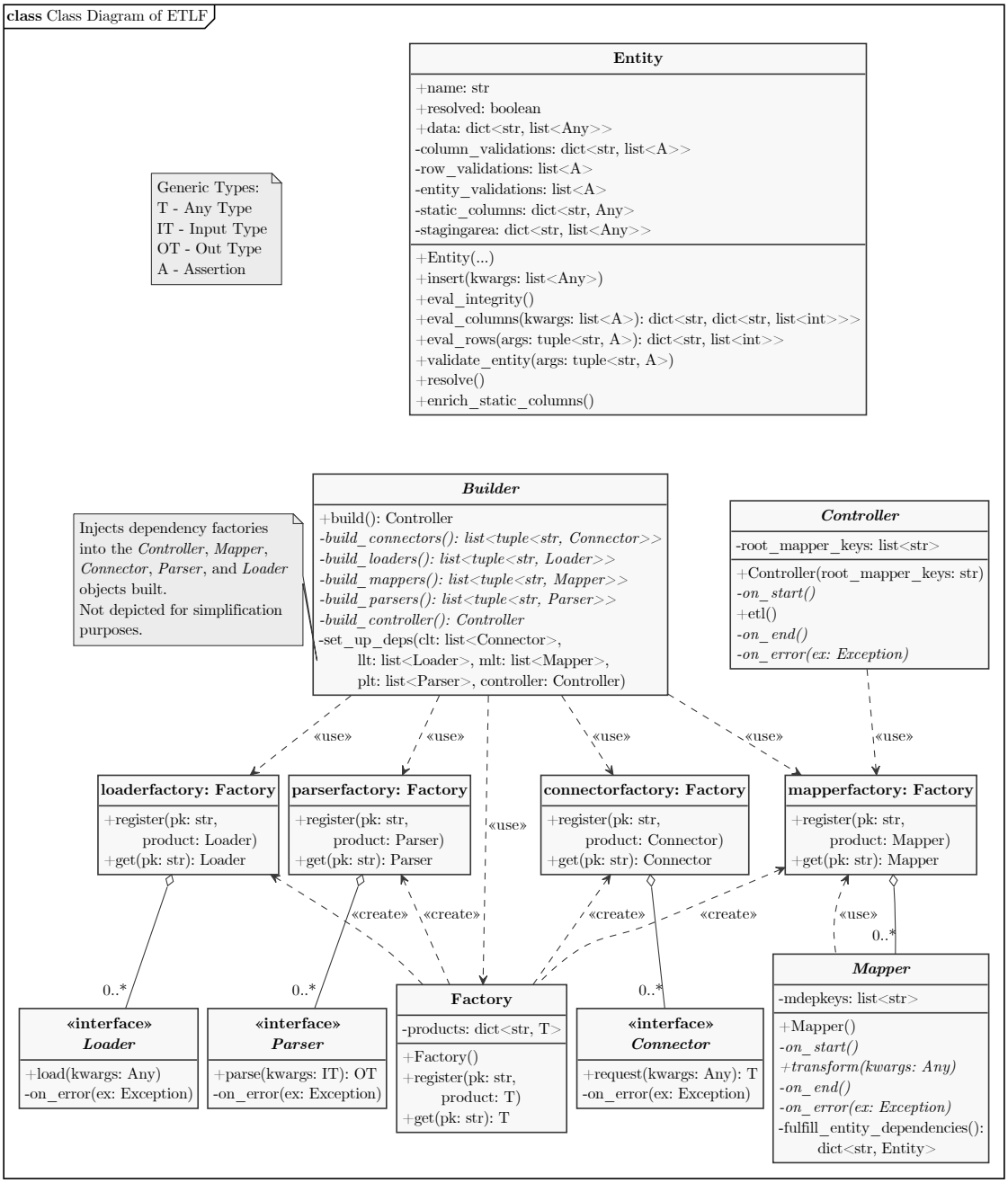

FIGURE 4.4: Class diagram of ETLF (Level 4 of C4).

implemented according to the specification of each project. The last step is to inject dependencies in the form of *Factories* into all relevant components (i.e., *Controller*, *Connectors*, *Parsers*, *Loaders*, *Mappers*) (*+MUEU*). This way, dependencies can be used from within the component by default, not requiring objects to be passed on through constructors.

A `Factory` is an implementation of the *Factory Method* which is used to "[d]efine an interface for creating an object, but let subclasses decide which class to instantiate" (Gamma et al. 1995). This will become quite handy for ETLF considering that the

framework itself cannot predict which concrete classes it will need to create or call — remember that most of what ETLF provides are blueprints for ETL components and not the ETL components themselves. Considering all the above, ETLF will contain *Factories* for the following components: *Mappers*, *Connectors*, *Parsers*, and *Loaders*. In short, the main points that explain why it was decided to use this design pattern include:

- The framework can provide some standard and commonly used products[2], while also enabling to easily add new customized ones, according to the requirements of each project (*+MUEU; +MDRE*);
- Classes can easily change dependency classes or extend to new behaviors that require other classes (*+MUEU; +MDRE: +MDRM*).

The `Controller` is another abstract class that, as the name suggests, controls the entire workflow of the process. ETL processes are initiated by calling the `etl()` method, which uses the `root_mappers` property to trigger the transformation of the main `Mappers` (*+MUEU; +MDRE; +MDRM*). The `on_start()`, `on_end()`, and `on_error()` are abstract methods that need to be properly implemented in subclasses according to the requirements of each ETL process (*+MUEU; +MDRE*).

`Entities` are directly instantiable model classes that represent any kind of business information type. They are by far the most relevant ETLF class. Here, all operations are already implemented and can be used as-is. An `Entity` contains the following properties and methods:

- `name` — a name for the `Entity`;
- `resolved` — a boolean to know if the `Entity` is already resolved or not[3];
- `staging_area` — a native dictionary of lists (selected using AHP in Section A.1) that collects data while the `Entity` is still not resolved;
- `data` — the `Entity`'s main property;
- `column_validations` — a list of assertions to be performed to all cells by columns;
- `row_validations` — a list of assertions to be performed to all the rows present in the `staging_area`;
- `entity_validations` — a list of assertions that need to be fulfilled in order for an `Entity` to be valid;
- `static_columns` — a business specific dictionary used to populate new columns with static data;
- `insert(...)` — insert data into the `Entity`'s `staging_area`;
- `eval_integrity()` — evaluate the integrity of the `staging_area` (*+PRI*);
- `eval_columns(...)` — evaluate cells of the `staging_area` by column (uses the `column_validations` property) (*+PRI*);
- `eval_rows(...)` — evaluate rows of the `staging_area` (uses `row_validations`) (*+PRI*);

[2] A product is an object the factory creates.

[3] *Resolved Entities* are *Entities* whose data structures are final. *Non-resolved Entities* are *Entities* that are not yet final, and, therefore, their data structure is subject to change.

- `validate_entity(...)` — validate the entire `Entity` (uses `entity_validations`) (*+PRI*);
- `resolve()` — resolve the `staging_area` of the `Entity` into the `data` property;
- `enrich_static_columns()` — *Content Enricher* to populate the `staging_area` with static columns (uses the `static_columns` property).

`Mappers` are abstract classes that control how an `Entity` is to be integrated. In most cases, for each `Entity` there should be a corresponding `Mapper`. `Mappers` are very specific classes that contain very specific logic, which cannot be overstepped by the framework itself. Because of this, the `on_start()`, `transform()`, `on_end()`, and `on_error()` operations must be properly implemented in all subclasses (*+MUEU; +MDRE*). With that in mind, `transform()` can be implemented with either *Pipes and Filters*, *Routing Slip*, or *Process Manager*, depending on (a) how the flow of data is to be managed, (b) where business logic is to be located, and (c) the complexity level desired for each processing component. Here is where, directly or indirectly, most of the *Routing* (Section 2.3.2) and *Transformation* EIPs (Section 2.3.3) might be useful if the `Mapper` contains the translation logic. `on_start()` and `on_end()` were designed for tasks not directly related to data operations, but rather for activities such as logging, *wire tapping*, etc. (*+MUEU; +PFA; +PSM*). The framework uses dependency trees to integrate the required content of a `Mapper`. More concretely, each `Mapper` contains an `mdepkeys` property used to (a) know what are the other `Mappers` that the current one depends on (*+MUEU; +PRI*), and (b) to trigger their transformation through the `fulfill_entity_dependencies()` method (*+MUEU; +MDRE; +MDRM*). Theoretically speaking, there can be two types of *Mappers*: *EntityMappers* and *EntryMappers*. *EntityMappers* are used to extract and transform entire `Entities`, while *EntryMappers* are used to extract and transform single items of an `Entity`. The main difference between the two is that the `transform()` method of *EntityMapper* implementations should *not* require parameters, while *EntryMappers should* require them in order to identify which specific item to transform.

`Connectors` are interface classes that follow the *Service Connector* pattern and, therefore, make connections to all external systems where no other component should be able to do so. They are interface classes that need to properly implement the `request()` and `on_error()` methods (*+MUEU*). Multiple `Connectors` can be chained together in order to reuse common behaviors, such as authentication and encryption, thus acting like *Envelop Wrappers*. Besides that, `Connectors` can also make use of caching mechanisms to limit the number of HTTP requests.

`Parsers` are interface classes used to process input responses and generate object-oriented outputs. With that in mind, they can be used for fundamentally two purposes: convert communication data formats into Python objects, and convert data into business objects. The first one will always need to be performed when using Web services. The latter one will depend on the integration approach, but, if it's used, it should also implement a *Content Filter* to remove irrelevant and unnecessary information, therefore implementing a *Tolerant Reader* pattern to fetch only the required data (*+PRR*). Moreover and if desired, a `Connector` can use directly a `Parser` of the second identified kind, being transformed into a *Service Gateway* component that returns a business object and, thus, acting like a *Translator*.

A `Loader` is an interface class that's responsible for persisting information in any desirable way. This means that it does not have to use directly a database. It can, for instance, use

another Web service through another `Connector` to persist data in another distributed system. With that in mind, `Loader` classes must implement the `load()` and `on_error()` methods on their own way (*+MUEU*).

Alternatives:

- `Factories` could be removed and their products directly called by the implemented classes, but this would turn the code into something more static, not protecting it from software evolution and maintenance (*-MUEU; -MDRE; -MDRM*).
- The `on_start()`, `on_end()`, and `on_error()` methods could be removed from the classes' contract and provide a more flexible approach. However, by not compelling subclasses to implement `on_error()`, ETLF would most often fail in the provision of a reliable, systematic ETL processes to developers. Nonetheless, such methods can still be skipped by simply using the `pass` Python statement in a conscious and explicit manner.

4.3.4 Process View With Detail

Figure 4.5 depicts how the execution of processes created using ETLF should be done. The illustrated `Executer` class is not specific to ETLF, it is just a general component that's responsible for triggering the execution of the process. Firstly, a `Builder` is instantiated and then it's `build` operation called. After that, when all components are properly set up, the ETL process is triggered through the `Controller`.

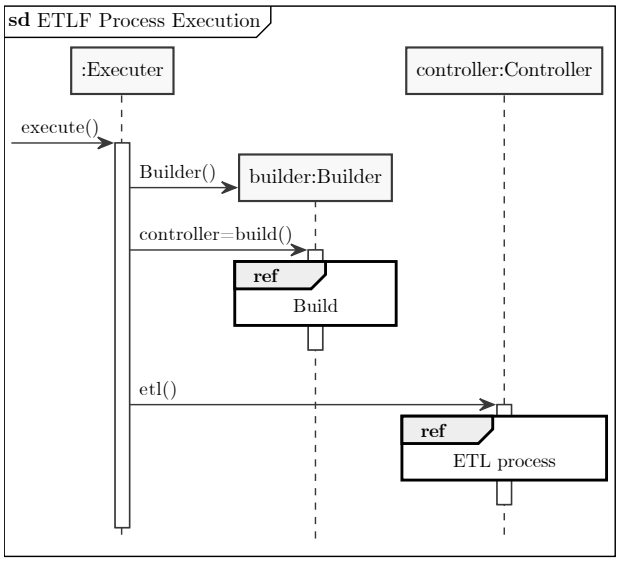

FIGURE 4.5: Sequence diagram of executing processes built using ETLF (Level 4 of C4).

The `build()` method is responsible for triggering all other building operations (Figure 4.6). After building all components, dependencies need to be properly set up and injected into the components that require them. To do so, `Factory` objects are created and their respective products registered. After that, all objects just built are injected with classes that they depend on (*+MUEU*). Please refer to Section 6.1.1 for more information regarding ETLF's dependency injection through the *Builder* component.

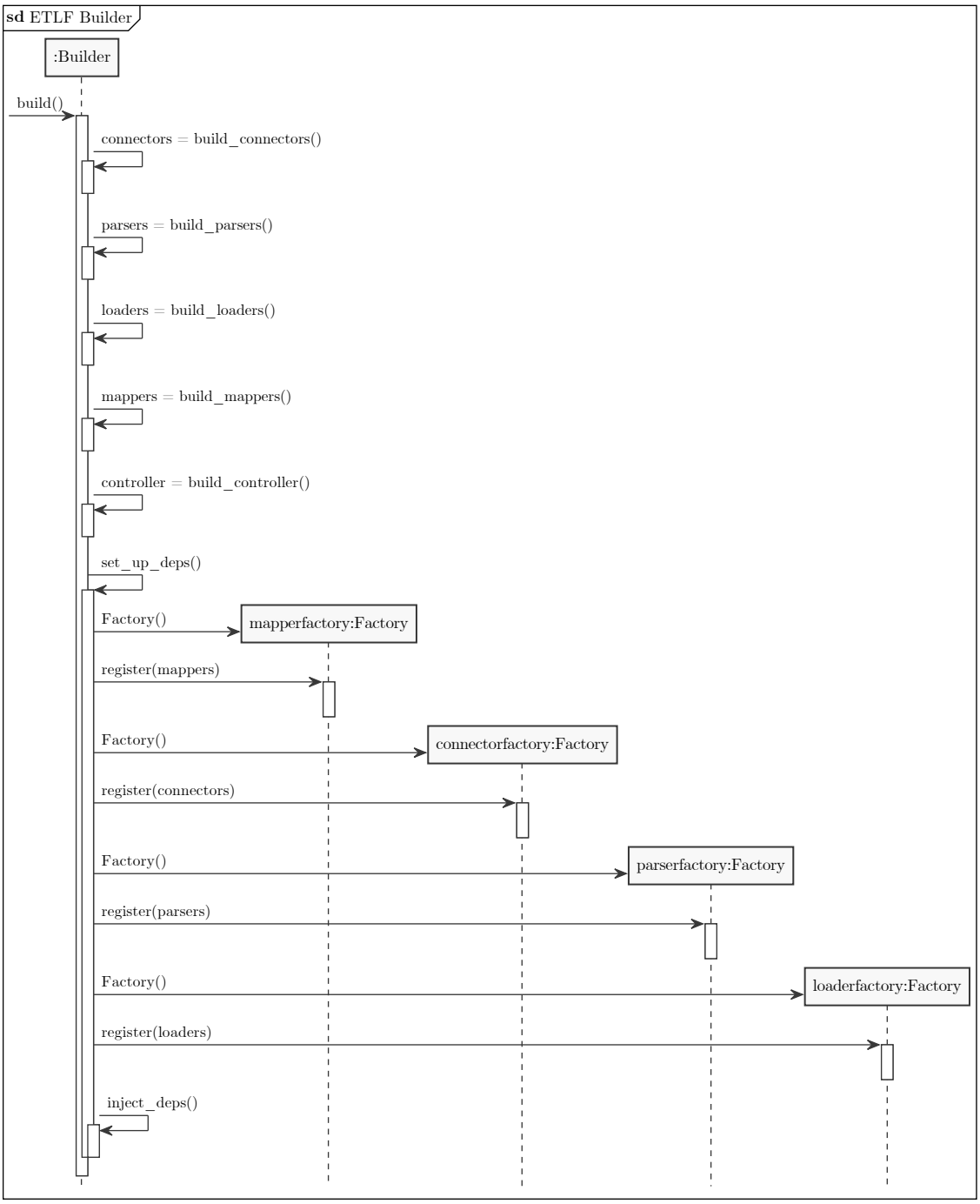

FIGURE 4.6: Sequence diagram of the ETLF Builder (Level 4 of C4).

The actual ETL process only starts with the `Controller`'s `etl()` method (Figure 4.7). The transformation of all identified `root Mappers` is carried out after the execution of the `on_start()` operation. Here is where all `Entities` of a specific business should be extracted, transformed, and persisted. After the transformation of all `root Mappers`, the `on_end()` method is called to terminate the process.

All `Mappers` should have the same workflow that Figure 4.8 presents. They start off by triggering the transformation of dependencies through the `fulfill_entity_dependencies()`

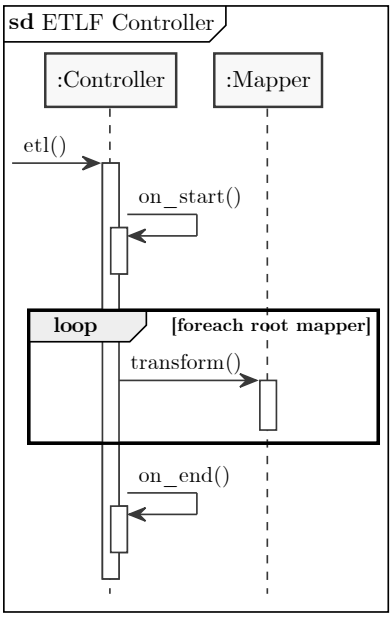

FIGURE 4.7: Sequence diagram of the ETLF Controller (Level 4 of C4).

operation of ETLF. Only when all its dependencies are satisfied does the root `Mapper` continues to operate by executing the `transform()` method. Here, data needs to be (a) fetched from the service through a `Connector`, (b) parsed through a `Parser`, and (c) transformed. When all of this is accomplished, rows can then be inserted and validated through the `Entity` and, thereafter, persisted through a `Loader`. This is not depicted in the sequence diagram because the framework itself does not contain the logic to do so. This logic depends on concrete process implementations.

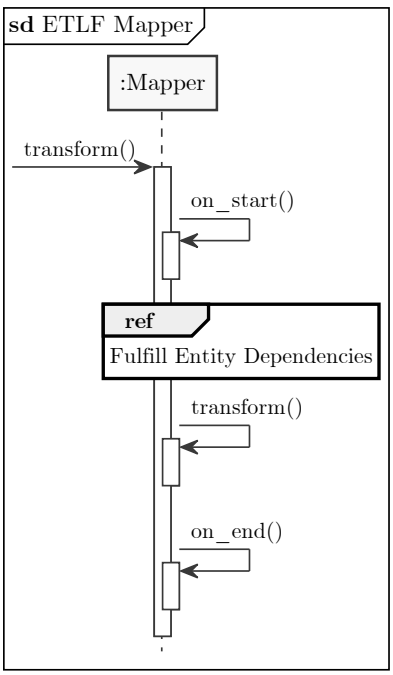

FIGURE 4.8: Sequence diagram of the ETLF Mapper (Level 4 of C4).

To fulfill `Mapper` dependencies, the `mdepkeys` property is iterated, triggering the transformation of the key's respective `Mapper` (Figure 4.9). From this point forward, a normal workflow of a `Mapper` is carried out. Please refer to Section 6.1.3 for more information about the dependency fulfillment of *Mappers*.

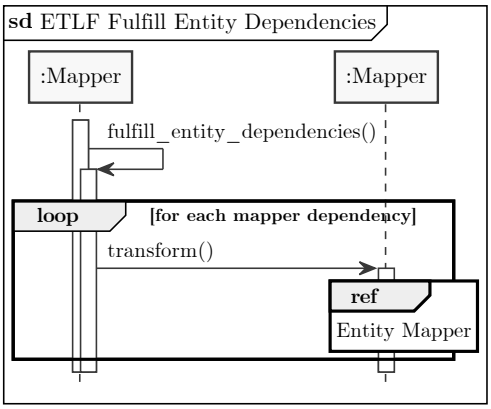

FIGURE 4.9: Sequence diagram of fulfilling entity dependencies with ETLF (Level 4 of C4).

Chapter 5

Medication Integration Platform

In *Analysis* (Chapter 3), the current project is thoroughly analyzed and detailed. Section 3.4.1 provided a *System Context* and a *Logical View* that introduced the reader to the envisioned MIP software system. After that, in *ETL Framework* (Chapter 4), the first major component of this dissertation has been explained and designed. Now, this chapter further explains (Section 5.1) and designs (Sections 5.2, 5.3, and 5.4) the main objective of this dissertation: MIP.

Some relevant and valid alternatives are provided after the described and accepted solution. Moreover, whenever a design decision affects a requirement, the acronym of that requirement (identified in Section 3.3.2) is laid out right next to it between parentheses. When it is positively affected, a positive sign is associated to it. When negatively affected, a negative sign is displayed.

5.1 What is the Medication Integration Platform?

In this dissertation, MIP has been described as a "software platform used to build the actual integration processes responsible for integrating medication data into the ALERT® final product" or as a "healthcare IT solution for reliable medication integration". This means that MIP is more than integration processes: it is with MIP that reliable medication integration processes are built. Besides containing the actual processes, MIP will also incorporate libraries for executing medication operations specific to ALERT®. This includes data models, medication-specific validations, staging area operations, helper functions, configurations, and many others. Added to that, it will also enclose libraries aimed at connecting to relevant external services and appropriately parsing their supplied information. In other words, MIP will allow developers to systematically create medication integration processes without requiring them to be built from scratch over and over again.

Picking up the jigsaw puzzle analogy from Section 4.1, MIP uses the blueprints provided by ETLF to assemble reusable medication pieces. These pieces can then be used to mount multiple jigsaw puzzles. The end result will be several distinct medication-oriented puzzles, where some parts can be identified as being transversal to all of them, even though they are all different from each other. This end result is the ultimate objective for which ETLF and MIP were created: to create numerous medication ETL processes in a systematic and reliable fashion.

5.2 Containers

This section represents level 2 of the C4 view model. Here, MIP's containers will be detailed using a logical view of the system.

5.2.1 Logical View

Figure 5.1 depicts relevant containers regarding MIP itself and the ALERT® destination system. As one can see, integration is a completely independent process, triggered and operated in its own way, and only coupled to ALERT® through the primary database. As long as the extracted data continues to be loaded in a valid format, ALERT® requires no modifications whatsoever. The *Medication Service* is consumed via HTTP by the *Medication Integration* container in order to extract all required information. This data is then transformed and finally persisted in the *Primary Database* of ALERT®. The *Staging Area* is used to hold the extracted and transformed information in an intermediary database, which is completely disconnected from the primary one. It uses a *Canonical Data Model* specific to ALERT®, where all data is terminologically and ontologically compatible with all versions. Thus, it acts as an ACL to the overall integration system. Additionally, the *Command-Line Interface* provided by MIP is delegated to the *Data Integration* container, which is the container responsible for managing the entire workflow of the integration process from start to finish.

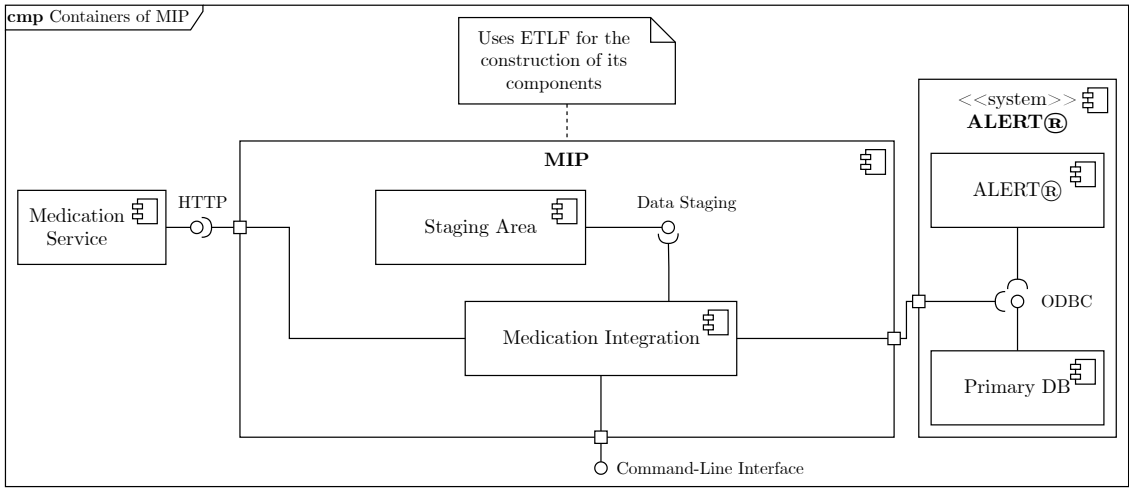

FIGURE 5.1: Component diagram of MIP depicting its containers (Level 2 of C4).

This specific approach provides the following perks:

- Data integration is an independent, self-governing process;
- Runtime failures are contained (the main system is not affected in any way by medication integration processes).

Although it can be contemplated as a good solution considering the requirements, the fact that the primary database is used as the final destination for medication content bears a few drawbacks:

- Direct coupling to the primary database;
- The integrity of the primary database can be compromised by integration processes;

- Decreased maintainability.

Alternative:

In my view as the author of the current dissertation, a more suitable approach to store medication content in the ALERT® system would be to use two completely separate databases —one for medication and another for all ALERT®-related data— and make ALERT® communicate with both (Figure 5.2). This would mean that medication integration processes do not require, in any way, to manipulate the primary database, which contains critical data for the correct functioning of ALERT®. With this approach, it's possible to take advantage of SQLite's capabilities, which is a self-contained, easily deployable, and serverless database (SQLite n.d.). This would allow for a completely separate handling of medication data, while not requiring a fully fledged database management system.

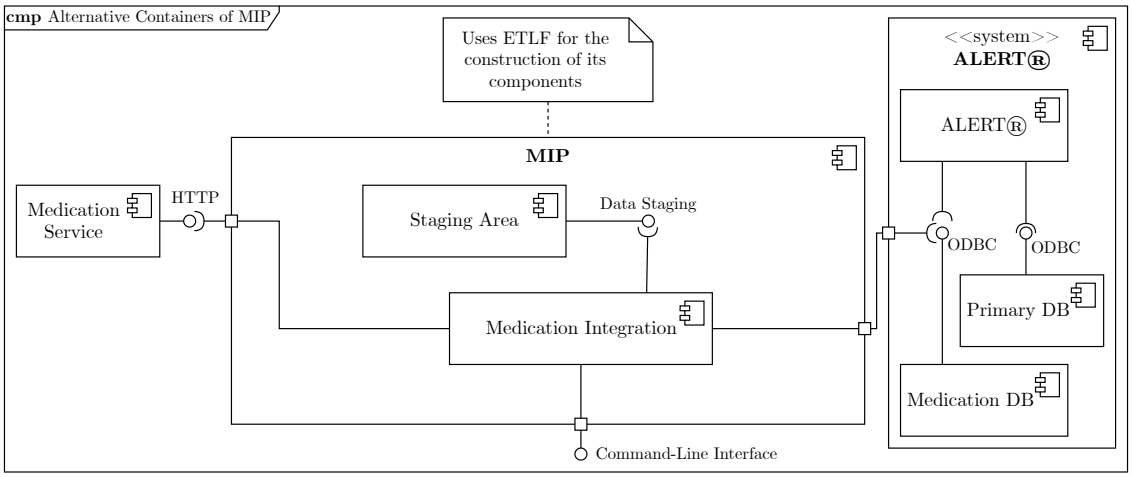

FIGURE 5.2: Component diagram of MIP where the data destination is a completely separate medication database (Level 2 of C4).

This approach would allow for all the perks described for the first solution and fix its identified disadvantages:

- Data integration is very loosely coupled to the main system;
- Naturally enforced domain boundaries between the medication BC and all other ALERT® content types, which means:
  - The primary database remains completely protected from any external data that is not directly managed by ALERT;
  - There is a clear separation of concerns, which would only further increase software sustainability;
- Medication databases could be easily deployed to multiple clients that operate within the same market rules, without worry of security breaches to the primary database;
- Regardless of the location of the primary database, integration processes can be easily executed on customers' machines and its resulting content easily stored.

However, this approach would introduce the following disadvantages:

- Decreased database maintainability by the Content Team;
- Higher complexity level for ALERT® as there are more databases that need to be connected to.

Be that as it may, this approach must be disregarded because (a) ALERT prefers its current approach, and (b) it would require a considerable amount of expense to modify the existing ALERT® product and its primary database.

5.3 Components

Now that the containers of the system have been detailed, this section will expand upon the components of those said containers (level 3 of C4). To do so, process, logical, and deployment views are provided in the next subsections.

5.3.1 Process View

As Figure 5.3 depicts, *Medication Integration* is decomposed in two subprocesses.

- *Medication ETL* is where external data is extracted, transformed, and loaded into the intermediary staging area;
- *Medication Importation* is where data from the staging area is imported into ALERT® according to its version.

Medication ETL acts like a *Normalizer* responsible for creating and populating the staging area. Here, two main stages have been identified: *Build*, and *ETL*. Regarding the process build, it can either run on schedule or be manually triggered, which are both easily achievable in Python due to its scripting nature (*+MFO*). Different ETL processes have different requirements, which might also differ according to how the service provider operates their data. Ergo, each process requires an appropriate configuration set up according to its specific restrictions (*+MSC*). Next comes the ETL process itself. While data extraction and transformation are to be executed in partial blocks, it is only in the end that the persistence step is carried out. This allows for data to be extracted (*FRE*) and transformed (*FRT*) while in memory, and, in the end, loaded into the staging area (*FRL*). This ensures that (a) all data is not extracted at once (*+PPT*), (b) better performance is achieved (*+PPF*), and (c) no invalid data is loaded into the staging area ever (*+PRI*). Furthermore and as expected, resources do not come with all the required information to be integrated according to the domain rules of ALERT®. Because of this, many of them require additional calls to the external service to fetch all the required information (*+PRI*).

When the ETL process ends, the final phase of the integration process is triggered. *Medication Importation* must then populate the primary database with the information contained in the staging area. Conceptually, *Medication Importation* is an entirely distinct process that, firstly, focuses on transforming the data from a canonical format into its final version, and then loading this data into the primary database. In practice, however, it can be considered another ETL process, where, instead of extracting data from a Web service, this data is extracted from a database, transformed according to the ALERT® version, and loaded into its final destination. To emphasize again, only Medication ETL will be dealt with in the current dissertation. However, MIP must allow for the Medication Importation to be executed in a simplistic way.

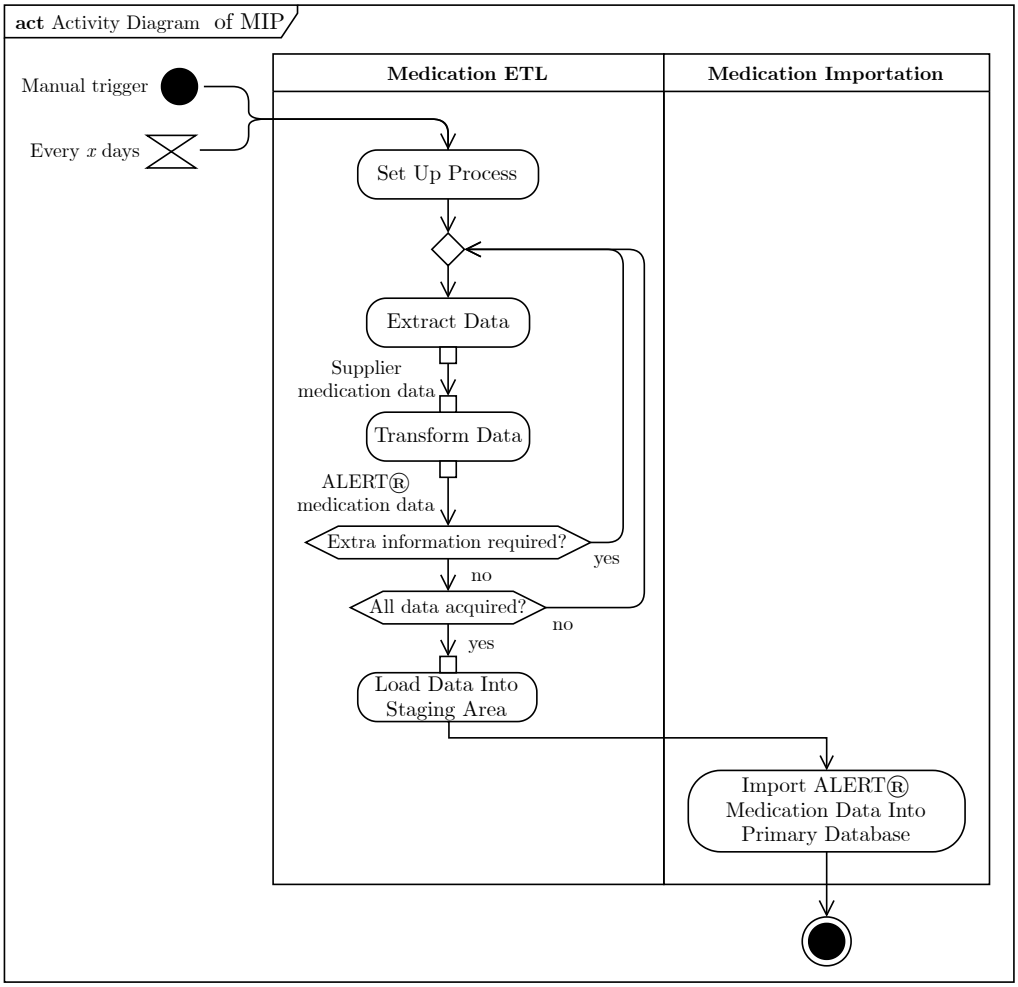

FIGURE 5.3: Activity Diagram of MIP (Level 3 of C4).

This specific approach provides the following perks:

- Compartmentalization of the data extraction and importation mechanisms (*+MDRE; +MDRM*);
- The scope of failures is partitioned by the task at hand;
- Data extraction is protected from the numerous ALERT® versions (*+MDRM*);
- Data extraction is protected from ALERT® requirement changes (*+MDRM*).

Alternative:

A very subtle alternative would be to, instead of first fetching and transforming all data in-memory for all *Entities* and only then persist it, the ETL process could extract, transform, and persist data for a single *Entity* sequentially (Figure 5.4). This means that all data of a single *Entity* would be extracted and transformed, and then immediately persisted to avoid having large amounts of information in memory (*+PPT*).

This alternative would, however, bear two downsides:

- Increased integration time due to Input/Output (I/O) operations to the staging database during validations (*-PPF*);

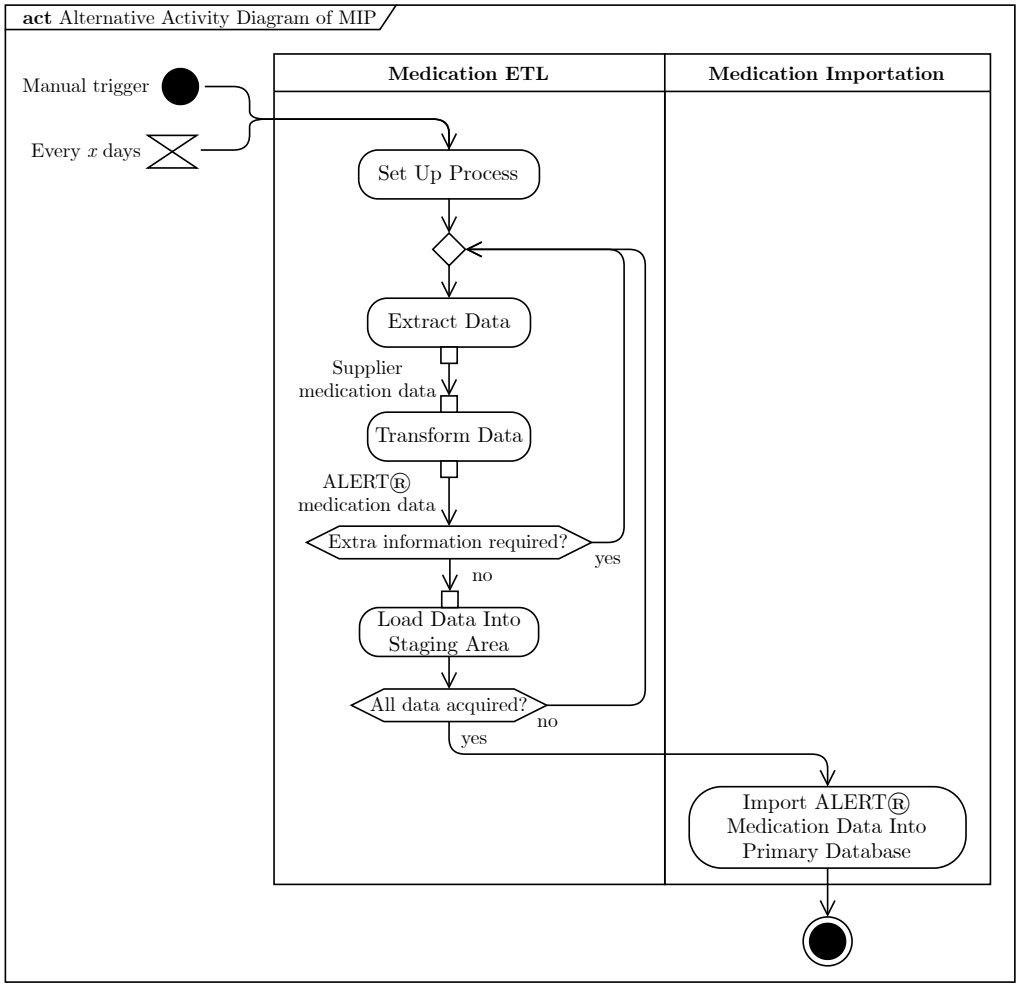

FIGURE 5.4: Alternative activity diagram of integration processes in MIP where data is persisted sequentially (Level 3 of C4).

- The staging area would temporarily contain invalid and incomplete data (*-PRI*).

5.3.2 Logical View

Delving further into the logical view, Figure 5.5 depicts the components of MIP's containers. Regarding *Medication Integration*, one can see that it combines four components whose purposes are very specific: *Orchestrator*, *Medication ETL*, *Medication Importation*, and *Service Gateway*. The *Orchestrator* can be a quite simple component responsible for managing the general workflow of the process. It first uses the *Medication ETL* component to carry out the Medication ETL process (*FRETL*), and, afterwards, triggers the subsequent Medication Importation process through the *Medication Importation* component (*FRI*). The *Service Gateway* is only used by *Medication ETL* in order to properly extract data from the desired external Web service (*FRE*). On the other hand, the *Staging Area* can be decomposed into two components: *In-Memory* and *Persistent SA*. The *In-Memory SA* is used to hold the data while it is being extracted and transformed. After that, it loads its own data into the *Persistent SA* (*FRL*) through a PEP-249 API[1]. In

[1]PEP-249 is a Python-specific module API that defines rules to access databases (Lemburg 2001).

the end, *Medication Importation* uses this *Persistent SA* to import data from an internal canonical data format into the desired ALERT® version (*FRI*).

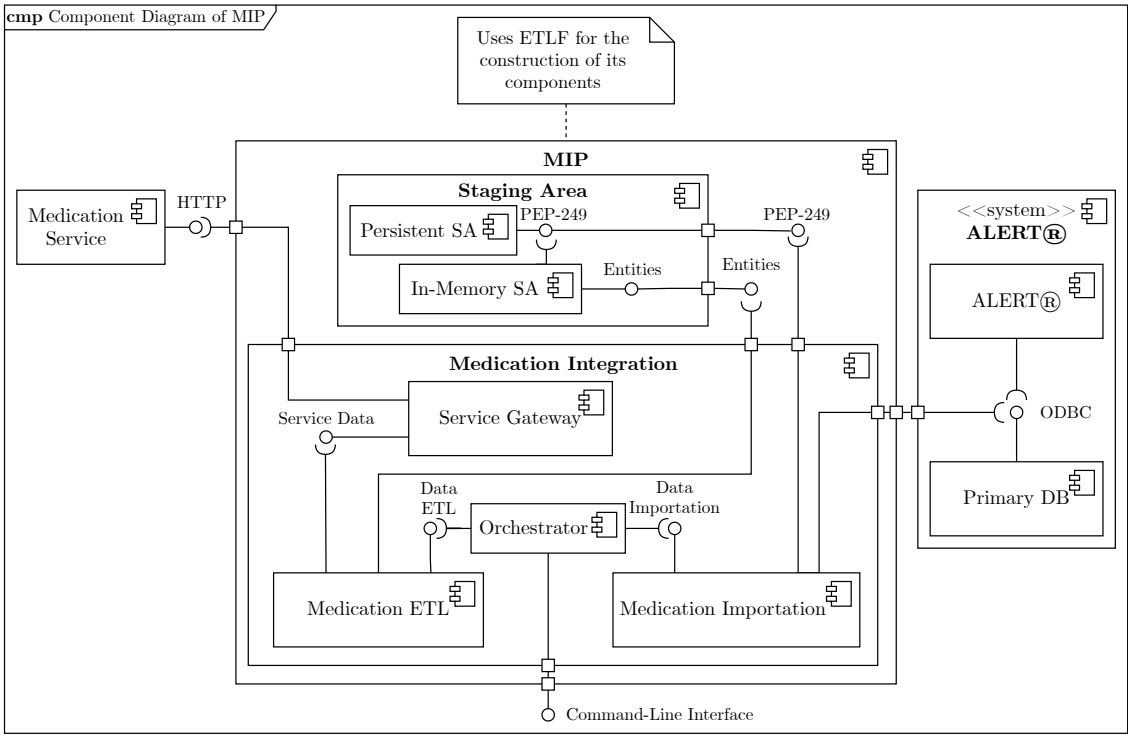

FIGURE 5.5: Component diagram of MIP (Level 3 of C4).

Regarding the *Orchestrator* component, its use bears the following advantages and disadvantages:

- *Medication ETL* and *Importation* processes are completely oblivious to each other, which means they have no knowledge regarding the other one (*+MUEU; +MDRE; +MDRM*);
- Reduced component-level complexity (*+MUEU*);
- Increased container-level complexity (*-MUEU*), as there are more components that need to be properly established;

The main perk for using the *In-Memory SA* is to be able only load data into the *Persistent SA* in its final and valid state, avoiding data inconsistencies and incompleteness (*+PRI; +PRSur*). Additionally, this approach allows one to first extract, and then filter, enrich and validate all data without requiring to many I/O operations. This means that the process will turn out to be slightly faster (*+PPF*), but requiring more memory to be able to handle all data (*-PPT*).

Alternatives:

An alternative to using an *Orchestrator* component would be to set up the integration process as a *Choreography*, introducing workflow logic into the other components, as depicted in Figure 5.6. This option would, however, revert the component-level advantages of the *Orchestrator* (*-MUEU; -MDRE; -MDRM*).

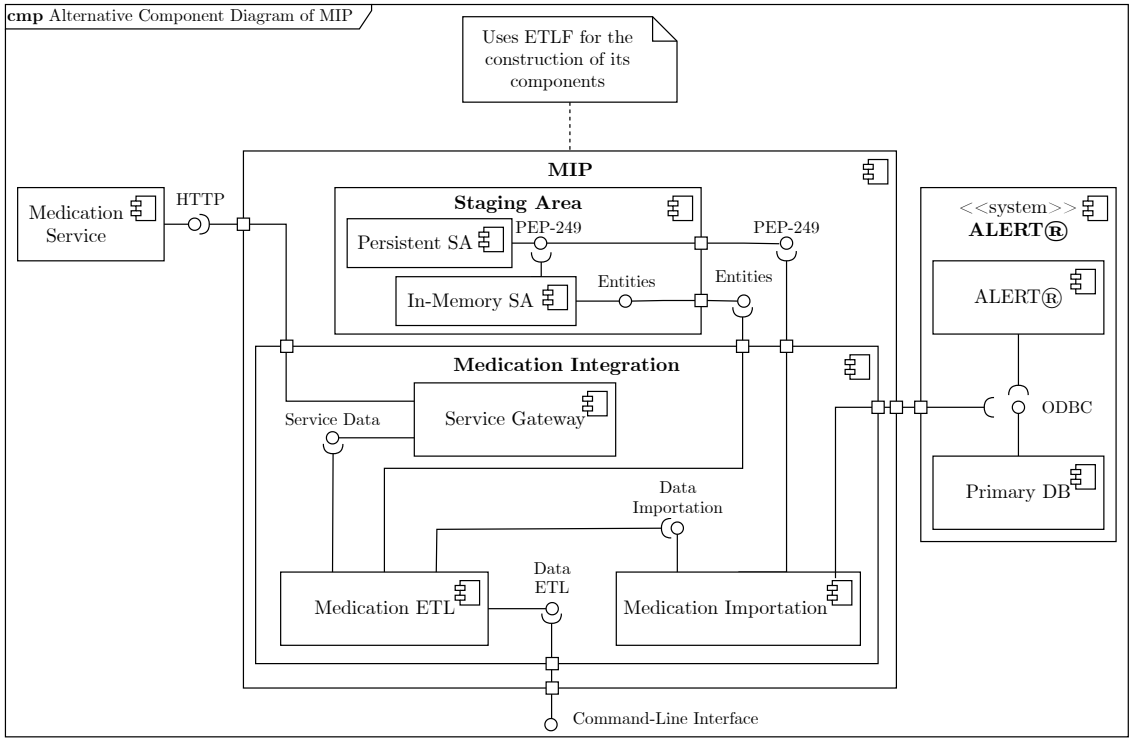

FIGURE 5.6: Component diagram of MIP without a process orchestrator (Level 3 of C4).

Another alternative regarding the *Staging Area* has also been identified. Here, the *In-Memory SA* component could be removed and only have the persistent staging database (Figure 5.7). This approach would allow for less memory to be used at certain times (*+PPT*). However, by saving all data instantaneously in a persistent database, the number of I/O operation would dramatically increase when performing validations that require previously transformed data (*-PPF*). Another negative aspect is that this approach would mean that data would be loaded into the staging area without being entirely validated, not preventively protecting the system from incorporating invalid data. This poses a major threat to the *PRI* requirement.

5.3.3 Physical View

The idea is to deploy integration processes directly to the client servers and set it up to execute periodically, relieving ALERT from extra server maintenance costs. Figure 5.8 depicts this outlook for a physical allocation of resources.

Alternatives:

In cases that justify it (e.g., incompatible server, isolated environment required), the integration process could be containerized via Docker (Figure 5.9). This would boost the adaptability of the solution to all relevant environments (*+MSA*), while maintaining a local process execution in the clients' servers.

Another alternative in cases that justify it (e.g., incompatible client server, service cannot be overloaded with requests), would be for the ETL phase to be executed in ALERT servers and then its artifacts deployed to the client server (Figure 5.10). Here, the deployment of the staging area would trigger the data importation process. This alternative

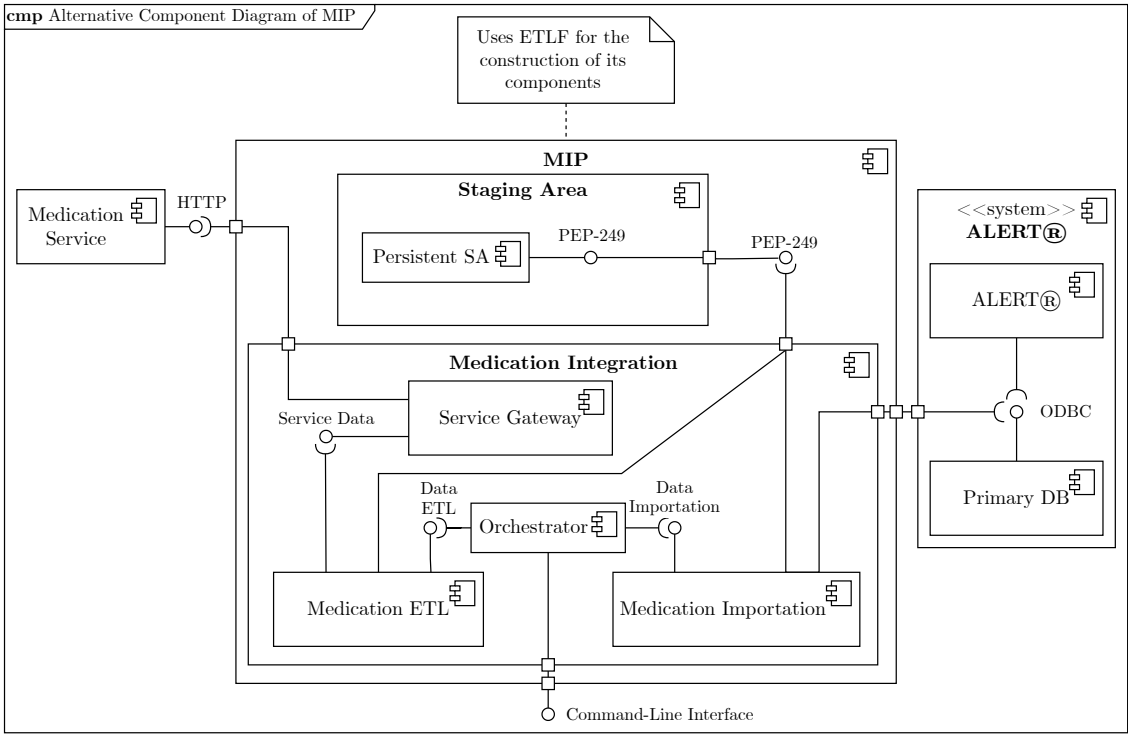

FIGURE 5.7: Component diagram of MIP without an in-memory staging area (Level 3 of C4).

enables the reuse of the same staging area across all clients that operate within the same market. On the contrary, it would increase the complexity level of the overall integration (*-MUEU*) and slightly increase chances of failure (*-PRR*).

Additionally, both alternatives could be merged into a single one (Figure 5.11). This would mean that *Medication ETL* was executed in the ALERT server, the staging area deployed, and then the *Medication Importation* process used this deployed database within a containerized environment.

5.4 Code

With the main components of the system detailed, it is now time to zoom in and evidence finer-grained details of the described components (level 4 of C4). To do so, logical and process views have been created and described in the next two subsections.

5.4.1 Logical View

To be able to better describe (a) MIP as a software platform for the creation of integration processes, and (b) ETL processes constructed using MIP, these have been separated into two groups (with and without details) and each group describes two different elements:

- *MIP Reusable Classes* — describes MIP's classes destined to be reused across multiple ETL processes;
- *Medication ETL Classes* — describes classes within MIP that are directly responsible for executing ETL processes.

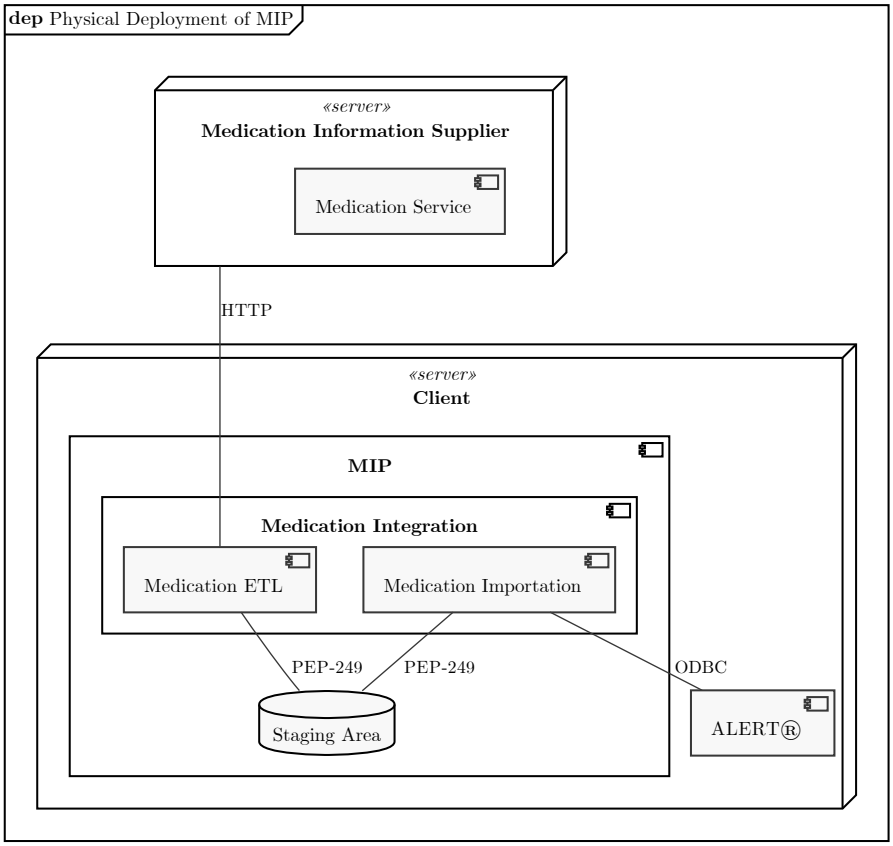

FIGURE 5.8: Physical deployment of MIP (Level 3 of C4).

MIP Reusable Classes Without Details

To better explain how MIP is organized, a class diagram without details is depicted in Figure 5.12. There are two crucial packages in MIP: `core` and `services`. `Core` contains the foundation for all ETL/integration processes:

- *MipExecuter* — responsible for orchestrating medication integration processes;
- *MipBuilder* — responsible for building the staging area, the logger, and all other ETLF components;
- *MipController* — responsible for controlling the workflow of medication ETL processes;
- *Model* — contains all domain logic (i.e., Entities, business data, validations);
- *StagingArea* — operates the staging area, which includes both the in-memory and persistent one;
- *SaLoader* — loads data into the staging area (*FRL*).

`Services` contains separate modules dedicated to the provision of service-oriented components:

- *ServiceConnector* — responsible for connecting to the external service (*FRE*);
- *ServiceParser* — responsible for parsing a service response into a Python dict;

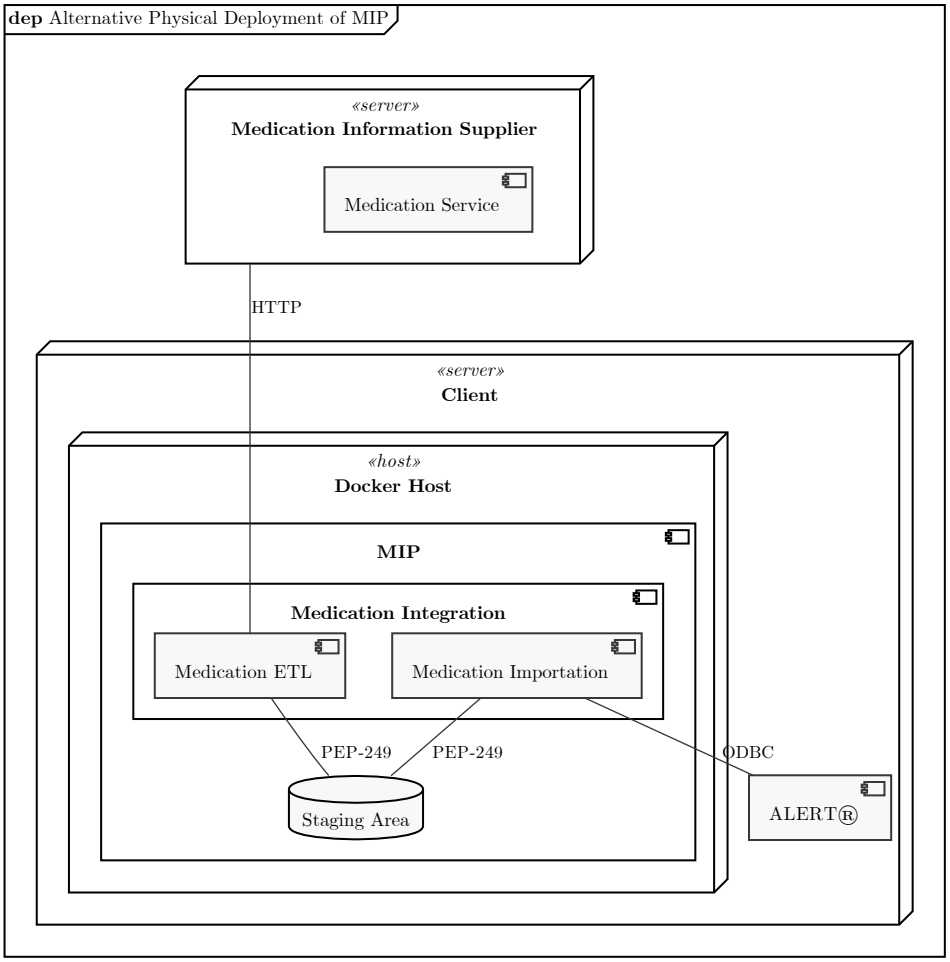

FIGURE 5.9: Alternative physical deployment diagram of MIP using Docker (Level 3 of C4).

- *ServiceMapper* — responsible for integrating service-dependent static Entities.

As one can see, almost all MIP classes are built upon ETLF. Here, not all classes are completely implemented. *MipBuilder* continues to be an abstract class due to the fact that it cannot build components that do not yet exist. The *Model*, *MipExecuter*, *StagingArea*, and *SaLoader* are the only ones that are complete and usable across *all* Medication ETL processes. *MipController* is also complete, but only serves as a way to reuse common functionality across all processes. The *MipExecuter* is responsible for using a given *MipBuilder* and trigger the ETL process (*FRETL*). It is also capable of triggering the importation process if ordered to do so (*FRI*).

With regard to the *Service Gateway* component, one can see that, collectively, the *ServiceConnector* and *ServiceParser* are the entry point to the external service and implement the *Service Gateway* pattern. This means that components that require the *ServiceConnector* to get the external data are completely eluded from the distributed nature of the system as a result of the *Connector* hiding the intricacies of external connections within the code (*+MUEU; +MDRE; +MDRM*). In relation to the *ServiceMapper*, it is only used as the means to transform ALERT predefined configurations, specific to the service (*+MSC*).

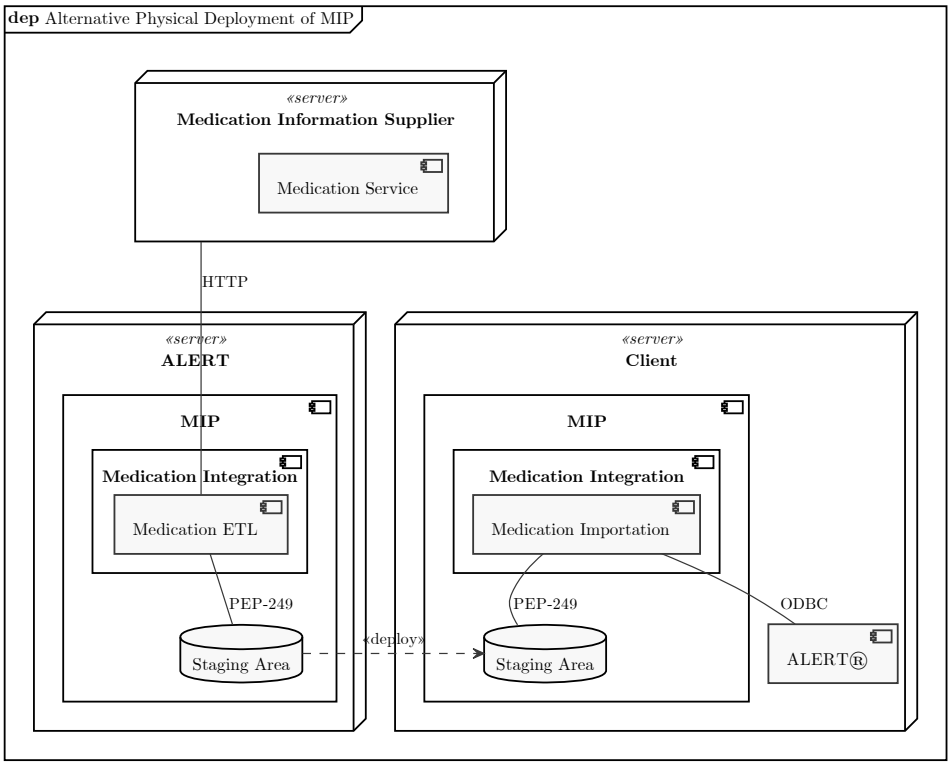

FIGURE 5.10: Alternative physical deployment diagram of MIP, where medication ETL is executed in an ALERT server. (Level 3 of C4).

Medication ETL Classes Without Details

Now, to better explain how Medication ETL classes are organized, a class diagram without details is depicted in Figure 5.13. Each class has its own purpose.

- *ProcessBuilder*: responsible for building all the components of a medication ETL process;
- *ProcessController*: responsible for controlling a specific medication ETL process;
- *ProcessMapper*: transforms primary medication data from the external service format into the internal one (*FRT*).

As it is possible to see, all Medication ETL classes are built upon ETLF and are completely implemented —contrary to what happens in ETLF and MIP. The *ProcessController* is a *MipController* with extended behaviors that answers the needs of each concrete ETL process. The *ProcessMapper* class comprises multiple *Mappers*. Each *Mapper* has its own purpose: to transform *external data* into *internal Entities*. Each one of the *Mappers* makes use of the *ServiceConnector* to fetch data from the service, and of the *StagingArea* to load data into memory. These *Mappers* are triggered either through the *Controller* that contains the ETL workflow logic, or by other *Mappers* that depend on them. When finished the extraction and transformation steps, the *Controller* uses the *StagingArea* to persist all *Entities* into the staging database. Lastly but not least, the *ProcessBuilder* is responsible for building all components associated to the process, which include: *Controller*, *Connectors*, *Parsers*, *Loaders*, *Mappers*, *Staging DB*, and *Logger*.

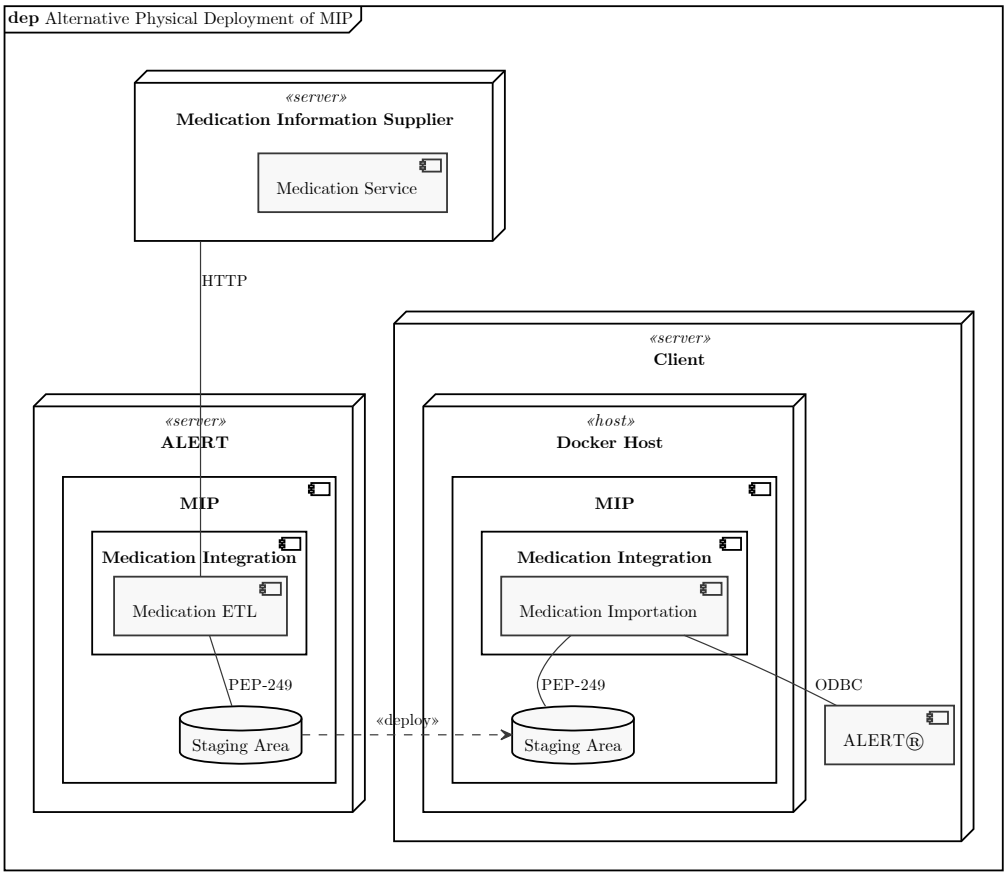

FIGURE 5.11: Alternative physical deployment diagram of MIP, where medication ETL is executed in an ALERT server and the staging area deployed into a containerized environment in the client (Level 3 of C4).

MIP Reusable Classes With Details

Figure 5.14 depicts a class diagram of MIP with a greater level of detail. In ALERT®, medication ETL processes are not all exactly the same. They might differ from one another in the way data is extracted and transformed because each service provider has its own way to provide the service. With that in mind, the `ServiceConnector`, `ServiceParser`, and `ServiceMapper` classes need to properly implement some operations specific to the needs of each supplier. The `MipExecuter`, `InMemorySa`, `Model`, and `SaLoader` are fully implemented classes reusable across all ETL processes, as they do not differ from one to the other. `MipController` provides some default behaviors that are common to all medication ETL processes. These behaviors can then be extended in concrete subclass implementations (*+MDRE*). `EmailUtil` and `MiscUtil` are utilities for email and miscellaneous purposes, reusable across all integration processes.

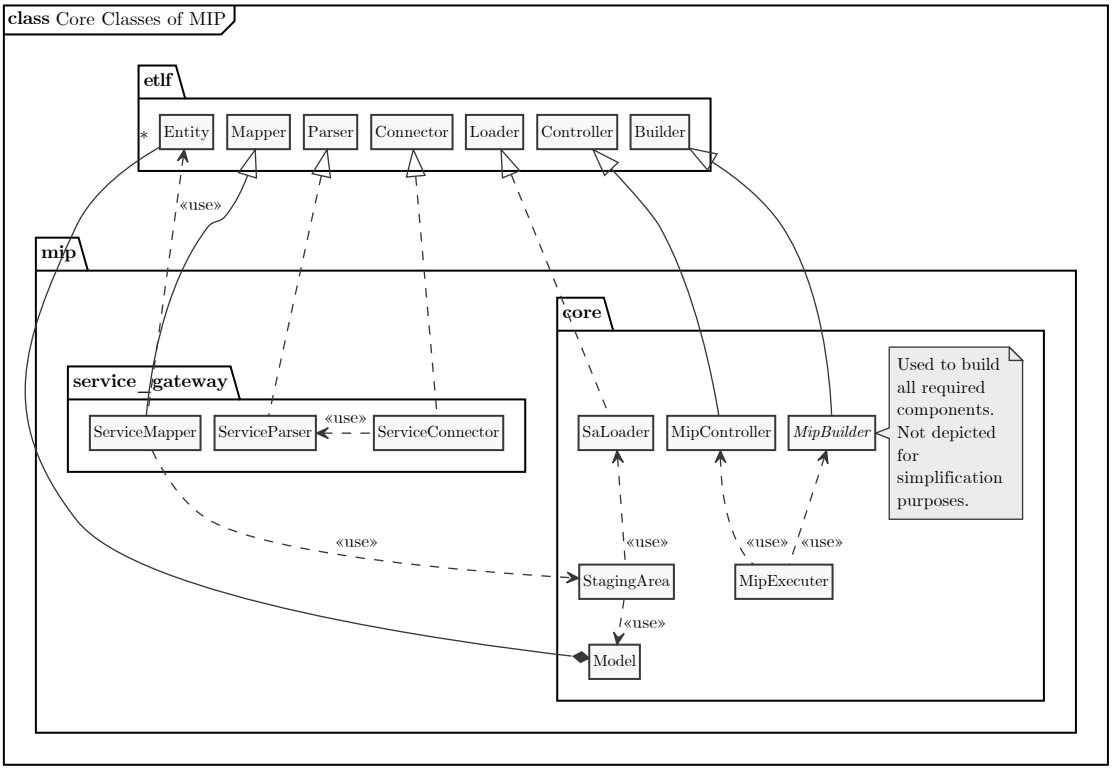

FIGURE 5.12: Class diagram of MIP without detail (Level 4 of C4).

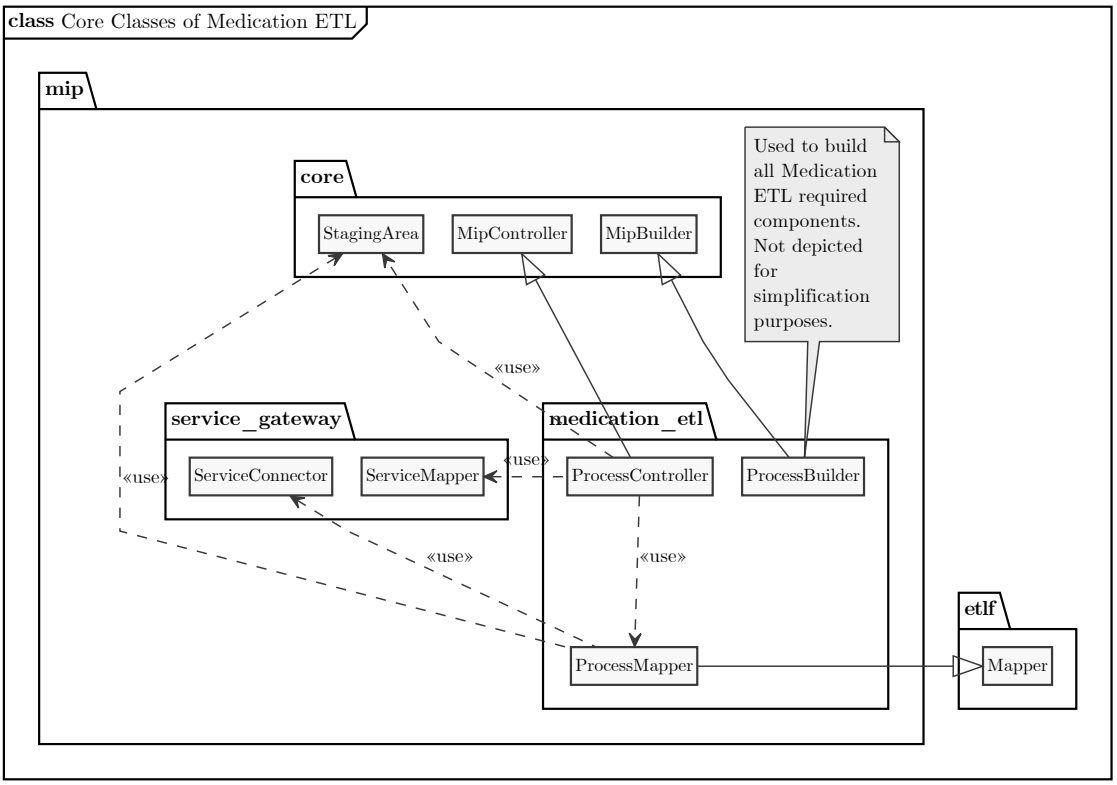

FIGURE 5.13: Class diagram of Medication ETL without detail (Level 4 of C4).

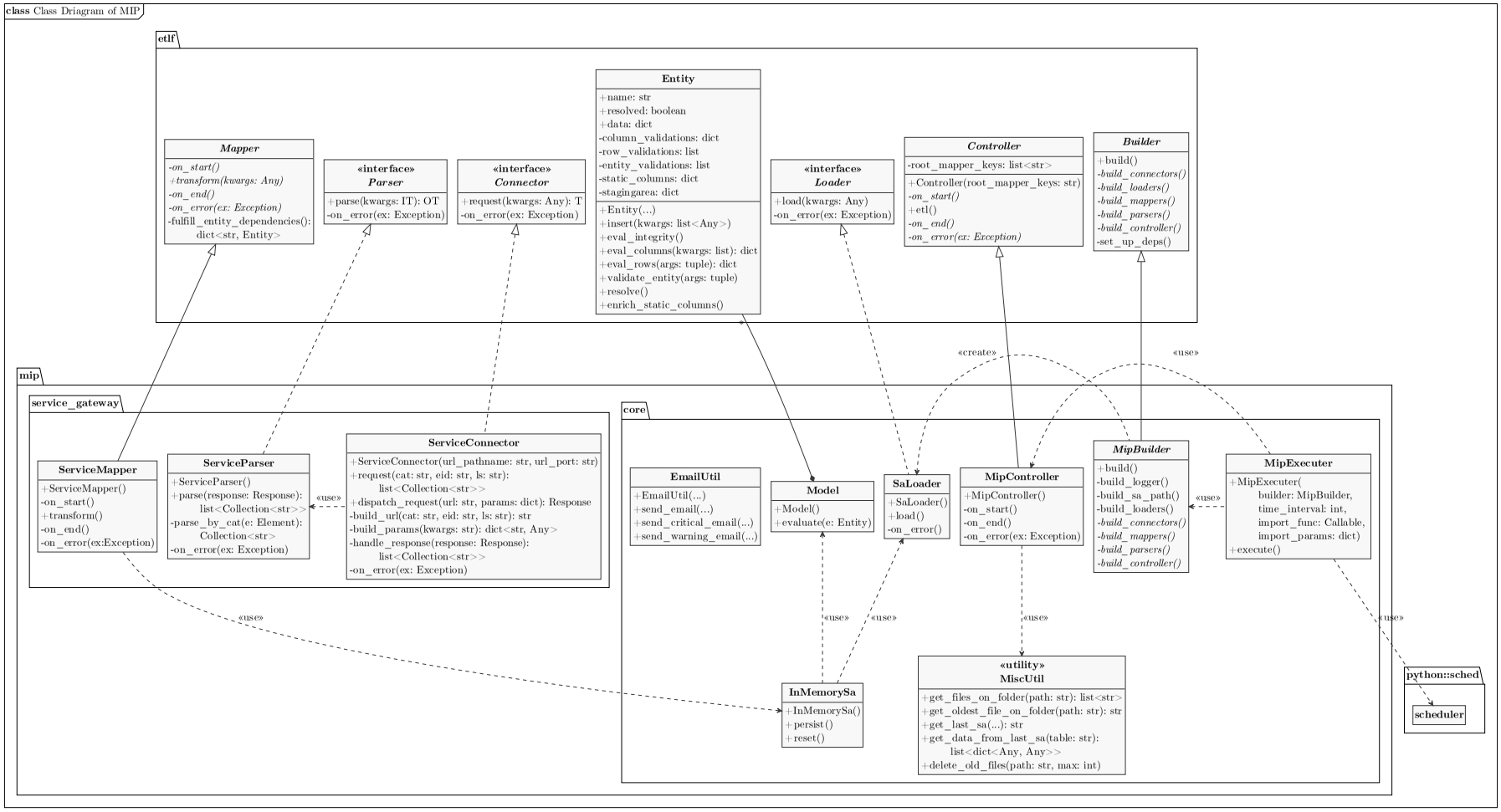

FIGURE 5.14: Class diagram of MIP with detail (Level 4 of C4).

The `MipExecuter` is the orchestrator of all medication integration processes. It requires some external parameters passed on through the constructor. These include the actual `ProcessBuilder` (which is an implementation of the `MipBuilder`), the function that is the entry point to the importation process, parameters to pass into the `import_func`, and execution time intervals. Thereafter, `execute()` uses these variables to properly trigger the execution of the process (*+MFO*). If `time_interval` is not `None`, a periodic execution is triggered using Python's `sched` library[2]. If the `ProcessBuilder` is not `None`, the ETL process is triggered (*FRETL*). If the `import_func` is not `None`, the importation process is triggered (*FRI*).

`MipBuilder` is responsible for setting up the components of each medication ETL process. Except `build_loaders()`, all other building methods are still abstract as this logic depends on the processes at hand. In MIP, the `build()` method is extended so that a logger and persistent SA are properly set up.

The `MipController` controls the overall ETL workflow. In most cases, MIP is only required to integrate medication content, but, sometimes, it might also need to integrate more than that (e.g., allergies). In such cases, the workflow can be to, first, integrate all medication content and then the other types. This can be achieved by creating the required `Mappers` and adding the root ones to the `root_mappers` list thought the constructor of the `Controller` (*+MDRE*). `on_start()` is used for logging (*+PFA*) and for deleting old, no longer needed artifacts. `on_end()` and `on_error()` are only used for logging purposes (*+PFA*). However, this behavior should be extended with other ones later on. The `on_error()` method of `MipController` has an extreme importance to the process because it catches *any* critical failure that might occur to the ETL process. This way, *all* exceptions are properly handled and the process itself raises no runtime error, allowing, therefore, for failures to be contained and for the next execution to be carried out (*+PRR; +PRSur*).

`InMemorySa` contains a `Model` that represents the ALERT staging area using only ETLF `Entities`. The `persist()` operation is used to persist all these `Entities` into the staging database. It uses the `StagingAreaLoader` in order to be able to do so. In MIP, this `Loader` is to be used only by the `InMemorySa` class because *all* data is first loaded and validated in memory and only then persisted into the staging database. This way, data integrity is assured while also increasing the performance of the process (*+PRI; +PPF*). `reset()` is used to reestablish `InMemorySa`'s `Entities` into their original state.

The `ServiceConnector` is a *Request/Response* `Connector` destined to be used only by `Mappers`. The `request()` operation receives all necessary parameters so that an appropriate request can be made. It should be implemented in such a way that the intricacies of external connections are hidden from the other components (*+MUEU; +MDRM*). To be able to do so, the `build_params()`, `build_url()`, `dispatch_request()`, and `handle_response()` operations are used. The *Idempotent Retry* pattern should be applied whenever data requests fail in order to retry the attempts that failed (*+PRR*). In MIP, all `ServiceConnectors` are implemented with the logic of not returning the actual response, but rather an internal object that represents those resources in the form of a `Collection` (*+MUEU; +MDRM*). With that in mind, the `ServiceParser` is used to parse responses from the external service. `parse()` receives a `Response`, which is a collection of resources, and then iterates through its contents and parses them according to

[2]This decision of using a Python built-in scheduler component is a direct consequence of the AHP selection from Appendix A.2.

their category. This approach makes the code more dynamic and more extendable to new types of resources (*+MUEU; +MDRE*). Additionally, all `Parsers` implement a *Tolerant Reader* pattern (*+PRI; +PRR*). To do so, only known resources and required fields are fetched. Even when a required field is not present, the `Parser` raises no exceptions as this logic does not belong to it. In such case, the field is returned with a `None` value for the receiver component to properly handle it. Regarding the `ServiceMapper`, it is only used to replace some default constants that vary depending on the service used.

Alternatives:

- Depending on how ETL processes are designed, it might make sense in some cases to implement `Connectors` with *Asynchronous Response Handlers* (*-PRR; +PPF*). However, if good processing performance is not a requirement, it would be best to only use *Synchronous Response Handlers*, and avoid race conditions in, for example, the integration of *Entity* dependencies.
- The `dispatch_request()` operation could be refactored into another `Connector` to improve code reusability. Perhaps this `RestConnector` could even be a part of ETLF in order to provide it as an out-of-the-box feature.
- A *Control Bus* could be set up to better control configuration, logging, and monitoring in each and every running process (*+MSC; +PFA; +PSM*). Apache Airflow and Luigi come with a built-in UI that allows the handling of these described features. To use this UI, developers would only be required to access the server where Airflow and Luigi are running via browser. However, considering the requirements of the project and the AHP outcome regarding workflow management in Section A.2, this would unnecessarily increase the level of complexity of MIP (*-MUEU*).
- A *Message Store* could be used to save all received messages. In case of a provider-related error, this could ease the discovery of its source (*+PFA; +PSM*). However, it would also increase the workload of the process and, therefore, decrease its performance (*-PPF*). Simply logging what happened for a failure to occur should be sufficient considering the process at hand.

Medication ETL Classes With Details

Figure 5.15 depicts an object diagram for the Medication ETL component with a greater level of detail. As it is possible to see, only the *Builder*, *Controller*, and *Mapper* components need to be created for each actual ETL process. Represented `Mappers` and `Entities` are for demonstration purposes. There are far more than the depicted ones.

Now, the `ProcessBuilder` fully implements ETLF's `Builder` because it has the knowledge of the actual `Controller`, `Connectors`, `Parsers`, and `Mappers` that it requires.

The `ProcessController` extends `MipController` and provides custom behaviors for each Medication ETL. `on_start()` is used to transport required data from the last SA into the current one (*+PRI*). `on_end()` is used to, first, trigger the persistence of the `InMemorySa` (*+MImpRDB*) and then reset it. `on_error()` sends an email whenever critical failures occur to the running process (*+PFN*).

Process Mappers have been devised in such a way that neither the domain nor the connector layers are aware of each other nor of the `Mapper` itself. Analyzing the ones depicted in the diagram, one can see how these are used to extract and transform the required data. In this specific case, the `etl()` function in `ProcessController` contains only a key

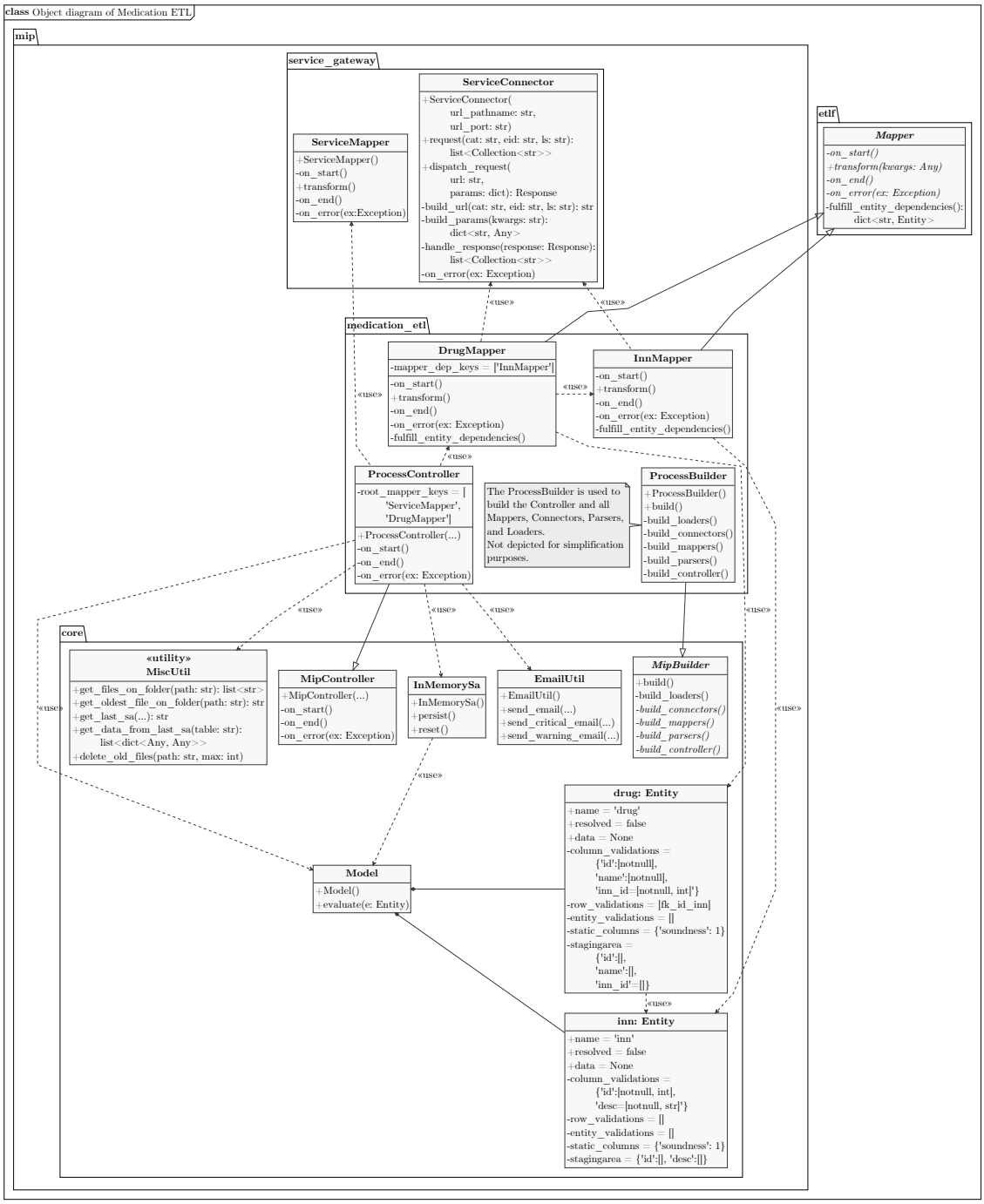

FIGURE 5.15: Object diagram of Medication ETL with detail (Level 4 of C4).

to the `DrugMapper`, which, in turn, will contain a dependency on the `InnMapper`. This means that, first, all data from the INN service is extracted and transformed, creating an `Entity` with all the available INNs in the ALERT® vocabulary. It's only after that that all drugs' information is extracted and transformed, thus enabling for all data to be complete and valid at any given time (*+PRI*). However, this `Mapper` approach has a very simple assumption: the provider has a service that returns *all* INNs. But what if it doesn't? What if INNs can only be fetched for each `drug` resource individually? In

such cases, instead of trying to extract all data required by an `Entity`, it extracts only a single entry of it, being called for all drugs (see the distinction between *EntityMapper* and *EntryMapper* in Section 4.3.3).

5.4.2 Process View

In this subsection a general view is provided regarding how ETL processes are carried out in MIP. To do so, a series of sequence diagrams are created and described next. These only include the so called "*happy path*". Considering that the number of `Mappers` and `Entities` can be quite numerous, instead of depicting the actual ones, only generic ones are used.

In Figure 5.16 a sequence diagram for `MipExecuter` is provided. The `Executer` can be either triggered manually or by any automated mechanism (*+MFO*). If `ti` is not `None` then a periodic execution is triggered. If it is `None`, the process is only execute once.

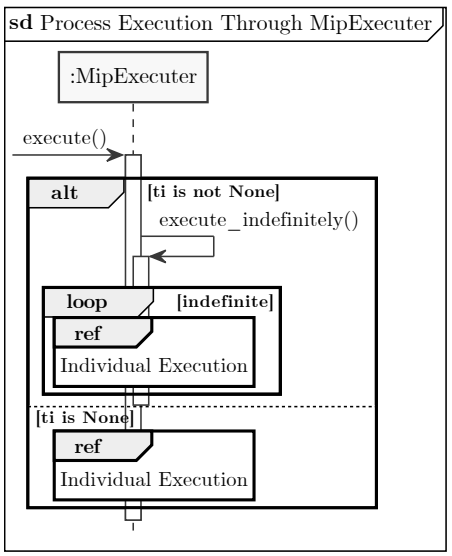

FIGURE 5.16: Sequence diagram of the Executer in Medication ETL (Level 4 of C4).

Figure 5.17 demonstrates how the process execution should work. If the `ProcessBuilder` is not `None`, it is instantiated and then it's `build` operation called. After that, when all ETLF components are properly set up, the ETL process is triggered through the `ProcessController`. When the ETL process is finished, the subsequent importation process is triggered if `import_func` is not `None`.

The `build()` method is responsible for triggering all other building operations that were properly set up using the blueprints provided by ETLF (Figure 5.18). In the end, all ETLF dependencies are injected with their required components (*+MUEU*). This dependency injection will be better explained in the *Builder* implementation (Section 6.1.1).

The actual ETL process only starts with the `Controller`'s `etl()` method (Figure 5.19). The transformation of all identified `root Mappers` is carried out after the execution of the `on_start()` operation. When all `Entities` have been successfully extracted and transformed in-memory, the `Controller` persists the `InMemorySa` and resets it.

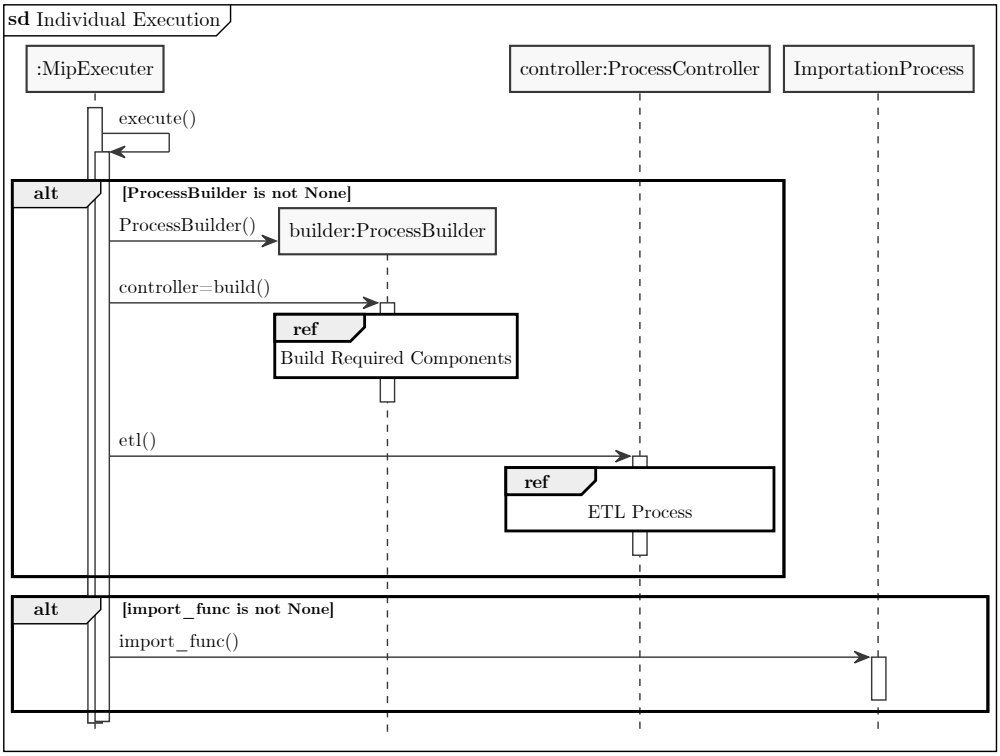

FIGURE 5.17: Sequence diagram of the execution method in Medication ETL (Level 4 of C4).

Almost all `Mappers` have the same workflow that Figure 5.20 presents, which are *EntityMappers*. The only very specific scenario where this *Mapper* approach is not to be used is when integrating an `Entity` one entry at a time. In MIP, *EntityMappers* start off by triggering the transformation of other `Mapper` dependencies through the `fulfill_entity_dependencies()` operation of ETLF. Only when all of its dependencies are fulfilled that the root `Mapper` continues to operate by executing the `transform()` method. This is where almost all the mapping logic for an `Entity` regarding a specific service is located. Fetching all data at once can be problematic at times, so it's more prolific to fetch it in partial blocks (*+PRR; +PPF*). After doing so, the response is mapped and all information iterated. If additional data and operations are required, supplementary requests can be made through `Connectors` and additional transformations carried out through *EntryMappers* (*+PRI*). When all of this is accomplished, rows can then be inserted into the `Entity`. In the end, after all entries for all requests have been transformed, all data is properly evaluated (*+PRI*). If it contains invalid entries, these are updated to a `soundness` of `0` and a warning notification is triggered (*+PRI; +PFN*).

Regarding how `Connectors` are used to make external calls, Figure 5.21 depicts how one `Connector` for a specific service is implemented. A `Mapper` requests data by specifying which resource's category it requires. Then, the `ServiceConnector` properly builds the URL and parameters of the request by taking into consideration the parameters inputted into the method. After that, the request is dispatched and the response handled into a `ServiceParser` that fetches only the required information and delivers it back. Although not depicted in the diagram, *linked services* and resources' identification should be also considered by the `ServiceConnector` to fetch more specific information.

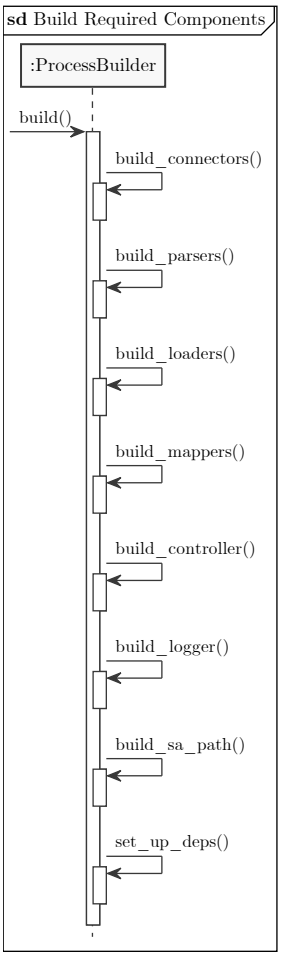

FIGURE 5.18: Sequence diagram of the Builder in Medication ETL (Level 4 of C4).

EntryMappers are just like any other *Mapper*, but with one fundamental difference: instead of being created to integrate multiple rows of external data, it integrates a single entry. Figure 5.22 presents how this kind of *Mapper* is used. Basically, the `transform()` operation receives some sort of identification for the *Mapper* to use in order to know exactly what entry it needs to extract and transform.

Alternatives:

- `Mappers` could validate data as soon as it was mapped into the ALERT® vocabulary. But considering that in ALERT® even invalid medication is to be integrated with a `soundness` of `0` and that, in the end, the complete table needs to also be validated, it was decided that all validations would be performed in the end.
- `Connectors` could return a model object right away and, therefore, contain some mapping logic.

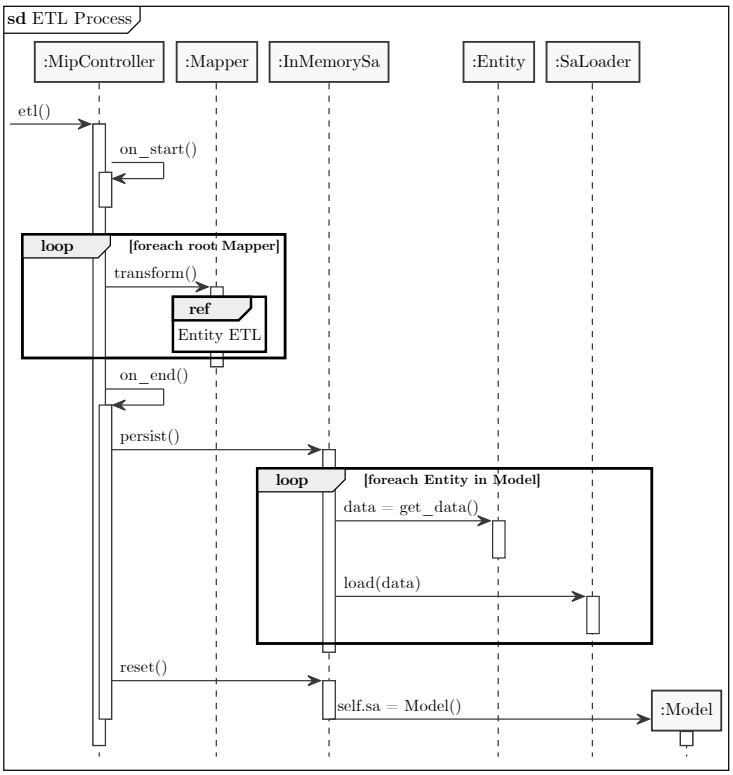

FIGURE 5.19: Sequence diagram of the Controller in Medication ETL (Level 4 of C4).

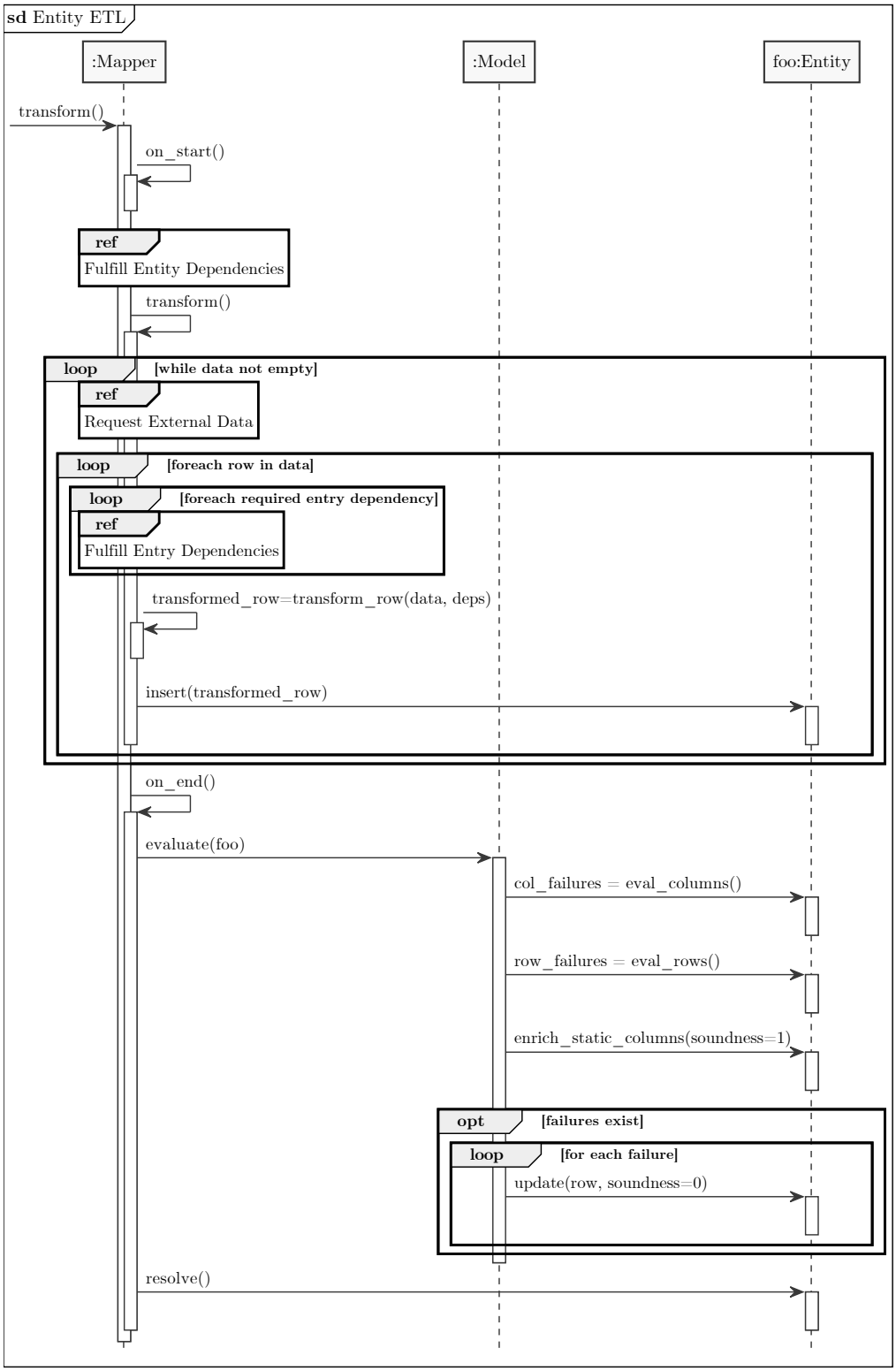

FIGURE 5.20: Sequence diagram of an Entity Mapper in Medication ETL (Level 4 of C4).

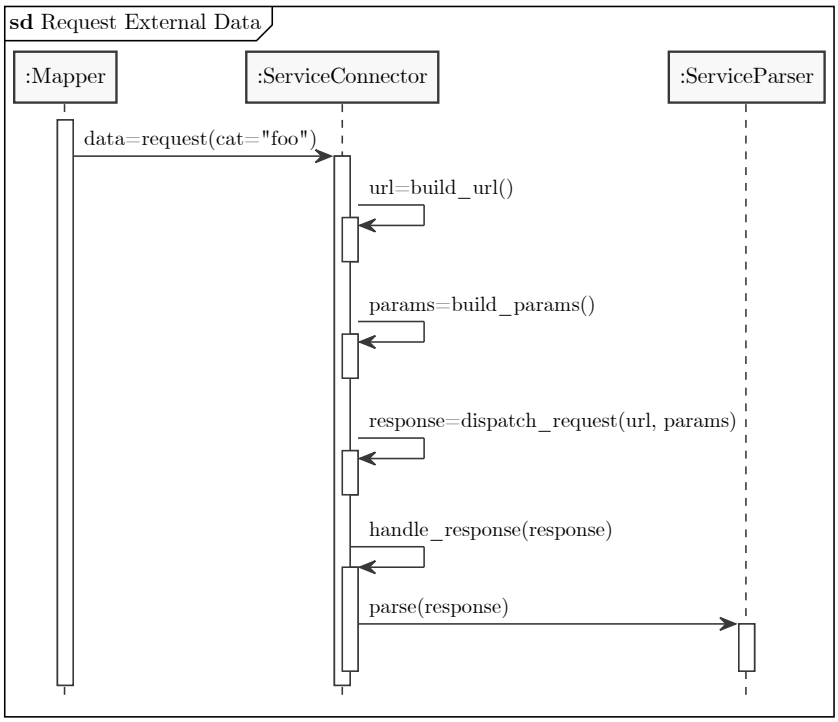

FIGURE 5.21: Sequence diagram of the Executer in Medication ETL (Level 4 of C4).

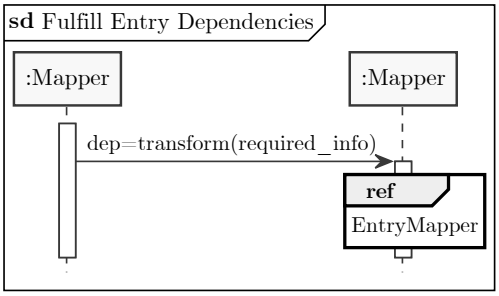

FIGURE 5.22: Sequence diagram of the Executer in Medication ETL (Level 4 of C4).

Chapter 6

Implementation

In *ETL Framework* (Chapter 4) and *Medication Integration Platform* (Chapter 5) a solution for a reliable data integration has been designed. Now, this chapter describes the implementation of that said solution. Firstly, Sections 6.1 and 6.2 detail the implementation of ETLF and MIP, respectively. Then, Section 6.3 details the implementation of actual medication ETL processes. After that, the software testing approach is described in Section 6.4.

Whenever an implementation decision affects a requirement, the acronym of that requirement (identified in Section 3.3.2) is laid out right next to it between parentheses. When it is positively affected, a positive sign is associated to it. When negatively affected, a negative sign is displayed.

Python is a dynamically typed language, which means that variable types are inferred at runtime. When codebases start growing into several thousand, if not millions of lines, the code can become quite hard to understand. Consider the following function definition: `def do_stuff(foo)`. Can `foo` be a `string`, an `integer`, or both? What does it return? This can only be known by looking at its implementation. However, if the function was defined like `def do_stuff(foo: str) -> int`, one would know for sure that it receives a `string` and returns an `integer`. With that in mind, type verification becomes essential in demanding projects. Because of all of this, `mypy` is used throughout the entire implementation of ETLF and MIP. Mypy is a static type checker for Python that yields "verified documentation" (Lehtosalo 2019). In other words, by providing "a formal language for describing types, and by validating that the provided types match the implementation" (Lehtosalo 2019), `mypy` helps developers fixing and avoiding bugs, refactoring code, ensuring that all types are properly handled, and much more (*+MUEU; +PRI; +PRR*). Apart from that, Python docstrings are also extensively used throughout code implementations. A docstring is a "string literal that occurs as the first statement in a module, function, class, or method definition" (Goodger and G. v. Rossum 2001). They provide in-code documentation for all the structures just described (*+MUEU*).

6.1 ETL Framework

Figure 6.1 depicts a package diagram of `etlf`, which is a completely independent Python module, oblivious to *any* concrete business logic. It contains a total of six public modules: four essential ones and two dedicated to the provisioning of lower level tasks.

The following packages are the four essential ones:

- `domain` — combines components that deal with business operations (i.e., *Entity*, *Mapper*);
- `gateway` — deals with components destined to interact with external resources (i.e., *Connector*, *Loader*);
- `generic` — aggregates generic components that do not belong to any other category (i.e., *Factory*, *Parser*);
- `operation` — provides components that hold knowledge regarding how processes operate (i.e., *Builder*, *Controller*).

The two lower level ones are:

- `handlers` — aggregates Python decorators that automate specific tasks;
- `utils` — provides miscellaneous utility functions.

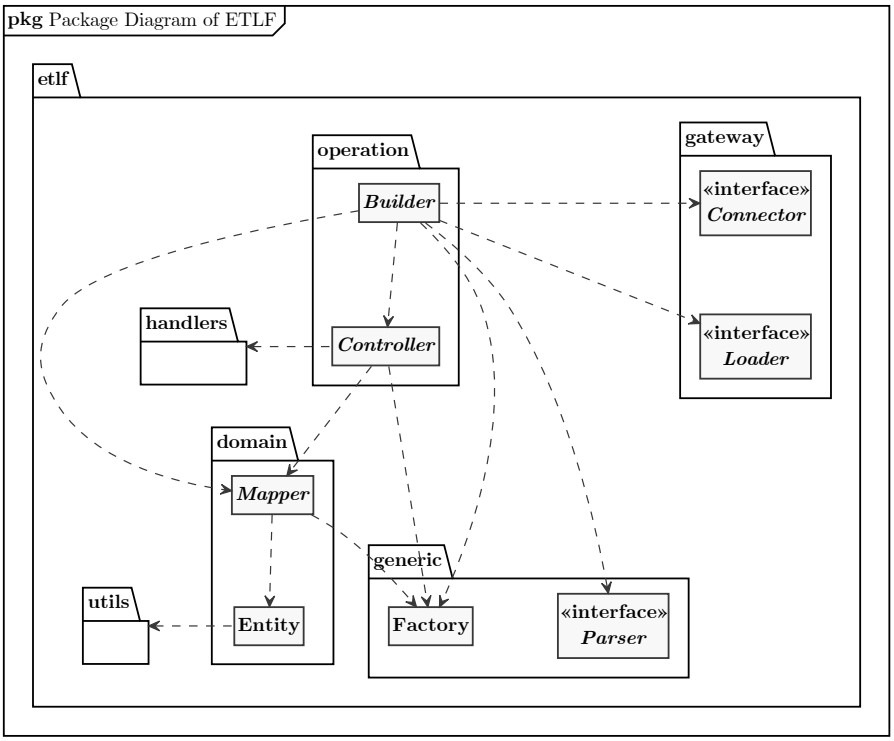

FIGURE 6.1: Package diagram of ETLF.

6.1.1 Builder

The `Builder` is the component responsible for building all the ETLF components that will be used in the ETL process. It is here that the dependency injection mechanism is implemented. After building all components, the `Builder` registers them into `Factories`. When finished, these `Factories` are then injected into the components that require them. Please refer to Section 6.1.4 for more information regarding *Factories*, whose implementation details are quite important to fully comprehend ETLF's benefits. Next follows a list with the components assembled by the `Builder` and their respective dependencies:

- `Connectors` — depend on `Connectors` and `Parsers`;

- `Loaders` — depend on `Connectors` and `Loaders`;
- `Mappers` — depend on `Mappers`, `Connectors`, `Loaders`, and `Parsers`;
- `Parsers` — depend on `Parsers`;
- `Controller` — depends on `Connectors`, `Mappers`, and `Loaders`.

Implemented Methods:

- `build() -> Controller`

  ↪ Builds all the necessary components for the execution of the ETL process (*+MDRE; +MDRM*).

  ↪ Calls all other abstract methods and, thereafter, creates `Factories` for `Connectors`, `Parsers`, `Loaders` and `Mappers`. In the end, each object built is injected with the `Factories` it might require (Listing 6.1).

```python
def build(self) -> Controller:
    # Build all ETLF components
    connectors = self._build_connectors()
    loaders = self._build_loaders()
    mappers = self._build_mappers()
    parsers = self._build_parsers()
    controller = self._build_controller()
    # Set up their dependencies
    self._set_up_deps(connectors, loaders, mappers, parsers, controller)
    return controller


def _set_up_deps(self, connectors, loaders, mappers, parsers, controller) -> None:
    # Build Factories and register products
    cf = Factory[Connector[Any]]()
    cf.register_flt(*connectors)
    lf = Factory[Loader[Any]]()
    lf.register_flt(*loaders)
    mf = Factory[Mapper]()
    mf.register_flt(*mappers)
    pf = Factory[Parser[Any, Any]]()
    pf.register_flt(*parsers)
    # Inject these Factories into the components that require them
    for _, c in connectors:
        c._Connector__inject_deps(cf, pf)
    for _, l in loaders:
        l._Loader__inject_deps(cf, lf)
    for _, m in mappers:
        m._Mapper__inject_deps(cf, lf, mf, pf)
    for _, p in parsers:
        p._Parser__inject_deps(pf)
    controller._Controller__inject_deps(cf, lf, mf)
```

Listing 6.1: Implementation of the build method.

Abstract Methods:

- `_build_connectors() -> list[tuple[str, Connector[Any]]]`

  ↪ Builds all `Connectors` and provides their keys to be registered into a `Factory`.

- `_build_loaders() -> list[tuple[str, Loader[Any]]]`

  ↪ Builds all `Loaders` and provides their keys to be registered into a `Factory`.

- `_build_parsers() -> list[tuple[str, Parser[Any, Any]]]`

  ↪ Builds all `Parsers` and provides their keys to be registered into a `Factory`.

- `_build_mappers() -> list[tuple[str, Mapper]]`

  ↪ Builds all `Mappers` and provides their keys to be registered into a `Factory`.

- `_build_controller() -> Controller`

  ↪ Builds the `Controller` that orchestrates the entire process.

Usage:

Listing 6.2 shows a very basic usage of the ETLF `Builder`. Firstly, the abstract methods need to be properly implemented beforehand. Then, after instantiating the `Builder`, its `build()` operation is called to trigger the construction of all ETLF components. It returns the `Controller` so that the ETL process can be triggered by calling the `etl()` method on it.

```
# NOTE: Foo classes represent implementation of their respective abstractions
class FooBuilder(Builder):
    def _build_connectors(self) -> list[tuple[str, Connector]]:
        return [("ConnectorName", FooConnector())]

    def _build_loaders(self) -> list[tuple[str, Loader]]:
        return [("LoaderName", FooLoader())]

    def _build_mappers(self) -> list[tuple[str, Mapper]]:
        return [("MapperName", FooMapper())]

    def _build_parsers(self) -> list[tuple[str, Parser]]:
        return [("ParserName", FooParser())]

    def _build_controller(self) -> Controller:
        return FooController()


# Instantiate the FooBuilder
builder = FooBuilder()
# Build all components
controller = builder.build()
# Trigger the ETL process
controller.etl()
```

Listing 6.2: Builder usage.

6.1.2 Controller

The `Controller` is the ETLF component responsible for orchestrating the general workflow of ETL processes, which are initiated by calling the `etl()` method.

Initialization Parameters:

- `*root_mapper_keys: str`

↪ A list of the root `Mapper` keys for which to trigger data ETL.

↪ Must reference the same keys that were created in the `Builder`.

↪ Root `Mappers` are `Mappers` that are not dependencies of any other.

Implemented Methods:

- `etl() -> None`

  ↪ Triggers the entire ETL workflow.

  ↪ Iterates through `root_mapper_keys`, triggering the `transform()` operation in each one of them through the `Mapper Factory` injected by the `Builder` (Listing 6.3) (*+MDRE; +MDRM*).

```
def etl(self) -> None:
    for mk in self._root_mapper_keys:
        self._mapper_factory.get(mk).transform()
```

Listing 6.3: Controller's etl() method.

Abstract Methods:

- `_on_start() -> None`

  ↪ Tasks to be performed before triggering the ETL workflow.

  ↪ Implementation in subclasses is optional, see *Handlers* (Section 6.1.9).

- `_on_end() -> None`

  ↪ Tasks to be performed after the ETL workflow.

  ↪ Implementation in subclasses is optional, see *Handlers* (Section 6.1.9).

- `_on_error(ex: Exception) -> None`

  ↪ Handle `etl()` failures, see *Handlers* (Section 6.1.9).

Usage:

Listing 6.4 exemplifies a basic usage of a `Controller`. Firstly, the abstract methods need to be properly implemented beforehand. Then, after instantiating the `Controller` (ideally through the `Builder`), its `etl()` operation is called to trigger the actual ETL process.

6.1.3 Mapper

`Mappers` are abstract classes responsible for mapping specific `Entities` of the domain, thereby controlling how `Entities` are integrated. They are very specific classes that contain very specific logic, which cannot be implemented by the framework itself. Because of this, it defines an abstract `transform()` method that must be implemented in each `Mapper` (see *Abstract Methods*). It is in this kind of methods that ETLF's `Entities` are most predominantly used. Please refer to *Entity* (Section 6.1.8) for more information about them. Additionally, `Connectors` (Section 6.1.5) and `Parsers` (Section 6.1.6) are also expected to be called from within `Mappers`. Apart from that, the framework provides

```python
# NOTE: root Mappers of the process should be defined in the Controller class,
#   which is where this specific logic belongs
class FooController(Controller):

    root_mapper_keys = ["Mapper1", "Mapper2", "Mapper3"]

    def __init__(self) -> None:
        super().__init__(*FooController.root_mapper_keys)

    def _on_start(self) -> None:
        pass  # Do something before the ETL process

    def _on_end(self) -> None:
        pass  # Do something after the ETL process

    def _on_error(self, ex: Exception) -> None:
        pass  # Do something when a critical error occurs


# Build the new Controller (ideally through the Builder)
foo = FooController()
# Now, calling the etl() method on it will trigger the transform() operation of
# "Mapper1", "Mapper2", and "Mapper3" using the Mapper Factory
foo.etl()
```

Listing 6.4: Controller usage.

an approach to systematically trigger dependency fulfillment (see *Implemented Methods*) (*+MUEU; +MDRE; +PRI*).

Initialization Parameters:

- `*mdepkeys: str`
  - ↪ Dependency keys that the current `Mapper` depends on.
  - ↪ Used by the `_fulfill_entity_dependencies()` implemented method.

Implemented Methods:

- `_fulfill_entity_dependencies() -> None`
  - ↪ Fulfill dependencies required by a given `Entity`.
  - ↪ Iterates through `mdepkeys`, triggering the `transform()` operation in each one of them through the `Mapper Factory` injected by the `Builder` (Listing 6.5) (*+MDRE; +MDRM*).
  - ↪ Triggered by applying the `@entity_deps_handler` decorator to `transform()` (see *Handlers* in Section 6.1.9).

```python
def _fulfill_entity_dependencies(self) -> None:
    for dep in self._mdepkeys:
        e = self._mapper_factory.get(dep).transform()
```

Listing 6.5: Fulfilling Entity dependencies.

Abstract Methods:

- `_on_start() -> None`
  - ↪ Tasks to be performed before triggering the transformation of an `Entity`.
  - ↪ Implementation in subclasses is optional, see *Handlers* (Section 6.1.9).
- `transform(**kwargs: Any) -> None`
  - ↪ It is here that all the logic for the transformation between external and internal *Entities* occurs.
  - ↪ `kwargs` are any keyword arguments that the transformation operation requires. They should be only used in *Entry Mappers*, as it will be explained later on in Section 6.3.4.
- `_on_end() -> None`
  - ↪ Tasks to be performed after the transformation of an `Entity`.
  - ↪ Implementation in subclasses is optional, see *Handlers* (Section 6.1.9).
- `_on_error(ex: Exception) -> None`
  - ↪ Handle `transform()` failures, see *Handlers* (Section 6.1.9).

Usage:

Listing 6.6 exemplifies a basic usage of a `Mapper`. Firstly, the abstract methods need to be properly implemented beforehand. `mdepkeys` must properly reference actual `Mappers` that are built through ETLF's `Builder`, or this step will fail. Then, after instantiating the `Controller` (ideally through the `Builder`), its `etl()` operation can be called to trigger the actual ETL process.

6.1.4 Factory

In ETLF, as already referred, `Factories` are used as a way to provide some standard and commonly used software components while also enabling to easily add new customized ones. With that in mind, they can be used in two different purposes. Above all else, they are used by ETLF itself as a way to trigger framework-specific operations (e.g., `Builder.build()`, `Mapper._fulfill_entity_dependencies()`). Secondarily, they can be used by class implementations to call methods on other components without being required to forcefully inject them through constructors (*+MUEU; +MDRE; +MDRM*).

Implemented Methods:

- `register(pk: str, product: T) -> None`
  - ↪ Register a new product into the `Factory`.
- `register_flt(*args: tuple[str, T]) -> None`
  - ↪ Register multiple products from a list of tuples.
- `get(pk: str) -> T`
  - ↪ Find a registered product in the `Factory`.

```python
# NOTE: Mapper dependencies should be defined in an actual Mapper,
#   which is where this specific logic belongs
class FooMapper(Mapper):

    mdepkeys = ["DepMapper1", "DepMapper2", "DepMapper3"]

    def __init__(self) -> None:
        super().__init__(*FooMapper.mdepkeys)

    def _on_start(self) -> None:
        pass  # Do something before the transform() operation

    # NOTE: If desired, apply the @exception_handler decorator (as exemplified) to
    #       trigger _on_error() whenever an error occurs to transform()
    # NOTE: If desired, apply the @workflow decorator (as exemplified) to
    #       trigger _on_start() and _on_end() before and after transform()
    @exception_handler(htype=HandlerType.ETLF)
    @workflow(htype=HandlerType.ETLF)
    def transform(self) -> None:
        # Fetch data using the Connector Factory
        # Parse data using the Parser Factory
        # Insert data into an Entity
        # Validate data using Entity methods
        # Load data using the Loader Factory
        pass

    def _on_end(self) -> None:
        pass  # Do something after the transform() operation

    def _on_error(self, ex: Exception) -> None:
        raise ex  # Do something when an error occurs
```

Listing 6.6: Mapper usage.

Usage:

Listing 6.7 exemplifies how `Factories` can be used by implementations of the `Mapper` class to call methods on other components without being required to forcefully inject them through constructors. The `_connector_factory`, `_parser_factory`, and `_loader_factory` are `Factories` that the `Builder` injects into all `Mappers`. In Listing 6.1 (lines 24 to 32) it is possible to see which `Factories` are inserted into which ETLF components.

```python
class FooMapper(Mapper):
    def transform(self) -> None:
        raw_data = self._connector_factory.get("FooServiceConnector").request()
        parsed_data = self._parser_factory.get("FooServiceParser").parse(raw_data)
        self._loader_factory.get("SaLoader").load(parsed_data)
```

Listing 6.7: Factory usage.

6.1.5 Connector

`Connectors` are interface classes responsible for establishing a connection and, thereby, carry out requests to external services. Just as the *Service Connector Endpoint Pattern*

indicates, multiple `Connectors` may be chained together so that (a) common behaviors are reused, and (b) service-related operations are properly insulated.

Signature Methods:

- `request(**kwargs: Any) -> OT`

  ↪ Deliver a request to an external system.

- `_on_error(ex: Exception) -> Optional[OT]`

  ↪ Handle `request()` failures, see *Handlers* (Section 6.1.9).

6.1.6 Parser

`Parsers` are interface classes responsible for analyzing, processing, and converting data into any format. Fundamentally, they can be used either for low- or high-level purposes, such as parsing raw data from an interchangeable format into a Python object (e.g., `dict`, `list`), or parsing any kind of data into a custom, business-oriented object.

Signature Methods:

- `parse(**kwargs: IT) -> OT`

  ↪ Analyse, process, and convert data.

- `_on_error(ex: Exception) -> Optional[OT]`

  ↪ Handle `parse()` failures, see *Handlers* (Section 6.1.9).

6.1.7 Loader

`Loaders` are interface classes responsible for storing data into any given container. This can be a database, a remote service, or any other relevant ones. Multiple `Loaders` may be chained together so that (a) common behaviors are reused, and (b) loading operations properly insulate business- and software-specific operations.

Signature Methods:

- `load(**kwargs: Any) -> None`

  ↪ Load data into a container.

- `_on_error(ex: Exception) -> None`

  ↪ Handle `load()` failures, see *Handlers* (Section 6.1.9).

6.1.8 Entity

In `etlf`, an instantiated `Entity` is a model object that represents any kind of business-related *Entity*. As previously mentioned, at any given moment, there can be two different types of *Entities*: resolved and non-resolved. *Resolved Entities* are *Entities* whose data structures are final. *Non-resolved Entities* are *Entities* that are not yet final, and, therefore, their data structure is subject to change. The latter kind is characterized by enclosing a staging dictionary to temporarily hold the `Entity`'s data while it is not resolved. Once the `Entity` is resolved, data can no longer be inserted into it. This was an implementation decision to ensure that (a) an `Entity`'s data is evaluated as a whole ($+PRI$), and (b) dependencies can be completely fulfilled before transforming the dependent one ($+PRI$).

Next, the following type aliases will be used to further detail the `Entity`'s implementation for simplification purposes:

- `A = Callable[[Any], bool]`
- `NamedAssertion = tuple[str, A]`
- `TD = dict[str, list[Any]]`

The `Entity`'s data structure if of type `TD`, where keys represent column labels, and their respective values represent the column's data. This means that data is organized by columns and not by rows (*-MUEU; +PPT*). Alternatively, a `list[dict[str, Any]]` could be used to organize data by rows, allowing one to fetch cell data in a more simplistic manner (*+MUEU*). However, having that many dictionaries could cause a memory problem (*-PPT*). Apart from that, the `Entity`'s data structure should not be, in most cases, directly manipulated because using ETLF's provided functions allows one to ensure data integrity (*+PRI*). For now, it is only recommended manipulating it directly when a more sophisticated enrichment is necessary and invalid data that requires treatment is found.

Moreover, `etlf` provides systematic validations as a way to ensure that only valid data is integrated into the system (*+PRI*). This has been achieved by using pure functions as parameters. In `etlf`, all data evaluation operations are, essentially, performed in the same way. Provided by a list of `NamedAssertions`, evaluating functions iterate through that list and execute the assertions whilst providing them with the to-be-evaluated object (Listing 6.8).

```
for name, assertion in named_assertion_list:
    if not assertion(obj):
        # do something with 'name'
```

Listing 6.8: ETLF validation approach.

Initialization Parameters:

- `name`: `str`, optional
  - ↪ A name for the `Entity`.
- `data`: `TD`, optional
  - ↪ Complete data for the `Entity`.
  - ↪ Should contain all the data to fulfill it.
  - ↪ When present, it is assumed that the `Entity` is resolved.
- `columns`: `list[str]`, optional
  - ↪ Column labels that represent the `Entity`'s data.
- `col_vals`: `dict[str, list[NamedAssertion]]`, optional
  - ↪ Assertions for each column that need to be truthful for a cell to be valid.
- `row_vals`: `list[NamedAssertion]`, optional
  - ↪ Assertions for each row that need to be truthful for a row to be valid.

- `entity_vals`: `list[A]`, optional
  ↪ Assertions that need to be fulfilled so that an `Entity` is valid.
- `static_columns`: `dict[str, Any]`, optional
  ↪ Dictionary for to-be-appended static columns.
  ↪ Only used when the `enrich_static_columns()` method is triggered.

Attributes:

- `name`: `str`, `None`
  ↪ The `Entity`'s name.
- `resolved`: `bool`
  ↪ The `Entity`'s current resolution state.
  ↪ `'True'` if resolved, `'False'` if not.
- `data`: `TD`
  ↪ The `Entity`'s data holder.
  ↪ Only available on resolved Entities.
  ↪ Should be considered an immutable object.

Methods:

- `get_sa_size() -> int`
  ↪ Get the current size of the staging dictionary.
- `insert(**kwargs: list[Any]) -> None`
  ↪ Insert data into the staging dictionary.
- `eval_integrity() -> None`
  ↪ Evaluate the integrity of the staging area.
- ```
  eval_columns(
      **kwargs: list[NamedAssertion],
  ) -> dict[str, dict[str, list[int]]]
  ```
  ↪ Evaluate columns.
- `eval_rows(*args: NamedAssertion) -> dict[str, list[int]]`
  ↪ Evaluate rows.
- `validate_entity(*args: NamedAssertion) -> None`
  ↪ Validate the entire `Entity`.
- `resolve() -> None`
  ↪ Resolve the staging dictionary into the `data` attribute.
  ```

Usage

Listing 6.9 shows very roughly how to use the `Entity` class. To ensure data integrity, the `insert()` method safeguards that (a) all Entity-defined columns are present as keyword arguments (*+PRI*), and (b) inserted values must all have the same length (*+PRI*).

```
# 1. Instantiate a new Entity
waldo = Entity(columns=["x", "y"])
# waldo._sa == {
#     "x": [],
#     "y": [],
# }
# waldo.resolved == False
# Trying to access waldo.data raises AttributeError while waldo.resolved is False

# 2. Insert data into 'waldo'
waldo.insert(
    x=["0x", "1x"],
    y=["0y", "1y"],
)
# waldo._sa == {
#     "x": ["0x", "1x"],
#     "y": ["0y", "1y"],
# }
# waldo.resolved == False

# 3. When all data has been inserted, resolve the Entity
waldo.resolve()
# waldo.data == {"x": ["0x", "1x"], "y": ["0y", "1y"]}
# waldo._sa no longer exists
# waldo.resolved == True
```

Listing 6.9: Creating, populating, and resolving Entities.

Alternatively, `Entities` can be also hot-wired with their definitive data, just as Listing 6.10 shows. This can be quite useful for creating `Entities` through configuration constants (*+MSC*).

```
waldo = Entity(
    data={
        "x": ["0x", "1x"],
        "y": ["0y", "1y"],
    }
)
# waldo.data == {"x": ["0x", "1x"], "y": ["0y", "1y"]}
# waldo.resolved == True
```

Listing 6.10: Creating Entities with hot-wired data.

Moving into how to concretely evaluate data validity, Listing 6.11 shows how to use the `eval_columns()` method. It starts of by fetching the column to be evaluated. Then, it iterates through each cell, passing it into the assertion. It's only then that the data for that specific cell is evaluated. When this process is completed for all desirable columns, the method returns a list of failed indexes for each assertion, separated by the column's label.

In the provided example, the `'failures'` variable can be interpreted in the following manner:

> In column `'x'`, evaluated assertions `'x_A1'` and `'x_A2'` contained failures.
> `'x_A1'` failed at index 2, and `'x_A2'` failed at indexes 0 and 4.
> Column `'y'` failed in the `'y_A2'` assertion for indexes 1, 2, and 3.

```
column_labels = ["x", "y"]
# Create a list of assertions for column 'x'
x_al = [
    ("x_A1", lambda cell: type(cell) == int),
    ("x_A2", lambda cell: cell < 5),
]
# Create a list of assertions for column 'y'
y_al = [
    ("y_A1", lambda cell: type(cell) == str),
    ("y_A2", lambda cell: len(cell) == 2),
]
# Associate assertion lists to respective column
cvd = {"x": x_al, "y": y_al}
# Now, initialize the Entity using 'cvd'
waldo = Entity(columns=column_labels, col_vals=cvd)
# Insert data
waldo.insert(
    x=[5, 1, "c", 3, 6],
    y=["0y", "1yyy", "y", "3yyy", "4y"],
)
# Evaluate columns
failures = waldo.eval_columns()
# failures == {
#     "x": {
#         "x_A1": [2],
#         "x_A2": [0, 4],
#     },
#     "x": {
#         "y_A2": [1, 2, 3],
#     },
# }
```

Listing 6.11: Example of evaluating data by columns.

Regarding row validation, Listing 6.12 shows how to systematically validate each row of an `Entity`. Here, the `eval_rows()` method iterates through each row, passing it into assertion, which must be created in such a way that it receives a dictionary containing the row's data. When this process reaches the end of the `TD`, the method returns a list of failed indexes for each assertion. In the provided example, the `'failures'` variable can be interpreted in the following manner:

> Assertion named `'A1'` failed for rows indexed at 2 and 4.
> Assertion named `'A2'` failed for rows indexed at 3 and 4.

Listing 6.13 shows how to validate entire Entities using the `validate_entity()` method. This method should be used to validate criteria that can compromise the `Entity`'s own integrity from a business point of view. Therefore, if any assertion fails, an `Exception` is raised.

```
# Create a list of assertions to be applied to all rows
rvl = [
    ("A1", lambda row: str(row["x"]) in row["y"]),
    ("A2", lambda row: "y" in row["y"]),
]
# Now, initialize the Entity using 'rvl'
waldo = Entity(columns=["x", "y"], row_vals=rvl)
# Insert data
waldo.insert(
    x=[0, 1, 2, 3, 4],
    y=["0y", "1y", "y", "3", "foo"],
)
# Evaluate rows
failures = waldo.eval_rows()
# failures == {
#     "A1": [2, 4],
#     "A2": [3, 4],
# }
```

Listing 6.12: Example of evaluating data by rows.

```
waldo = Entity(columns=["col_x", "col_y"])
waldo.insert(
    x=["0x", "1x"],
    y=["0y", "1y"],
)
waldo.validate_entity(
    ("A1", lambda entity: entity.get_sa_size() > 0),
    ("A2", lambda entity: "foo" in entity._sa["col_x"]),
)
# In this specific case, waldo.validate_entity(...) would raise
#   Exception("'A2' failed") because assertion 'A2' is not truthful
```

Listing 6.13: Example of validating an Entity.

In practice, resolving an `Entity` means checking its integrity and locking its data holder (Listing 6.14). Because of that, before resolving an `Entity`, its data should be first validated. This approach ensures that data dependencies on other Entities are validated consistently (*+PRI*). For now, evaluating the integrity of an `Entity` simply verifies that all columns have the same length. In the future, this can, however, be extended to verify other types of checks (e.g., primary keys, foreign keys).

```
def resolve(self) -> None:
    self.__validate_is_not_resolved()
    self.eval_integrity()
    data = self._sa
    self._data = data
    del self._sa
    self.__resolved__ = True
```

Listing 6.14: Entity resolution approach.

Finally, enriching an `Entity` with static columns is also quite simple, as Listing 6.15

demonstrates. This can be quite useful for whenever a systematic approach is needed to grow the `Entity`'s data schema.

```
waldo = Entity(
    columns=["x", "y"],
    static_columns={"z": True},
)
# waldo._sa == {"x": [], "y": []}
waldo.insert(
    x=["0x", "1x"],
    y=["0y", "1y"],
)
# waldo._sa == {"x": ["0x", "1x"], "y": ["0y", "1y"]}
waldo.enrich_static_columns()
# waldo._sa == {
#     "x": ["0x", "1x"],
#     "y": ["0y", "1y"],
#     "z": [True, True],
# }
```

Listing 6.15: Entity resolution approach.

6.1.9 Handlers

In ETLF, handlers are Python decorators responsible for the execution of certain tasks. There are a total of three handlers, and each one of them can be one of the following types at implementation-level: `ETLF` and `REF`. `ETLF` handlers obey to ETLF predefined contracts. This means that they are located within a class that contains all or at least some of the following methods: `_on_start()`, `_on_end()`, `_on_error()`, and `_fulfill_entity_dependencies()`. `REF` handlers, on the other hand, are handlers specified through a reference to a function. Code-wise, these types are specified as an enumerator to facilitate their usage.

Listings 6.16, 6.17, and 6.18 respectively show the usage of ETLF's three decorators: `@exception_handler`, `@workflow`, and `@entity_deps_handler`. Considering that the `@entity_deps_handler` decorator is too specific, this handler is not expected to be used directly by external parties other than ETLF itself. Apart from that, when the logic to use or not handlers is located within implementations, not all handlers need to be forcefully used. Not even when ETLF forces class implementations to implement methods specific to the handlers. This means that class implementations can decide to just `pass` the method at hand and/or not apply the decorator. This is an implementation decision that forces concrete class implementations to make a conscious decision about that detail.

6.2 Medication Integration Platform

Figure 6.2 depicts a package diagram of `mip`. As can be seen, it encloses three main packages:

- `core` — contains the foundation for the construction of all medication ETL processes (which includes the `MipCfg` and `EmailCfg` configuration modules);
- `services` — contains modules for the extraction and parsing of external data (which includes the `ServiceCfg` configuration module);

```
# Consider the following class:
class Waldo:
    def foo(self):
        pass

    def _on_error(self, ex: Exception):
        pass  # Handle error


# Apply @exception_handler to Waldo.foo() to automatically handle errors with _on_error()
@exception_handler(htype=HandlerType.ETLF)
def foo(self):
    pass


# Alternatively, a reference handler (internal or external to the class) can be used
@exception_handler(htype=HandlerType.REF, handler=bar)  #  bar represents such handler
def foo(self):
    pass
```

Listing 6.16: Exception handler usage.

```
# Consider the following class:
class Waldo:
    def _on_start(self):
        pass  # Tasks to execute before foo()

    def foo(self):
        pass

    def _on_end(self):
        pass  # Tasks to execute after foo()


# Applying @workflow to Waldo.foo() automatically triggers on_start() and on_end():
@workflow(htype=HandlerType.ETLF)
def foo(self):
    pass


# Alternatively, functions can also be passed on by reference.
# @workflow triggers f1(), f2(), foo(), f3(), and f4() in that order
@workflow(htype=HandlerType.REF, before=[f1, f2], after=[f3, f4])
def foo(self):
    pass
```

Listing 6.17: Workflow handler usage.

- `processes` — contains the actual ETL process built using the previous two modules.

6.2.1 MipExecuter

The `MipExecuter` is the *orchestrator* of entire medication integration processes. It is, therefore, the starting point for all of them. This component has been implemented in

```python
# Consider the following class:
class Waldo:
    def foo(self):
        pass

    def _fulfill_entity_dependencies(self):
        pass  # Fulfill entity dependencies code


# Apply @entity_deps_handler to Waldo.foo() to automatically fulfill Entity deps
@entity_deps_handler(htype=HandlerType.ETLF)
def foo(self):
    pass


# To automatically handle Entity deps with a REF function, use the decorator like so:
@entity_deps_handler(htype=HandlerType.REF, handler=bar)
def foo(self):
    pass
```

Listing 6.18: Entity dependencies handler usage.

such a way that the ETL and importation processes can be triggered independently of each other or sequentially (*FRETL; FRI*).

Initialization Parameters:

- `builder`: `MipBuilder`, optional
  - ↪ The ETLF `Builder` used to automate the ETL process execution (*FRETL*).
- `import_func`: `Callable[..., None]`, optional
  - ↪ The function that is the entry point to the importation process (*FRI*).
- `importation_params`: `dict[str, Any]`, optional
  - ↪ Parameters to pass into the `import_func`.
- `ti`: `int`, optional
  - ↪ Time interval between executions in seconds.
  - ↪ If `None`, execution is only triggered once (*+MFO*).

Methods:

- `execute() -> None`
  - ↪ Trigger the execution of the process (Listing 6.19).
  - ↪ If `ti` is not `None`, the process is triggered indefinitely.
  - ↪ If `etl_builder` is defined through initialization parameters, the ETL process is triggered.
  - ↪ If `import_func` is defined through initialization parameters, the importation process is triggered.

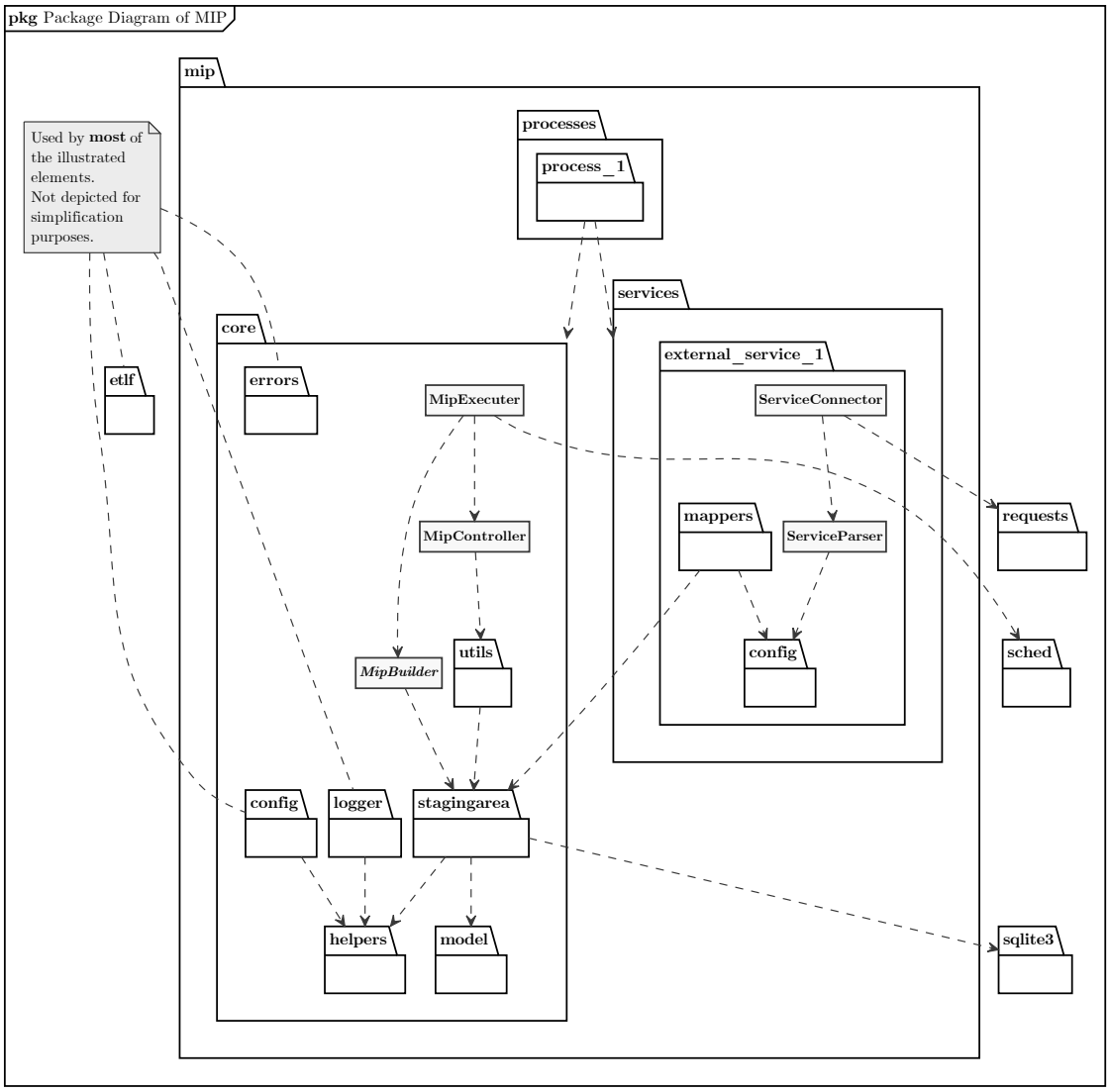

FIGURE 6.2: Package diagram of MIP.

6.2.2 MipBuilder

`MipBuilder` extends ETLF's `Builder`, and is thereby responsible for building the necessary components to carry out medication ETL processes. The `build()` method has been extended so that all medication ETL processes properly build the logger and staging area path.

Implemented Methods:

- `build() -> MipController`

  ↪ Overrides the `Builder`'s `build()` method to (a) create the output directory if it does not yet exist, and (b) build the logger and staging area path for each execution (Listing 6.20).

- `_build_loaders() -> list[tuple[str, Loader[Any]]]`

  ↪ Build all `Loaders` to be used in medication ETL processes.

```
def execute(self) -> None:
    if self._ti:
        self._execute_indefinitely()
    else:
        self._execute_()


def _execute_indefinitely(self) -> None:
    # To properly trigger next execution on the right time
    self.__nextcall__ += self._ti
    # Trigger execution
    self._execute_()
    # When execution finishes, start a Timer that waits until self.__nextcall__
    threading.Timer(
        self.__nextcall__ - time.time(),
        self._execute_indefinitely,
    ).start()


def _execute_(self) -> None:
    # ETL process only triggered if etl_builder has been defined
    if self._etl_builder is not None:
        self._etl_builder().build().etl()
    # Importation process only triggered if import_func has been defined
    if self.import_func is not None:
        self.import_func(**self._import_func_params)
```

Listing 6.19: Execution operation of MipExecuter.

```
def build(self) -> MipController:
    self.CURRENT_TIME = datetime.today().strftime("%H%M%S_%d%m%Y")
    Path(MipCfg.OUT_DIR).mkdir(exist_ok=True)
    self._build_logger()
    self._build_sa_path()
    return super().build()


def _build_logger(self) -> None:
    Path(MipCfg.LOG_DIR).mkdir(exist_ok=True)
    MipLogger.logger = logging.getLogger(self.CURRENT_TIME)
    MipLogger.logger.setLevel(MipCfg.LOGLEVEL)
    LOG_PATH = MipCfg.LOG_DIR + self.CURRENT_TIME + ".mip.log"
    file_handler = logging.FileHandler(LOG_PATH, mode="a")
    format_string = MipCfg.LOGFORMAT
    log_formatter = logging.Formatter(format_string)
    file_handler.setFormatter(log_formatter)
    MipLogger.logger.addHandler(file_handler)


def _build_sa_path(self) -> None:
    Path(MipCfg.SA_DIR).mkdir(exist_ok=True)
    self.SA_PATH = MipCfg.SA_DIR + self.CURRENT_TIME + ".stagingarea.db"
```

Listing 6.20: MipBuilder's overridden build() method.

Abstract Methods:

- `_build_connectors() -> list[tuple[str, Connector[Any]]]`

  ↪ Build all `Connectors` to be used in the medication ETL process.

- `_build_mappers() -> list[tuple[str, Mapper]]`

  ↪ Build all `Mappers` to be used in the medication ETL process.

- `_build_parsers() -> list[tuple[str, Parser[Any, Any]]]`

  ↪ Build all `Parsers` to be used in the medication ETL process.

- `_build_controller() -> MipController`

  ↪ Build the `MipController` to be used in the medication ETL process.

6.2.3 MipController

`MipController` is a fully implemented ETLF `Controller` that reuses common code. As Listing 6.21 shows, it is mainly used for logging (*+PFA*), deleting old artifacts no longer required by ETL processes, and for injecting an email utility that can be used by all implementations. Additionally, `on_end()` and `on_error()` are also used to close the connection to the staging area so that its resources are properly released, avoiding unexpected exceptions (*+PRR*).

```python
class MipController(Controller):
    def __init__(self, email_util: EmailUtil, *root_mappers: str) -> None:
        MipLogger.logger.debug("Initiating MIP controller")
        super().__init__(*root_mappers)
        self._eu = email_util

    def _on_start(self) -> None:
        MipLogger.logger.info("Starting medication ETL process")
        delete_old_files(mip_dir="log")
        delete_old_files(mip_dir="sa")
        MipLogger.logger.debug("Old logs and staging areas cleaned")

    def _on_end(self) -> None:
        MipLogger.logger.debug("Phase 1 of Medication Integration finished")

    def _on_error(self, ex: Exception) -> None:
        tb = traceback.format_exc()
        MipLogger.logger.critical(f"Critical exception raised: {repr(ex)}")
        MipLogger.logger.critical(f"Traceback: {tb}")
```

Listing 6.21: MipController class implementation.

6.2.4 Model

The `Model` is a single class that contains all the business logic directly associated to medication ETL processes of ALERT®. This logic is implemented using ETLF's `Entities`. With that in mind, there is one `Entity` object for each medication-related table of ALERT®'s primary database. Each `Entity` is governed by its own rules, which means that they are all implemented independently of each other. Additionally, special care was also taken regarding data validations. To do so, a special class, entitled `EV`, has been created to hold all sorts of ALERT®-related validations (*+MUEU; +PRI*).

Implemented Methods:

- `__init__() -> None`
  - ↪ Initialize the `Model` object with all ALERT®-related `Entities`.
- `evaluate(e: Entity)`
  - ↪ Evaluate a given `Model Entity`.
  - ↪ Populates the provided `Entity` with a new `soundness` column and proceeds with the evaluation by marking invalid entries with an invalid soundness.
  - ↪ See Listing 6.22 for implementation details.

```
def evaluate(self, e: Entity) -> None:
    def markinvalid(i: int) -> None:
        # Simply updates the soundness column
        e._sa["soundness"][i] = MipCfg.INVALID_SOUNDNESS

    # Populate the Entity with the soundness column
    e.enrich_static_columns()
    # Evaluate columns and then iterate through failed columns
    for col, failuredict in e.eval_columns().items():
        # Iterate through failure items
        for failurename, il in failuredict.items():
            EntityFailureLogger.log_col_failures(e, col, failurename, il)
            # Mark failed indexes as invalid
            for i in il:
                markinvalid(i)
    # Evaluate rows and the iterate through row failure items
    for failurename, il in e.eval_rows().items():
        EntityFailureLogger.log_row_failures(e, failurename, il)
        # Mark failed indexes as invalid
        for i in il:
            markinvalid(i)
    # Evaluate the entire Entity
    try:
        e.validate_entity()
    except Exception as ex:
        EntityFailureLogger.log_entity_failures(e, str(ex))
```

Listing 6.22: Model's evaluate() method.

Example of Validations and Entities:

Listing 6.23 shows an example of *type assertions*, which are used in column validations to validate that a cell can be cast into a specified type (*+PRI*).

Listing 6.24 shows the implementation details of *foreign key validations*, which are also used in column validations to verify if the foreign key refers to a valid target (*+PRI*).

Listing 6.25 shows the implementation details of *foreign key of linking tables validations*. This kind of validation, on the other hand, is a row validation that checks whether an item of the current `Entity` contains at least one valid linked resource in a link table that links the current resource to the foreign entity (*+PRI*).

Listing 6.26 shows an example of a hypothetical *Drug Entity*. It contains three columns: `id_drug`, `desc_drug`, and `id_inn`. None of them can be `null`. `id_drug` and `id_inn` must

```
@staticmethod
def gta(type_: Any) -> A:
    """Generate type assertion."""

    def assertion(cell: str) -> bool:
        if cell:
            try:
                type_(cell)
                return True
            except Exception:
                return False
        else:
            return True

    return assertion


# Cell must not be null
notnull: tuple[str, A] = ("NOTNULL", lambda cell: cell is not None)
# Cell must be castable to int
integer: tuple[str, A] = ("INTEGER", gta(int))
# Cell must be str and can be no bigger than inputted size
varchar: Callable[[int], tuple[str, A]] = lambda size: (
    "VARCHAR(" + str(size) + ")",
    lambda cell: gta(str) and (len(str(cell)) <= size if cell else True),
)
```

Listing 6.23: Type assertions.

```
@staticmethod
def vfk(foreign_entity: Entity, column: str = None) -> tuple[str, A]:
    """Generate a valid foreign key assertion."""

    def assertion(cell: str) -> bool:
        if cell:
            # Get data structure
            td: TD = (
                foreign_entity.data if foreign_entity.resolved else foreign_entity._sa
            )
            # Get primary key of the table
            c: str = column if column else next(iter(td))
            # Get foreign key soundness
            fk_soundness: Optional[int] = (
                td["soundness"][td[c].index(cell)] if cell in td[c] else None
            )
            return bool(fk_soundness)
        return True

    return ("ValidFK", assertion)
```

Listing 6.24: Foreign key validation.

be `integers`. `desc_drug` must be a `string` with no more than `30` characters. `id_inn` must reference a valid INN for a given `drug` to be valid. Additionally, each `drug` item must have at least one valid measure unit associated to it for it to be also valid. The `soundness` static column is the column that keeps track of the validity state of any given

```python
@staticmethod
def lnk_fk(
    lnk_entity: Entity, src_pk: str, foreign_entity: Entity, foreign_pk: str, name: str
) -> tuple[str, A]:
    """Generare a link FK validation."""

    def assertion(row: dict[str, Any]) -> bool:
        # Get the table of the link Entity
        lnk_td: TD = lnk_entity.data if lnk_entity.resolved else lnk_entity._sa
        # Get the table of the foreign Entity
        foreign_td: TD = (
            foreign_entity.data if foreign_entity.resolved else foreign_entity._sa
        )
        # Get an index list of where 'src_pk' is in 'lnk_td'
        lnk_with_src_pk_il: list[int] = [
            i
            for i, v in enumerate(lnk_td[src_pk])
            if v == row[src_pk] and bool(row["soundness"])
        ]
        # Get associated FKs
        lnk_fks: set[str] = {lnk_td[foreign_pk][i] for i in lnk_with_src_pk_il}
        # Get associated FKs' index list in 'foreign_td'
        fpk_il: list[int] = [
            i for i, v in enumerate(foreign_td[foreign_pk]) if v in lnk_fks
        ]
        # Get the 'soundness' of the FKs
        lnk_fpk_soundness_list: list[int] = [
            v for i, v in enumerate(foreign_td["soundness"]) if i in fpk_il
        ]
        # Return true if any of them is valid
        return any(lnk_fpk_soundness_list)

    return (name, assertion)
```

Listing 6.25: Foreign key of linking tables validation.

item in the `Entities` of the `Model`. As already demonstrated in the evaluation method (Listing 6.22), when an `Entity` is evaluated, its data structure is populated with this `soundness` column through the `Entity`'s `enrich_static_columns()` method.

6.2.5 StagingArea

The `stagingarea` is a Python module composed of three different classes: `StagingAreaDao`, `StagingAreaLoader`, and `InMemorySa`.

The `StagingAreaDao` is a data access object used to (a) build and close a connection to the staging area, (b) initialize it with the required tables, and (c) commit changes. The `SaLoader` is an implementation of the ETLF `Loader`, used to directly load data into the staging database.

On the other hand, the `InMemorySa` (Listing 6.27) contains all ETLF's `Entities` instantiated through the `Model`. It has been implemented as a singleton: the constructor simply returns the current object each time it is instantiated, or instantiates a new object if it does not already exist. The `persist()` method is quite important here, as it allows one to trigger the persistence all `Model Entities` through a single command. It has been

```
# self.drug, self.inn, self.lnk_prod_unit_measure, and
#   self.unit_measure are all Entities of the Model
self.drug = Entity(
    name="drug",
    columns=["id_drug", "desc_drug", "id_inn"],
    col_vals={
        "id_drug": [EV.integer, EV.notnull],
        "desc_drug": [EV.varchar(30), EV.notnull],
        "id_inn": [EV.integer, EV.vfk(self.inn), EV.notnull],
    },
    row_vals=[
        EV.lnk_fk(
            name="Has at least one valid measure unit associated",
            lnk_entity=self.lnk_prod_unit_measure,
            src_pk="id_drug",
            foreign_entity=self.unit_measure,
            foreign_pk="id_unit_measure",
        ),
    ],
    static_columns={"soundness": MipCfg.DEF_SOUNDNESS},
)
```

Listing 6.26: Drug Entity example.

implemented using a `Loader` as an argument to not restrict the `InMemorySa` to a single staging database.

```
class InMemorySa(metaclass=Singleton):
    def __init__(self) -> None:
        # Instantiate a new Model object, which contain all Entities
        self.sa = Model()

    def persist(self, loader: Loader) -> None:
        # Fetch all Entities of the Model
        el: list[Entity] = [
            obj for _, obj in vars(self.sa).items() if type(obj) == Entity
        ]
        # For each resolved Entity, load its data using the provided Loader
        for e in el:
            if e.resolved:
                loader.load(table=f"staging_{e.name}", td=e.data)
        loader._sa_dao.commit()

    def reset(self) -> None:
        # Replace self.sa with a new, empty Model object
        self.sa = Model()
```

Listing 6.27: InMemorySa implementation details.

6.2.6 ServiceConnector

The `ServiceConnector` is an implementation of the ETLF `Connector` that follows the *Service Gateway* pattern. For explanation purposes, this described `Connector` is a representation of an actual one that connects to an external medication service. Listing 6.28 shows how the `request` operation has been implemented. The `request()` method,

for this service, can receive a `category`, an `id` for an element to be fetched within the previously specified `category`, a `linked_service`, or even the final `url`. Taking into consideration all of them, the method properly builds the final `url`, and, after that, the request is dispatched. Special care was taken to avoid timeouts at all costs by implementing the *Idempotent Retry* pattern. This means that the request will be dispatched over and over again until a predefined upper threshold is reached (*+PRR*). When a response arrives, it is then sent to the `ServiceParser` for it to properly transform it into a Python object (see Section 6.2.7).

```
class ServiceConnector(Connector):
    @exception_handler(htype=HandlerType.ETLF)
    def request(self, **kwargs: Any) -> Optional[list[Collection[str]]]:
        cat = kwargs["cat"].upper()
        element_id = kwargs.get("element_id", None)
        linked_service = kwargs.get("linked_service", None)
        # If an url is provided, use it directly
        # If not, built the URL for this specific service using the category, element_id,
        #   or linked_service
        url: str = kwargs.get("url", self._build_url(cat, element_id, linked_service))
        params: dict[str, Any] = self._build_params(**kwargs)
        response = self.dispatch_request(url, params)
        self.timeout = ServiceCfg.DEFAULT_TIMEOUT
        return self._pf.get("ServiceParser").parse(response=response.text)

    @exception_handler(htype=HandlerType.ETLF)
    def dispatch_request(self, url: str, params: dict[str, Any]) -> Response:
        try:
            response = self.session.get(url, params=params, timeout=self.timeout)
            return response
        except Timeout as tex:
            # Retries request until max timeout time is reached
            self.timeout = self.timeout + 10
            if self.timeout > ServiceCfg.MAX_TIMEOUT:
                raise CriticalError("Requests reached max defined timeout") from tex
            return self.dispatch_request(url, params)
```

Listing 6.28: ServiceConnector request() implementation.

6.2.7 ServiceParser

The `ServiceParser` is an implementation of the ETLF `Parser` that is specific to an external medication information service (Listing 6.29). The `parse()` method receives the XML response as a `string` and then proceeds to converting it into a list of `Elements`. For each element, it finds out its category and automatically parses it accordingly. It should be emphasized that special care was taken in order to implement a *Tolerant Reader* pattern (*+PRR*). If the category of an element is not found, no exception is launched. Instead, the `_parse_by_cat()` method simply returns `None`. The same applies for parsing child elements of a parent element. If, by any reason, an attribute (e.g., `'id'`, `'name'`, `'desc'`) is not found within the parent element (e.g., `'CAT_1'`, `'CAT_2'`), no exception is raised, and, instead, `None` is returned. Apart from that, it should be also duly noted that this `ServiceParser` implements the *Normalizer* pattern by translating the external format into an internal one, common to all external resources. Additionally, it also implements a *Content Filter* by removing unnecessary information from the response.

```python
class ServiceParser(Parser):
    @exception_handler(htype=HandlerType.ETLF)
    def parse(self, **kwargs: Any) -> list[Collection[str]]:
        response = kwargs["response"]
        el = fromstring(response).findall("atom:entry", ServiceCfg.NAMESPACES)
        pel = [self._parse_by_cat(e) for e in el]
        return [p for p in pel if p is not None]

    @staticmethod
    def _parse_by_cat(e: Element) -> Optional[Collection[str]]:
        cat = ServiceParser._find_cat(e)
        return {
            "CAT_1": ServiceParser._parse_cat_1,
            "CAT_2": ServiceParser._parse_cat_2,
        }.get(cat, (lambda _: None))(e)

    @staticmethod
    def _parse_cat_1(e: Element) -> dict[str, Any]:
        eid = e.find("service:id", ServiceCfg.NAMESPACES)
        ename = e.find("service:name", ServiceCfg.NAMESPACES)
        return {
            "id": eid.text if eid is not None else None,
            "name": ename.text if ename is not None else None,
        }

    @staticmethod
    def _parse_cat_2(e: Element) -> dict[str, Any]:
        eid = e.find("vidal:id", ServiceCfg.NAMESPACES)
        edesc = e.find("vidal:desc", ServiceCfg.NAMESPACES)
        return {
            "id": eid.text if eid is not None else None,
            "desc": edesc.text if edesc is not None else None,
        }
```

Listing 6.29: ServiceParser parse() implementation.

6.2.8 Service Mappers

Service Mappers are implementations of the ETLF `Mapper`. As already referred, these *Service Mappers* are only used to transform service-specific configurations into `Entities`. For example, lets suppose that some services do not provide branded drugs and, instead, only provide the generic ones. For this purpose, the *Service Mappers* module would contain a `Mapper` that deactivates the branded medication type and activates the non-branded one (Listing 6.30).

6.3 Medication ETL

Figure 6.3 depicts a package diagram of a concrete `medication ETL process`. As can be seen, actual ETL processes are contained inside the `processes` package. Each process contains the following elements:

- `alertmi`: Python script responsible for triggering the execution of the processes, which can be periodic or non-periodic;
- `ProcessBuilder`: builds the actual process components;

```python
class ServiceMapper(Mapper):
    @exception_handler(htype=HandlerType.ETLF)
    @workflow(htype=HandlerType.ETLF)
    def transform(self, **kwargs: Any) -> None:
        InMemorySa().sa.available_types.insert(
            id_type=[1, 2],
            desc_type=["NON-BRANDED", "BRANDED"],
            available=[True, False],
        )
```

Listing 6.30: ServiceMapper example.

- `ProcessController`: controls the workflow of a single unique process;
- `mappers`: contains `Mappers` responsible for the transformation of external resources into an internal format;
- `config`: contains configurations specific to the given process (i.e., `ProcessCfg`).

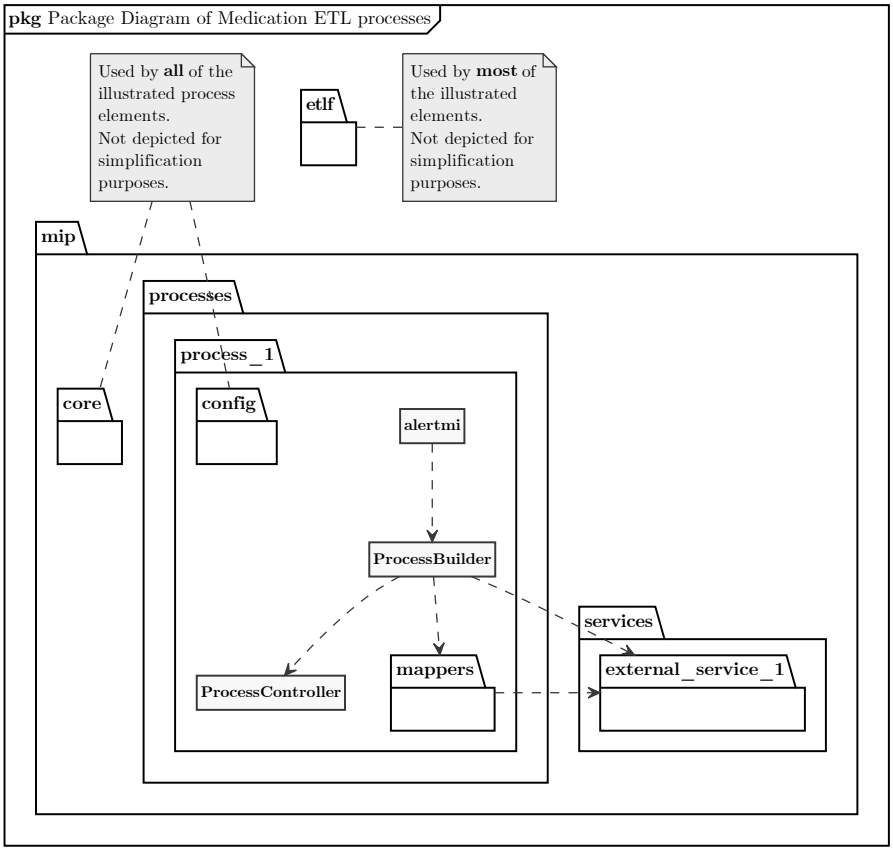

Figure 6.3: Package diagram of a Medication ETL Process.

For description purposes, the examples provided in *ProcessBuilder* (Section 6.3.2), *ProcessController* (Section 6.3.3), and *Process Mappers* (Section 6.3.4) are all interconnected.

6.3.1 Alertmi

`Alertmi` is a Python script that facilitates the high-level manipulation of integration processes. It receives through command-line arguments all the necessary configuration parameters to properly carry out the desired process, which includes the credentials of the primary database. Additionally, through `alertmi`, it is possible to trigger only the medication ETL process, only the medication importation process, or both sequentially. It contains a `main()` method, which is where this logic is actually implemented. Listing 6.31 shows its implementation details.

```
def main(
    only_etl: bool = False,  # provided by command-line arguments
    only_importation: bool = False,  # provided by command-line arguments
    ti: int = None,
) -> None:
    builder: Type[MipBuilder] = ProcessBuilder
    if only_etl:
        # Only trigger the ETL process
        MipExecuter(builder=builder).execute()
    elif only_importation:
        # Only trigger the importation process
        MipExecuter(importation_entry_function=import_func).execute()
    else:
        # Trigger the entire integration process
        MipExecuter(
            builder=builder,
            importation_entry_function=import_func,
            ti=ti,
        ).execute()
```

Listing 6.31: Implementation details of alertmi.

6.3.2 ProcessBuilder

The `ProcessBuilder` is a completely implemented `MipBuilder`, as Listing 6.32 demonstrates. Now, all `Connectors`, `Parsers`, `Loaders`, and `Mappers` of the current ETL process are properly built and returned, to be then handled by the `Builder` superclass.

6.3.3 ProcessController

The `ProcessController` is an implementation of `MipController`, as Listing 6.32 demonstrates. ETL processes start of by triggering the `on_start()` workflow of `MipController`. After that, some ETLF `Entities` of the last successful staging area must be loaded into memory. When that is accomplished, the `etl()` method is triggered, setting off the transformation of the `ServiceMapper`, `MedicationMapper`, and `AllergyMapper` by that order. When done, the `on_end()` workflow of `MipController` is triggered, and the `InMemorySa` persisted and reset to (a) avoid using memory while sleeping ($+PPT$) and (b) properly start over the entire process on the next execution. Whenever a critical error occurs to the process, `on_error()` will be executed, sending an email to whom it may concern at ALERT.

```python
class ProcessBuilder(MipBuilder):
    def _build_connectors(self) -> list[tuple[str, Connector[Any]]]:
        return [("ServiceConnector", ServiceConnector())]

    def _build_mappers(self) -> list[tuple[str, Mapper]]:
        return [
            ("MedicationMapper", MedicationMapper()),
            ("AllergyMapper", AllergyMapper()),
            ("RouteMapper", RouteMapper()),
            ("InnMapper", InnMapper()),
            ("ServiceMapper", ServiceMapper()),
        ]

    def _build_parsers(self) -> list[tuple[str, Parser[Any, Any]]]:
        return [("ServiceParser", ServiceParser())]

    def _build_controller(self) -> MipController:
        return ProcessController()
```

Listing 6.32: ProcessBuilder implementation example.

```python
class ProcessController(MipController):

    root_mappers = ("ServiceMapper", "MedicationMapper", "AllergyMapper")

    def __init__(self, email_util: EmailUtil) -> None:
        super().__init__(email_util, *ProcessController.root_mappers)

    def _on_start(self) -> None:
        super()._on_start()
        self._fetch_data_from_last_sa()

    def _on_end(self) -> None:
        super()._on_end()
        InMemSa().persist(self._lf.get("SaLoader"))
        InMemSa().reset()

    def _on_error(self, ex: Exception) -> None:
        super()._on_error(ex)
        self._eu.send_critical_email(ProcessCfg.LOCATION, ex)
        InMemSa().reset()
```

Listing 6.33: ProcessController implementation example.

6.3.4 Process Mappers

It is here, in the `Mappers` of the ETL process, that the actual medication information is transformed and enriched. Listings 6.34, and 6.35 show two examples of two different types of `Mappers`: *Entity* and *Entry Mappers*.

The `MedicationMapper` is an *Entity Mapper* that has a dependency on the `InnMapper`. This means that before triggering the transformation of `MedicationMapper`, the `InnMapper` will be triggered because of the `@entity_deps_handler` decorator. After that, the `transform()` operation in the `MedicationMapper` iteratively sends requests into the service until it returns no more content. For each response, medication data is iterated and

```
class MedicationMapper(Mapper):

    mdepkeys = ["InnMapper"]

    def __init__(self) -> None:
        super().__init__(*MedicationMapper.mdepkeys)

    def _on_start(self) -> None:
        MipLogger.logger.info("Transforming Service Medication into ALERT Medication")

    @exception_handler(htype=HandlerType.ETLF)
    @workflow(htype=HandlerType.ETLF)
    @entity_deps_handler(htype=HandlerType.ETLF)
    def transform(self, **kwargs: Any) -> Entity[TD]:
        startpage = 0
        while True:
            page += 1
            response = self._cf.get("ServiceConnector").request(cat="MED", page=page)
            if response:
                for med in response:
                    id_route = self._mf.get("RouteMapper").transform(desc=med["route"])
                    InMemSa().sa.medication.insert(
                        id_med=[med["id"]], desc=[med["name"]], id_route=[id_route]
                    )
            else:
                break
        return InMemSa().sa.medication

    def _on_end(self) -> None:
        InMemSa().sa.evaluate(InMemSa().sa.route)
        InMemSa().sa.route.resolve()
        MipLogger.logger.info("All Routes loaded into memory")
        InMemSa().sa.evaluate(InMemSa().sa.medication)
        InMemSa().sa.medication.resolve()
        MipLogger.logger.info("All Medication loaded into memory")

    def _on_error(self, ex: Exception) -> None:
        MipLogger.logger.error("<MedicationMapper>.transform() raised an exception")
        raise CriticalError("<MedicationMapper>.transform() failed") from ex
```

Listing 6.34: Medication Entity Mapper example.

for each medication item a route needs to be inserted through the *EntryMapper*. It is only after that that the medication content is inserted into the `InMemorySa`. After all medication has been transformed and loaded into memory, `on_end()` proceeds to evaluate the `route` and `medication Entities`.

Regarding the `RouteMapper` transformation, as previously referred, the main difference that can be seen is that it requires some kind of parameter to be able to properly integrate its data. In this specific case, it requires the description of the route because, hypothetically, the service contained no specific resources about it. In the end of the entry transformation, the generated route identification is returned to be used by the `MedicationMapper`. As a side note, a problem of generating IDs that are unique across all ALERT® systems has been presented in the *Logical View* of *System Context* (Section 3.4.1). Although not depicted for simplification purposes, this has been achieved in the generic `gen_alert_id()` method by applying BLAKE2 hashes to the description of the

```python
class RouteMapper(Mapper):

    mdepkeys = ["RouteMapper"]

    def __init__(self) -> None:
        super().__init__(*RouteMapper.mdepkeys)

    def _on_start(self) -> None:
        MipLogger.logger.info("Transforming single Service Route into ALERT Route")

    @exception_handler(htype=HandlerType.ETLF)
    @workflow(htype=HandlerType.ETLF)
    def transform(self, **kwargs: Any) -> str:
        desc = kwargs.get("desc")
        id_route = self._gen_alert_id()
        if desc not in InMemSa().sa.route._sa["desc_route"]:
            InMemSa().sa.route.insert(id_route=[id_route], desc_route=[desc])
        return id_route

    def _on_end(self) -> None:
        MipLogger.logger.info("New Route entry loaded into memory")

    def _on_error(self, ex: Exception) -> None:
        MipLogger.logger.error("<RouteMapper>.transform() raised an exception")
        raise CriticalError("<RouteMapper>.transform() failed") from ex
```

Listing 6.35: Route Entry Mapper example.

entry to be integrated.

6.4 Testing

Testing this specific software project is quite important considering that (a) its ultimate goal is to create reliable integration processes, and (b) components of ETLF and MIP will be reused many times often. For ETLF this is even more prominent because it is a very general software framework that can be used not only in medication ETL processes, but in a wide variety areas as long as their ultimate goal is to create a reliable ETL solution. Testing them will allow one to validate that the implemented solution has been properly constructed, leaving a diminished margin for errors (*+PRI; +PRR*). With that in mind, both ETLF and MIP have been extensively tested with *unit* and *integration* tests. *End-to-end* tests have also been created for medication ETL processes to ensure that they are always performing as expected. The `pytest` testing framework has been selected because it scales to support complex functional testing and requires explicit dependency declaration. Additionally, Docker environments were also used to ensure that both unit and end-to-end tests were successful in production-like operating systems (*+MSA*).

Unit and integration tests were created predominantly as a way to check specific component-level behaviors. In ETLF these tests reached a total coverage of 100% out of a total of 429 Python statements (Figure 6.4). Here, it should be duly noted that the excluded statements only included statements inside abstract functions that could never be reached due to how Python itself implements abstract functions. In MIP this coverage was a bit lower, reaching 98% out of a total of 1010 statements (Figure 6.5).

Module ↑	statements	missing	excluded	coverage
etlf/__init__.py	8	0	0	100%
etlf/_errors.py	4	0	0	100%
etlf/_helpers.py	5	0	0	100%
etlf/_typing.py	16	0	0	100%
etlf/domain/__init__.py	3	0	0	100%
etlf/domain/entity.py	129	0	0	100%
etlf/domain/mapper.py	33	0	5	100%
etlf/gateway/__init__.py	3	0	0	100%
etlf/gateway/connector.py	18	0	2	100%
etlf/gateway/loader.py	16	0	2	100%
etlf/generic/__init__.py	3	0	0	100%
etlf/generic/factory.py	19	0	0	100%
etlf/generic/parser.py	14	0	2	100%
etlf/handlers.py	73	0	0	100%
etlf/operation/__init__.py	3	0	0	100%
etlf/operation/builder.py	48	0	5	100%
etlf/operation/controller.py	28	0	1	100%
etlf/utils.py	6	0	0	100%
Total	**429**	**0**	**17**	**100%**

FIGURE 6.4: ETLF coverage.

Module ↑	statements	missing	excluded	coverage
mip/__init__.py	0	0	0	100%
mip/core/__init__.py	0	0	0	100%
mip/core/builder.py	45	0	0	100%
mip/core/controller.py	20	0	0	100%
mip/core/errors.py	2	0	0	100%
mip/core/executer.py	30	4	0	87%
mip/core/helper.py	21	0	0	100%
mip/core/logger.py	31	0	0	100%
mip/core/model.py	133	1	0	99%
mip/core/patterns.py	7	0	0	100%
mip/core/stagingarea.py	78	0	0	100%
mip/core/utils.py	69	0	0	100%
mip/processes/__init__.py	0	0	0	100%
mip/processes/[redacted]/__init__.py	0	0	0	100%
mip/processes/[redacted]/builder.py	27	0	0	100%
mip/processes/[redacted]/controller.py	40	1	0	98%
mip/processes/[redacted]/mappers.py	328	11	0	97%
mip/services/__init__.py	0	0	0	100%
mip/services/[redacted]/__init__.py	0	0	0	100%
mip/services/[redacted]/connector.py	59	0	0	100%
mip/services/[redacted]/mappers.py	23	0	0	100%
mip/services/[redacted]/parser.py	97	0	0	100%
Total	**1010**	**17**	**0**	**98%**

FIGURE 6.5: MIP coverage.

Next follows a list with examples of tests created for exemplification and description purposes, however, Sections 7.3.1 and 7.3.2 of the *Evaluation* chapter contain more examples for validation purposes:

- `test_evaluating_invalid_columns` — tests if evaluating invalid columns returns the right items;
- `test_builder_abstraction` — asserts that `Builder` subclasses properly implement their abstract methods;
- `test_ref_workflow_handler` — tests if the workflow handler decorator properly executes the workflow in the right order;
- `test_vfk_col_val` — tests if the "valid foreign key" assertion is properly validating that foreign columns reference a valid entry;
- `test_imsa_persist` — tests if `InMemorySa` is properly persisting all `Entities`;
- `test_parse_cat_x` — tests if parsing a specific category "x" of a service is outputting the correct data structure;
- `test_request` — tests if the `ServiceConnector` properly requests and parses data with the `ServiceParser`;
- `test_send_critical_email` — tests if critical emails are being sent with the desired information.

End-to-end tests were also created to verify that some requirements of successful and unsuccessful executions are being met. For successful executions, the following requirements and metrics are verified:

- Multiple executions are triggered right on schedule;
- The medication ETL process takes no more than one hour to execute;
- The medication ETL process uses no more than 1 GB of memory;
- Output folders are properly populated;
- Logs are properly built and populated with relevant information;
- Staging area tables that most definitely will not be empty are not empty;
- There are no duplicated entries in linking tables;
- Medication information directly extracted from the service is just like it is provided.

For unsuccessful executions, the following requirements are verified:

- Logs contain information regarding critical errors;
- Emails are properly sent;
- Multiple failed executions are still carried out one after the other.

Chapter 7

Evaluation

In *ETL Framework* (Chapter 4) and *Medication Integration Platform* (Chapter 5) a solution for a reliable data integration has been designed. Following that, implementation details of that solution are provided in Chapter 6. Now, in the present chapter, the designed and implemented solution is evaluated using QEF. Firstly, Section 7.1 details and explains the hypotheses of the current dissertation. After that, the evaluation approach is described in Section 7.2, and, finally, the assessment is conducted in Section 7.3.

7.1 Hypothesis Specification

In Section 1.4, the hypotheses have been only presented. Here, they are detailed to better evaluate the developed solution for the given objectives and problems. If both hypotheses are confirmed, then it is guaranteed that the developed solution is valid and an additional advantage over the existing ones.

H1: MIP is capable of systematically creating new ETL processes that, when avoidable, do not require human interaction.

H2: Integration uses a sustainable ETL process that ensures that external data is valid, complete, and sufficient.

H1 refers specifically to the capabilities of MIP as a solution that allows developers to create new integration/ETL processes without requiring them to be built from scratch over and over again. This is an almost entirely subjective hypothesis that requires human observations to be evaluated. It is derived from the objective of systematically creating new ETL processes, together with the problem of needing a unified codebase for all medication integration functionalities.

H2 refers to specific processes built using MIP. Its evaluation criteria are objective in nature and can be evaluated almost completely through automatic software tests. It was extracted from the objective of ensuring data integrity along with the problem of the current process not fully automated.

7.2 Approach

As previously referred, the method to evaluate the final solution is QEF. The QEF approach provides a systematic way for modelers to evaluate the quality of any artifact in a mathematical sense through a conceptual model. “One of the advantages of using conceptual modeling is that it captures the semantics of an application through the use of a formal notation and this has long proved to be of a significant help in a variety of

domains" (Heidari and Loucopoulos 2014). With QEF, a quality scenario corresponds to an ideal system where all requirements are completely fulfilled. Each quality scenario consists of dimensions, factors, and requirements: dimensions contain factors and factors contain requirements. Each requirement holds a weight and a fulfillment ratio, which add up to their factor's fulfillment. The fulfillment of each factor contributes to the dimension's fulfillment, which, in turn, add up to a final overall ratio. This ratio is the current quality of the evaluated solution, whether it is in its final form or not. It represents how much of the ideal system the actual system has fulfilled.

MIP's assessment is conceived and conducted by myself, the author of the current dissertation, and is divided in two phases. The first one evaluates MIP as a platform that allows the easy development of medication integration processes for new clients/suppliers. The second phase evaluates an actual ETL process built from MIP, in a production-like environment, using real production data provided by the external service. This way, the hypotheses are tested using real, production-ready medication content.

The evaluation criteria is directly derived from the project's specifications provided in Section 3.3.2. This way, the assessment of the solution is based upon the actual specifications on which the current project is built. In addition, special care was taken in order to properly evaluate the ETL process and the quality of data. To do so, relevant metrics were extracted from QoX (Dayal et al. 2009; Simitsis et al. 2009) and QoD (Akoka et al. 2007), which have been already thoroughly detailed in Section 3.3.1. The following evaluation criteria was extracted from QoX: reliability, maintainability, recoverability, robustness, and auditability. The following criteria was extracted from QoD: completeness, uniqueness, consistency, and accuracy.

The assessment scenarios and criteria were validated and approved by the main stakeholder of this project at the end of the evaluation draft. This way, it is guaranteed that they represent the system's intended use and leave no relevant specification out of the assessment process. The criteria weight was also provided by the stakeholder at this point.

The assessment for each one of the hypotheses is to be carried out as follows.

H1: Quality measured considering the overall QEF assessment of MIP.

H2: Quality measured considering the overall QEF assessment of an ETL process.

Quality scenarios for MIP and medication ETL processes are next described in Sections 7.2.1 and 7.2.2 with finer-grained level of detail. Appendix B contains the actual evaluation criteria thoroughly detailed for each one of the scenarios.

7.2.1 Quality Scenario of the Medication Integration Platform

Table 7.1 shows the assessment scenario for MIP. The most important evaluation criteria for it are related to the *Functionality*, *Supportability*, and *Usability* dimensions. Regarding *Functionality*, MIP will be evaluated in relation to the capabilities provided to the developer for (a) *ETL* processes (weight of 60%), and (b) its other *operational* tasks of operating either by schedule or on demand, and notifying on critical failures (weight of 40%). With regard to the *Supportability* dimension, the *adaptability* factor of processes to be executed in relevant operating systems while respecting their clients' configurations will be appraised with a weight of 33%. In addition, this dimension will also weight in, with a percentage of 67%, the *software evolution and maintenance* factor, which will be

measured by its extensibility, modifiability, and automaticity attributes. Finally, the *Usability* dimension will be evaluated with (a) the *easiness* MIP offers to the developer as a medication integration framework (weight of 67%), and (b) the *reusability* of domain model components (weight of 33%). Specifics on the assessment of each requirement are stated in Section B.1.

Table 7.1: QEF quality scenario for the assessment of MIP.

Dimensions	**Factors**	**Requirements**	$\boldsymbol{rw_{jk}}$
Functionality	ETL	ETL1 - Provides data extraction capabilities	10
		ETL2 - Provides transformation capabilities	10
		ETL3 - Provides data persistence capabilities	10
	Operation	O1 - Provides scheduling and on demand execution capabilities	6
		O2 - Provides failure notification capabilities	6
Supportability	Adaptability	A1 - Processes run in different environments	8
		A2 - Processes are configurable	6
	Software Evolution & Maintenance	SEM1 - Easily extended for new data suppliers	8
		SEM2 - Easily extended with new behaviors	8
		SEM3 - Easily modified to answer software changes	2
		SEM4 - Processes require no manual interaction whatsoever	8
Usability	Ease of use	EU1 - The platform is easy to use	6
		EU2 - Interfacing to external systems is eased	4
	Reusability	R1 - Domain logic can be reused	8

Legend:
rw_{jk} *– requirement weight k in factor weight j*

7.2.2 Quality Scenario of the Medication ETL Process

Regarding actual ETL processes, evaluation criteria is described in Table 7.2. The dimensions to be assessed are *Functionality*, *Reliability*, and *Sustainability*. The *Functionality* dimension is assessed in regard to (a) its ETL *functional* requirements (weight of 60%), and (b) its *operational* functionalities of executing either on scheduled time intervals or on demand, and notifying on critical failures (weight of 40%). As for integration *Reliability*, it is subdivided into three factors: *data integrity* (weight of 45%), *robustness* of the process (weight of 45%), and database *security* (weight of 10%). Finally, the *Sustainability* of the process is assessed in relation to (a) the *logging* quality provided (weight of 50%), and (b) its *performance* regarding execution time and memory usage (weight of 50%). Specifics on the assessment of each requirement are stated in Section B.2.

7.3 Assessment

Now that the evaluation approach has been comprehensibly detailed, the assessment of each one of the defined scenarios can be carried out in Sections 7.3.1 and 7.3.2.

Table 7.2: QEF quality scenario for the assessment of medication ETL processes.

Dimensions	Factors	Requirements	rw_{jk}
Functionality	Functional	F1 - Accesses Web services	10
		F2 - Transforms data	10
		F3 - Persists data	10
	Operation	O1 - Executes at stipulated times	6
		O2 - Notifies on critical failure	4
Reliability	Data Integrity	DI1 - Integrated data is complete	10
		DI2 - Integrated data is unique	10
		DI3 - Integrated data is consistent	10
		DI4 - Integrated data is accurate	10
		DI5 - Invalid data has an `INVALID` state	10
	Robustness	R1 - Protected from network failures	6
		R2 - Protected from server errors	6
		R3 - Protected from timeouts	6
		R4 - There are no runtime exceptions	6
		R5 - Scheduled runs are performed after failures	2
	Security	S1 - Database access is controlled	8
Sustainability	Logging	L1 - Error sources are pinpointed using logs	6
		L2 - Stores relevant information	2
	Performance	P1 - Doesn't take too long	2
		P2 - Uses no more than 1GB of memory	4

Legend:
rw_{jk} – requirement weight k in factor weight j

7.3.1 Medication Integration Platform

As previously referred, *H1* is more inclined to a subjective evaluation nature considering that it appraises (a) what components and behaviors MIP provides, and (b) its ease of use. Table B.1 is extensively used so that the QEF assessment is the most objective and systematic as possible. Table 7.3 shows the main output of this assessment.

ETL1 is fulfilled through the creation of an isolated *ServiceGateway* component. This component properly extracts data from the desired external Web service, and can be easily replaced by another one that complies with its *Service Data* interface. *ETL2* is satisfied by a set of three different domain-specific tasks that MIP provides: *transformation*, *enrichment*, and *validation*. Data *transformation* is provided through the *ServiceParser* that translates external data formats into a very generic, common one. Data *enrichment* is granted by MIP's `config` module that contains constants used to enrich the external information with ALERT®-specific mappings. Data *validation* is provided by the `Model` that contains validations for each `Entity` of the ALERT® domain. *ETL3* is realized by the *InMemorySa* component and the `SaLoader`, which, collectively, allow for processes to systematically trigger the persistence of all `Model` Entities. As a whole, *ETL1*, *ETL2*, and *ETL3* completely fulfill the *ETL* factor. *O1* is satisfied by the `MipExecuter` that can either trigger the process periodically if provided with a time interval, or just once if no time interval is provided. *O2* is fulfilled by the `ProcessController` that sends a notification via email whenever a critical exception that could not be properly dealt with during the ETL process is caught. Together, *O1* and *O2* completely satisfy the

Table 7.3: QEF assessment of MIP.

q	Dimensions	Q_i	Factors	W_{ij}	Q_j	Req.	rw_{jk}	wf_k
100%	Functionality	100%	ETL	0,6	100%	ETL1	10	100%
						ETL2	10	100%
						ETL3	10	100%
			Operation	0,4	100%	O1	6	100%
						O2	6	100%
	Supportability	100%	Adaptability	0,33	100%	A1	8	100%
						A2	6	100%
			Software Evolution & Maintenance	0,67	100%	SEM1	8	100%
						SEM2	8	100%
						SEM3	2	100%
						SEM4	8	100%
	Usability	100%	Ease of use	0,67	100%	EU1	6	100%
						EU2	4	100%
			Reusability	0,33	100%	R1	8	100%

Legend:
q – *quality fulfillment of the system*
Q_i – *quality fulfillment of dimension i*
W_{ij} – *factor weight j in dimension i*
Q_j – *quality fulfillment of factor j*
rw_{jk} – *requirement weight k in factor j*
wf_k – *requirement fulfillment k*

Operation factor. With all this in mind, both *ETL* and *Operation* completely satisfy the *Functionality* dimension.

A1 is validated through the end-to-end tests constructed, which are executed without errors in relevant Unix and Windows systems alike. *A2* has been also completely validated, considering that there are four implemented configuration classes, out of a total of four predicted ones (i.e., `MipCfg`, `EmailCfg`, `ServiceCfg`, `ProcessCfg`). Additionally, it should be noted that through the `alertmi` script, additional configurations that change more frequently are set up using command-line parameters. Together, *A1* and *A2* completely realize the *Adaptability* factor. *SEM1* and *SEM3* can be regarded as completed considering that MIP has been constructed using ETLF, and ETLF's components respect the *single responsibility principle* and follow a modular design. Additionally, ETLF's approach on workflow management —through the `Builder`, `Controller`, and `Mapper`— allows one to extend processes without modifying any existing structure, only extending it. *SEM2* can also be regarded as realized considering that ETLF's `@workflow` handler provides a way to easily extend functions with new behaviors before and after their execution. *SEM4* is fulfilled and validated through the end-to-end tests constructed for triggering and verifying sequential process executions. As a whole, *SEM1*, *SEM2*, *SEM3*, and *SEM4* completely fulfill the *Software Evolution and Maintenance* factor. With all this in mind, both *Adaptability* and *Software Evolution and Maintenance* factors completely satisfy the *Supportability* dimension.

EU1 is satisfied considering that *SEM1*, *SEM2*, and *SEM3* are also completely satisfied. *EU2* can be considered as fulfilled because MIP's *ServiceGateway* component is completely isolated from all other ones, and only itself contains the required logic to interact

with outside parties. As a whole, *EU1* and *EU2* completely fulfill the *Ease of Use* factor. *R1* can also be considered as satisfied because MIP's `core` package contains the `Model`, which is reused across all ETL processes built within MIP. The fulfillment of *R1* directly implies the fulfillment of the *Reusability* factor. With all this in mind, both *Ease of Use* and *Reusability* factors completely satisfy the *Usability* dimension.

Considering that all QEF's dimensions have a fulfillment ratio of 100%, the final quality fulfillment of the system will also be 100%. This means that the designed and implemented MIP solution expresses 100% of the ideal MIP system. For this reason alone, *H1* can be considered as completely satisfied. This means that, now, ALERT has a new integration approach that allows developers to systematically create integration processes without requiring them to be build from scratch over and over again. In comparison with the old integration solutions, the codebase is now protected from software evolution and maintenance, and can be easily modified and extended with new behaviors.

7.3.2 Medication ETL Process

As already referred, *H2* is objective in nature and can be evaluated almost completely through automatic software tests considering that it appraises actual medication ETL processes built using MIP. Table B.2 is extensively used so that the conducted QEF assessment is the most objective and systematic as possible. Table 7.4 shows the main output of this assessment.

F1 is completely fulfilled, being validated through the unit and integration tests created that ensure that the `ServiceConnector` properly extracts and parses the data from its specific data provider using the required `ServiceParser`. *F2* is completely satisfied, being validated through the unit tests created that ensure that (a) `ServiceMappers` properly transform and enrich the external data, and (b) the `ServiceParser` properly filters out irrelevant information. *F3* is completely realized, being validated through the unit tests created that ensure that the `SaLoader` properly loads data into the staging area, and the `InMemorySa` properly persists all `Model Entities`. For *F1*, *F2*, and *F3* the end-to-end tests also confirm that such behaviors are being met through the analysis of the final staging area output. Together, *F1*, *F2* and *F3* completely satisfy the *Functional* factor. *O1* is completely satisfied, being validated through the end-to-end tests created that ensure that multiple executions are triggered on the right time. *O2* is completely fulfilled, being validated through the unit tests created that ensure that the `EmailUtil` properly sends emails with all the required content, and that the `ServiceController` properly handles all critical failures by sending an email notification. End-to-end tests also validate that *O2* is being met for multiple failed executions. Together, *O1* and *O2* completely satisfy the *Operation* factor. With all this in mind, both the *Functional* and *Operation* factors completely satisfy the *Functionality* dimension.

Both *DI1* and *DI2* are completely satisfied, being validated through the end-to-end tests and `Model` validations that verify that the number of service provided items is the same as the ones in the staging area. Additionally, *DI1* is also validated through the unit tests on the `Model` component that verify if evaluation is properly carried out. *DI3* is completely fulfilled, being validated through ETLF's validation approach of `Entities`. *DI4* is completely satisfied, being validated through end-to-end tests that verify that all the extracted items that must be exactly like the service provides them, are present in the staging area in that specific manner. *DI5* is completely realized, being validated by unit tests that verify that evaluating `Model Entities` with invalid data properly marks

Table 7.4: QEF assessment of the Medication ETL process.

q	Dimensions	Q_i	Factors	W_{ij}	Q_j	Req.	rw_{jk}	wf_k
96%	Functionality	100%	Functional	0,6	100%	F1	10	100%
						F2	10	100%
						F3	10	100%
			Operation	0,4	100%	O1	6	100%
						O2	4	100%
	Reliability	91%	Data Integrity	0,45	100%	DI1	10	100%
						DI2	10	100%
						DI3	10	100%
						DI4	10	100%
						DI5	10	100%
			Robustness	0,45	100%	R1	6	100%
						R2	6	100%
						R3	6	100%
						R4	6	100%
						R5	2	100%
			Security	0,1	0%	S1	8	0%
	Sustainability	92%	Logging	0,5	100%	L1	6	100%
						L2	2	100%
			Performance	0,5	83%	P1	2	50%
						P2	4	100%

Legend:
q – quality fulfillment of the system
Q_i – quality fulfillment of dimension i
W_{ij} – factor weight j in dimension i
Q_j – quality fulfillment of factor j
rw_{jk} – requirement weight k in factor j
wf_k – requirement fulfillment k

them as being invalid. Together, *DI1*, *DI2*, *DI3*, *DI4* and *DI5* completely satisfy the *Data Integrity* factor. *R1* is completely satisfied, being validated through the unit tests that verify if any error within the `ServiceConnector` is properly caught by the exception handler. *R2* is completely fulfilled, being validated through the unit tests that verify if all expected server-side errors are properly handled. *R3* is completely satisfied, being validated through the unit tests that verify that timed out requests are properly retried until a maximum threshold is reached. *R4* is completely realized, being validated through the end-to-end tests that verify that multiple good executions launch no critical and unexpected exceptions. *R5* is completely satisfied, being validated through the end-to-end tests that verify if the next scheduled execution are still carried out after a critical failure occurs to the current one. Together, *R1*, *R2*, *R3*, *R4* and *R5* completely satisfy the *Robustness* factor. *S1* has been completely not satisfied, meaning that the fulfillment of the *Security* factor was also not achieved. A proper way to directly control accesses to SQLite databases through Python was not discovered. Only unreliable solutions were discovered that require some configuration before setting up processes. These include password protecting the output directory and encrypting the database file when the process finishes. With all this in mind, the *Data Integrity*, *Robustness*, and *Security* factors only satisfy the *Reliability* dimension by a ratio of 91%.

Both *L1* and *L2* are completely satisfied, and, consequently, so is the *Logging* factor. *L1* and *L2* were validated through the unit and end-to-end tests created to ensure that carried out processes log relevant information. *P1* is only partially satisfied at a fulfillment ratio of 50%. This is so because processes have an execution time between 40 and 50 minutes. *P2* is completely realized, because processes use no more than 250 MB of memory. Together, *P1* and *P2* only satisfy the *Performance* factor by a ratio of 83%. With all this in mind, both *Logging* and *Performance* only satisfy the *Sustainability* dimension by a ratio of 92%.

Considering that the *Functionality*, *Reliability*, and *Sustainability* dimensions have fulfillment ratios of, respectively, 100%, 91% and 92%, the final quality fulfillment of the process will be of 96%. This means that the designed and implemented medication ETL process expresses 96% of an ideal one. A percentage of 96% is very close to 100%, and, for this reason, *H2* can be considered as being achieved. This means that, now, ETL processes carry out their tasks efficiently, while guaranteeing that the integrated data is valid, complete, and sufficient at all times. In comparison with the old integration solutions, ETL processes are now failure-resistent and completely automated from top-to-bottom.

Chapter 8

Conclusion

This chapter describes the main conclusions taken from the work accomplished. First, the attained objectives are described. After that, the limitations and future work are presented, and, in the end, a final appreciation is made.

8.1 Attained Objectives

The main objective of this dissertation was to develop a software solution that enabled ALERT to systematically create new integration processes. These processes should ensure the integrity of data and be fault-tolerant. Additionally, such solution must make use of an ETL process that extracts, transforms, and loads the external data into a staging area using an ALERT®-specific format. Thereupon, this staging area shall be used by a separate importation process (oblivious to this dissertation) that properly imports the extracted information into the ALERT® product.

To accurately evaluate the outcome of the current dissertation, two hypotheses have been elaborated and corroborated.

H1: MIP is capable of systematically creating new ETL processes that, when avoidable, do not require human interaction.

H2: Integration uses a sustainable ETL process that ensures that external data is valid, complete, and sufficient.

MIP is, indeed, a software solution that allows a systematic creation of new integration processes that do not require any type of user interaction at runtime. It allows developers to create new integration/ETL processes with a head start, without requiring them to be built from scratch. It provides a core set of functionalities reusable across multiple medication integration processes, while enabling the actual ETL processes to have their own "*personality*". Here, ETLF proved to be a major asset due to the blueprints and systemized operations provided. The actual ETL processes demonstrated to properly validate all extracted information according to predefined domain rules. Additionally, they are also robust, meaning that they try their best to only fail when they really have no other choice. What's more is that all system functionalities detected in *Functional Analysis* (Section 3.4.2) have been accurately implemented thereafter.

Additionally, all other side-objectives have been also fulfilled. Information regarding good programming and integration practices, ETL processes, Web services, and norms in the healthcare IT sector were collected and synthesized in *Background* (Chapter 2). Existing technologies that ease the development of ETL processes were compared in Section 2.4 and selected in Appendix A.

In short, the objectives of the current work can be categorized as rightfully attained, being a practical and advantageous solution achieved.

8.2 Limitations and Future Work

This dissertation's evaluation criteria regarding the performance of ETL processes has a relatively high threshold. This is because (a) that is not a major issue for ALERT, and (b) the external data will never be considerably much more than that. However, if performance had an extreme influence, it could be a major drawback because all data is being loaded into memory at once. This means that the more external information is provided, the more memory the process will require. In such case, the alternative described in Section 5.3.2 of not using the *In-Memory SA* component should be further studied. Additionally, a solution using Python generators to reduce the memory footprint for runtime structures could be also investigated. Apart from that, no solution for controlling accesses to the staging area could be devised. Another limitation detected is related to the dependency fulfillment of `Mappers`. `Mappers` have no knowledge regarding the state of previously transformed *Entities* and, because of that, if two `Mappers` have a common dependency, its fulfillment will be satisfied multiple times. For now, this is avoided by marking the dependency only in the first `Mapper`, or by adding that dependency to the list of root `Mappers`.

ETLF itself has a great margin for evolution. A major improvement would be for the framework to trigger the entire process automatically, including fetching, parsing, and persisting data. This, however, would require an immense effort, considering that ETL processes can be built very differently from one another. Another enhancement would be to transform root and dependency *Mappers* in parallel, which would drastically improve performance. Besides that, ETLF's *Entities* can also be upgraded to support primary keys, foreign keys, and linking *Entities* directly. The *Builder* could warn whenever good programming practices, such as error handling, are not being applied to components. ETLF could also provide *Factories* for *Entities* and miscellaneous objects, which would hot-wire components with more items relevant to them. To improve ETLF's usability, custom wrapper objects, such as `NamedAssertions` instead of `tuple[str, Assertion]`, could be used. The framework could also provide out-of-the-box *Connectors* and *Parsers* for commonly used APIs (e.g., REST, SOAP) and data formats (e.g., JSON, XML).

Both ETLF and MIP could, in the future, provide an approach through *Model Driven Engineering (MDE)* to allow an even more systematic creation of ETL processes through code generation tools. This is because ETLF provides a theoretical, indirect, and simple metamodel of ETL processes: what components are needed, which components those components can interact with, what each component needs to be satisfied, etc. On top of that, it also provides the code base for creating and hot-wiring each one of those components. Bearing this in mind, the current dissertation could have further examined this ETL modeling approach by presenting an actual metamodel of ETL processes and building ETLF upon it.

Following this dissertation, MIP needs minor adjustments and some refactoring in order for it to fully support one legacy ETL process, which is not worth being built again from scratch. This, however, does not affect much of what was built in MIP. The importation process will also need to be refactored into the platform, which, taking into account that it is completely independent, will affect no other component whatsoever. Additionally

and in accordance with MIP's purpose, there are other ETL processes that need to be developed in the future.

8.3 Final Appreciation

From a high perspective, I believe to have developed the required knowledge of research and development work, leading to the ultimate creation of a software product that solves a given problem. I have improved my theoretical and practical software development aspects, which made me enroll into the course in the first place.

In the theoretical side, this dissertation taught me a vast awareness regarding data ETL and integration processes, distributed computing intricacies, and how demanding the healthcare market is in regard to the integrity of medication information.

In the practical side, this project taught me numerous know-hows regarding ETL processes, Python, software development as a tool to achieve specific goals, and how important the requirements are to develop a useful and appropriate solution for the problem at hand.

From a business perspective, this dissertation has allowed me to have a proper integration into the software development world in a work context, while providing ALERT with a solution that will help not only now, but also in the future. ETL processes require no more to be assembled from the ground up, having from now on a common infrastructure that allows for a better scaffolding and maintenance in the long run. Processes also have now a more comfortable way to ensure that the integrated data is reliable at all times. With both ETLF and MIP, ALERT is now able to integrate —when and if required— other types of content using an infrastructure that is similar to the integration of medication.

Bibliography

Akkaoui, Zineb El et al. (July 2013). "A BPMN-Based Design and Maintenance Framework for ETL Processes". In: *International Journal of Data Warehousing and Mining* 9.3, pp. 46–72. DOI: 10.4018/jdwm.2013070103.

Akoka, Jacky et al. (June 2007). "A Framework for Quality Evaluation in Data Integration Systems." In: pp. 170–175.

ALERT (Nov. 2019). *Onboarding Manual*.

– (n.d.[a]). *The ALERT Company*. Accessed January 12, 2021. URL: https://www.alert-online.com/.

– (n.d.[b]). *Who we are*. Accessed January 12, 2021. URL: https://www.alert-online.com/mission-values.

Anderson, James G. (Aug. 1997). "Clearing the way for physicians' use of clinical information systems". In: *Communications of the ACM* 40.8, pp. 83–90. DOI: 10.1145/257874.257895.

Apache Airflow Development Team (2020). *Apache Airflow*. Accessed January 11, 2021. URL: https://github.com/apache/airflow.

Araujo, Samur (2014). "Data integration over distributed and heterogeneous data endpoints". PhD thesis. S.l: Technische Universiteit Delft. ISBN: 9789064647529.

Barany, Gergö (2014). "Python Interpreter Performance Deconstructed". In: *Proceedings of the Workshop on Dynamic Languages and Applications - Dyla'14*. ACM Press. DOI: 10.1145/2617548.2617552.

Barnes, David (2001). *Understanding Business: Processes*. London: Routledge in association with the Open University. ISBN: 9780415238625.

Bentsen, Bent Guttorm (Jan. 1986). "International Classification of Primary Care". In: *Scandinavian Journal of Primary Health Care* 4.1, pp. 43–50. DOI: 10.3109/02813438609013970.

Beyer, Mario et al. (2004). "Towards a flexible, process-oriented IT architecture for an integrated healthcare network". In: *Proceedings of the 2004 ACM symposium on Applied computing - SAC '04*. ACM Press. DOI: 10.1145/967900.967958.

Bicer, Veli et al. (Sept. 2005). "Artemis message exchange framework". In: *ACM SIGMOD Record* 34.3, pp. 71–76. DOI: 10.1145/1084805.1084819.

Biswas, Neepa, Anamitra Sarkar, and Kartick Chandra Mondal (May 2019). "Efficient incremental loading in ETL processing for real-time data integration". In: *Innovations in Systems and Software Engineering* 16.1, pp. 53–61. DOI: 10.1007/s11334-019-00344-4.

Bonobo Development Team (2018a). *Bonobo*. Accessed January 8, 2021. URL: https://docs.bonobo-project.org/en/master/.

– (2018b). *Bonobo*. Accessed January 11, 2021. URL: https://github.com/python-bonobo/bonobo.

– (Jan. 2018c). *F.A.Q.* Accessed February 19, 2021.

Borza, John (2011). "FAST diagrams: The foundation for creating effective function models". In: *General Dynamics Land Systems*.

Brocke, Jan vom, Alan Hevner, and Alexander Maedche (2020). "Introduction to Design Science Research". In: *Progress in IS*. Springer International Publishing, pp. 1–13. DOI: 10.1007/978-3-030-46781-4_1.

Brown, Simon (n.d.). *The C4 model for visualising software architecture*. Accessed June 16, 2021. URL: https://c4model.com/.

Capelo, Mariana and Orlando Belo (2018). "Designing Safe and Reliable ETL Systems Using Alloy". In: *2018 Proceedings*.

Crupi, John (Aug. 2003). "Foreword". In: *Enterprise Integration Patterns*. 1st Edition. Addison Wesley. ISBN: 0321200683.

Czejdo, B. and M. C. Taylor (Oct. 1992). "Integration of Information Systems Using an Object-Oriented Approach". In: *The Computer Journal* 35.5, pp. 501–513. DOI: 10.1093/comjnl/35.5.501.

Czejdo, Bogdan D. and Malcolm C. Taylor (1992). "Integration of object-oriented programming languages and database systems in KOPERNIK". In: *Data & Knowledge Engineering* 7.4, pp. 271–298. ISSN: 0169-023X. DOI: https://doi.org/10.1016/0169-023X(92)90028-A.

Dabroek, M.G. (2016). "Scalable and Reuse-Oriented Data Integration: A Distributed Semi-Automatic Approach". MA thesis. Utrecht University.

Daigneau, Robert (Oct. 1, 2011). *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. ADDISON WESLEY PUB CO INC. 321 pp. ISBN: 032154420X.

Dayal, Umeshwar et al. (2009). "Data integration flows for business intelligence". In: *Proceedings of the 12th International Conference on Extending Database Technology Advances in Database Technology - EDBT '09*. ACM Press. DOI: 10.1145/1516360.1516362.

Deepika, Kumari (2017). "Performance Optimization of ETL Process". en. In: DOI: 10.13140/RG.2.2.13994.44480.

Demirkan, Haluk (Sept. 2013). "A Smart Healthcare Systems Framework". In: *IT Professional* 15.5, pp. 38–45. DOI: 10.1109/mitp.2013.35.

Deneke, Wesley (2012). "A Domain Specific Model for Generating ETL Workflows from Business Intents". PhD thesis. University of Arkansas.

DICOM (n.d.). *About DICOM: Overview*. Accessed January 7, 2021. URL: https://www.dicomstandard.org/about-home.

Dong, Xin Luna and Divesh Srivastava (Aug. 2013). "Big data integration". In: *Proceedings of the VLDB Endowment* 6.11, pp. 1188–1189. DOI: 10.14778/2536222.2536253.

Drozd, Aleksandr, Anna Gladkova, and Satoshi Matsuoka (2015). "Python, performance, and natural language processing". In: *Proceedings of the 5th Workshop on Python for High-Performance and Scientific Computing - PyHPC '15*. ACM Press. DOI: 10.1145/2835857.2835858.

European Commission (2013). *xDT*. Accessed January 7, 2021. URL: https://ec.europa.eu/eip/ageing/standards/ict-and-communication/other-ict/xdt_en.

Fielding, Roy Thomas (2000). "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. University of California. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

Fowler, Martin (Oct. 1, 2011). "Foreword". In: *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. ADDISON WESLEY PUB CO INC. ISBN: 032154420X.

– (Feb. 2015). *Refactoring code that accesses external services*. Accessed February 9, 2021. URL: https://martinfowler.com/articles/refactoring-external-service.html.

Gamma, Erich et al. (Dec. 1, 1995). *Design Patterns*. Prentice Hall. ISBN: 0201633612.

Goodger, David and Guido van Rossum (May 2001). *PEP 257 – Docstring Conventions*. Accessed September 11, 2021. URL: https://www.python.org/dev/peps/pep-0257/.

Gruenheid, Anja (2016). "Data Integration with Dynamic Data Sources". PhD thesis. ETH Zurich. DOI: 10.3929/ETHZ-A-010861625.

Gruninger, Michael and Jintae Lee (Feb. 2002). "Ontology applications and design". In: *Communications of the ACM* 45.2, pp. 39–41. DOI: 10.1145/503124.503146.

Hansen, Mark D., Stuart E. Madnick, and Michael Siegel (2003). "Data Integration using Web Services". In: *SSRN Electronic Journal*. DOI: 10.2139/ssrn.376822.

Heidari, Farideh and Pericles Loucopoulos (Sept. 2014). "Quality evaluation framework (QEF): Modeling and evaluating quality of business processes". In: *International Journal of Accounting Information Systems* 15.3, pp. 193–223. DOI: 10.1016/j.accinf.2013.09.002.

Helms, Marilyn M. and Judy Nixon (Aug. 2010). "Exploring SWOT analysis – where are we now?" In: *Journal of Strategy and Management* 3.3, pp. 215–251. DOI: 10.1108/17554251011064837.

HL7 (Nov. 2019a). *CDA® (HL7 Clinical Document Architecture)*. Accessed January 7, 2021. URL: https://www.hl7.org/implement/standards/product_brief.cfm?product_id=496.

– (Nov. 2019b). *Introducing HL7 FHIR*. Accessed January 7, 2021. URL: https://hl7.org/FHIR/summary.html.

– (Nov. 2019c). *Medications Module*. Accessed January 7, 2021. URL: https://www.hl7.org/fhir/medications-module.html.

Hohpe, Gregor and Bobby Woolf (Jan. 1, 2004). *Enterprise Integration Patterns*. 1st Edition. Addison Wesley. 480 pp. ISBN: 0321200683.

Homayouni, Hajar, Sudipto Ghosh, and Indrakshi Ray (2018). "An Approach for Testing the Extract-Transform-Load Process in Data Warehouse Systems". In: *Proceedings of the 22nd International Database Engineering & Applications Symposium on - IDEAS 2018*. ACM Press. DOI: 10.1145/3216122.3216149.

Ives, Zachary George and Alon Halevy (2002). "Efficient Query Processing for Data Integration". PhD thesis. USA.

Jain, R. et al. (June 2008). "Exploring the Impact of Systems Architecture and Systems Requirements on Systems Integration Complexity". In: *IEEE Systems Journal* 2.2, pp. 209–223. DOI: 10.1109/jsyst.2008.924130.

Jalili, Monire and Kamran Rezaie (2010). "Quality principles deployment to achieve strategic results". In: *International Journal of Business Excellence* 3.2, p. 226. DOI: 10.1504/ijbex.2010.030730.

Kadadi, Anirudh et al. (Oct. 2014). "Challenges of data integration and interoperability in big data". In: *2014 IEEE International Conference on Big Data (Big Data)*. IEEE. DOI: 10.1109/bigdata.2014.7004486.

Kalin, Martin (Sept. 27, 2013). *Java Web Services: Up and Running*. O'Reilly Media, Inc, USA. 350 pp. ISBN: 1449365116.

Kannampallil, Thomas G. et al. (Dec. 2011). "Considering complexity in healthcare systems". In: *Journal of Biomedical Informatics* 44.6, pp. 943–947. DOI: 10.1016/j.jbi.2011.06.006.

Kassenärztliche Bundesvereinigung (Nov. 2020). *Datensatzbeschreibung KVDT*. URL: ftp://ftp.kbv.de/ita-update/Abrechnung/KBV_ITA_VGEX_Datensatzbeschreibung_KVDT.pdf.

Keller, W., C. Mitterbauer, and K. Wagner (1998). "Object-Oriented Data Integration Running Several Generations of Database Technology in Parallel". In.

Khan, Wajahat Ali et al. (2012). "Achieving interoperability among healthcare standards". In: *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication - ICUIMC '12*. ACM Press. DOI: 10.1145/2184751.2184868.

Klas, Wolfgang, Gisela Fischer, and Karl Aberer (Dec. 1994). "Integrating relational and object-oriented database systems using a metaclass concept". In: *Journal of Systems Integration* 4.4, pp. 341–372. DOI: 10.1007/bf01976279.

Koen, Peter A. et al. (Jan. 2004). "The PDMA ToolBook 1 for New Product Development". In: John Wiley & Sons. Chap. Fuzzy Front End: Effective Methods, Tools, and Techniques. ISBN: 9780471271086.

Kruchten, Philippe (1995). "The 4+1 View Model of Architecture". In: *IEEE Softw.* 12.6, pp. 42–50. DOI: 10.1109/52.469759. URL: https://doi.org/10.1109/52.469759.

Kuhn, Klaus A and Dario A Giuse (2001). "From hospital information systems to health information systems-problems, challenges, perspectives". In: *Yearbook of medical Informatics* 10.01, pp. 63–76.

Lapierre, Jozée (Apr. 2000). "Customer-perceived value in industrial contexts". In: *Journal of Business & Industrial Marketing* 15.2/3, pp. 122–145. DOI: 10.1108/08858620010316831.

Lavrakas, Paul (2008). "Research Hypothesis". In: *Encyclopedia of Survey Research Methods*. Sage Publications, Inc. DOI: 10.4135/9781412963947.n472.

Le Sueur, Helen et al. (June 2020). "The challenges in data integration – heterogeneity and complexity in clinical trials and patient registries of Systemic Lupus Erythematosus". In: *BMC Medical Research Methodology* 20.1, p. 164. ISSN: 1471-2288. DOI: 10.1186/s12874-020-01057-0.

Lehtosalo, Jukka (May 2019). *Our journey to type checking 4 million lines of Python*. Accessed August 25, 2021. URL: https://dropbox.tech/application/our-journey-to-type-checking-4-million-lines-of-python.

Lemburg, Marc-André (Mar. 2001). *PEP 249 – Python Database API Specification v2.0*. Accessed August 12, 2021. URL: https://www.python.org/dev/peps/pep-0249/.

Lenz, Richard, Mario Beyer, and Klaus A Kuhn (Feb. 2007). "Semantic integration in healthcare networks". In: *International Journal of Medical Informatics* 76.2-3, pp. 201–207. DOI: 10.1016/j.ijmedinf.2006.05.008.

Lenzerini, Maurizio (2002). "Data integration". In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems - PODS '02*. ACM Press. DOI: 10.1145/543613.543644.

Lindgreen, Adam and Finn Wynstra (Oct. 2005). "Value in business markets: What do we know? Where are we going?" In: *Industrial Marketing Management* 34.7, pp. 732–748. DOI: 10.1016/j.indmarman.2005.01.001.

Liu, Xiufeng (2012). "Data warehousing technologies for large-scale and right-time data". PhD thesis. Aalborg University.

Liu, Xiufeng, Christian Thomsen, and Torben Bach Pedersen (Aug. 2012). "MapReduce-based dimensional ETL made easy". In: *Proceedings of the VLDB Endowment* 5.12, pp. 1882–1885. DOI: 10.14778/2367502.2367528.

LOINC (n.d.). *About LOINC*. Accessed January 7, 2021. URL: https://loinc.org/about/.

Luigi Development Team (2020). *Luigi*. Accessed January 11, 2021. URL: https://github.com/spotify/luigi.

Lutz, Mark (2001). *Programming Python*. Beijing Sebastopol, CA: O'Reilly. ISBN: 9780596000851.

Maaskant, Jolanda M. et al. (Jan. 2018). "Medication audit and feedback by a clinical pharmacist decrease medication errors at the PICU: An interrupted time series analysis". In: *Health Science Reports* 1.3, e23. DOI: 10.1002/hsr2.23.

Mara Development Team (2020). *Mara Pipelines*. Accessed January 11, 2021. URL: https://github.com/mara/mara-pipelines.

MarketsandMarkets™ (Apr. 2019). *Healthcare IT Market*. Accessed December 9, 2020. URL: https://www.marketsandmarkets.com/Market-Reports/healthcare-it-252.html.

Meier, Remigius and Thomas R. Gross (Oct. 2019). "Reflections on the compatibility, performance, and scalability of parallel Python". In: *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*. ACM. DOI: 10.1145/3359619.3359747.

Microsoft (2009). *Microsoft® Application Architecture Guide*. Redmond, Washington: Microsoft. ISBN: 9780735627109.

Miles, Lawrence (1972). *Techniques of Value Analysis and Engineering*. New York: McGraw-Hill. ISBN: 9780070419261.

Mulligan, Gavin and Denis Gračanin (2009). "A Comparison of SOAP and REST Implementations of a Service Based Interaction Independence Middleware Framework". In: *Winter Simulation Conference*. WSC '09. Austin, Texas: Winter Simulation Conference, pp. 1423–1432. ISBN: 9781424457717.

Nyman, Jonny (2019). "Data Integration - Steps towards an efficient and user-friendly process". MA thesis. Åbo Akademi.

Ojwang', Antony (2018). "Auto-etl: a Generic Methodology for Transforming Entity-attribute-value (Eav) Data Model Into Flat Tables Using Form Metadata to Optimize Report Generation. A Case Study of Kenya EMR". MA thesis. University of Nairobi.

Oliveira Martins, Pedro Miguel de (2015). "Elastic ETL+Q For Any Data-Warehouse Using Time Bounds". PhD thesis. Universidade de Coimbra.

openEHR (2018). *What is openEHR?* Accessed January 7, 2021. URL: https://www.openehr.org/about/what_is_openehr.

Pandas Development Team (2020). *pandas: powerful Python data analysis toolkit*. Accessed January 9, 2021. URL: https://github.com/pandas-dev/pandas.

Parkin, Simon, David Ingham, and Graham Morgan (2007). "A Message Oriented Middleware Solution Enabling Non-repudiation Evidence Generation for Reliable Web Services". In: *Service Availability*. Springer Berlin Heidelberg, pp. 9–19. DOI: 10.1007/978-3-540-72736-1_2.

Petl Development Team (2019). *petl*. Accessed January 12, 2021. URL: https://petl.readthedocs.io/en/stable/index.html.

Priya, K et al. (2017). "Impact of Electronic Prescription Audit Process to Reduce Outpatient Medication Errors". In: *Indian Journal of Pharmaceutical Sciences* 79.6. DOI: 10.4172/pharmaceutical-sciences.1000321.

Rajlich, Václav (May 2014). "Software evolution and maintenance". In: *Future of Software Engineering Proceedings*. ACM. DOI: 10.1145/2593882.2593893.

Ramalingam, Ganesan and Kapil Vaswani (2013). "Fault tolerance via idempotence". In: *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '13*. ACM Press. DOI: 10.1145/2429069.2429100.

Riehle, Dirk (2000). "Framework design: a role modeling approach". en. PhD thesis. DOI: 10.3929/ETHZ-A-003867001.

Risch, Tore and Vanja Josifovski (2001). "Distributed data integration by object-oriented mediator servers". In: *Concurrency and Computation: Practice and Experience* 13.11, pp. 933–953. DOI: 10.1002/cpe.607.

Robinson, Ian (Oct. 1, 2011). "Foreword". In: *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. ADDISON WESLEY PUB CO INC. ISBN: 032154420X.

Rossum, Guido Van and Python Development Team (Dec. 2020). *Python tutorial*. Vol. 620. Centrum voor Wiskunde en Informatica Amsterdam.

Russom, Philip (Nov. 2007). "Complex Data: A New Challenge for Data Integration". In: *TDWI Monograph Series*.

– (May 2011). "Next Generation Data Integration". In: *TDWI Research*.

Saaty, R.W. (1987). "The analytic hierarchy process—what it is and how it is used". In: *Mathematical Modelling* 9.3, pp. 161–176. ISSN: 0270-0255. DOI: https://doi.org/10.1016/0270-0255(87)90473-8.

Salem, Rashed (2009). "Complex data integration into an active XML repository". In: *Proceedings of the International Conference on Management of Emergent Digital EcoSystems - MEDES '09*. ACM Press. DOI: 10.1145/1643823.1643916.

El-Sappagh, Shaker H. Ali, Abdeltawab M. Ahmed Hendawi, and Ali Hamed El Bastawissy (July 2011). "A proposed model for data warehouse ETL processes". In: *Journal of King Saud University - Computer and Information Sciences* 23.2, pp. 91–104. DOI: 10.1016/j.jksuci.2011.05.005.

Silva, Luis Miguel Monteiro (2016). "ETL na era do Big Data". MA thesis. Instituto Superior Técnico.

Simitsis, Alkis (2004). "Modeling and optimization of extraction-transformation-loading (ETL) processes in data warehouse environments". PhD thesis. University of Athens.

Simitsis, Alkis et al. (2009). "QoX-driven ETL design". In: *Proceedings of the 35th SIGMOD international conference on Management of data - SIGMOD '09*. ACM Press. DOI: 10.1145/1559845.1559954.

Simon, Herbert A. (1991). "The Architecture of Complexity". In: *Facets of Systems Science*. Springer US, pp. 457–476. DOI: 10.1007/978-1-4899-0718-9_31.

Smith, Barry (2003). "Ontology". In: *Blackwell Guide to the Philosophy of Computing and Information*. Ed. by Luciano Floridi. Oxford: Blackwell, pp. 155–166.

Snehalatha, Suma P. (2010). "Logical Modeling of ETL Processes Using XML". MA thesis. University of Cincinnati.

SNOMED International (n.d.). *Our organization*. Accessed January 7, 2021. URL: https://www.snomed.org/our-organization/our-organization.

Spark Development Team (2020). *Apache Spark*. Accessed January 11, 2021. URL: https://github.com/apache/spark.

SQLite (n.d.). *Appropriate Uses For SQLite*. Accessed February 5, 2021. URL: https://www.sqlite.org/whentouse.html.

Thomsen, Christian and Søren Kejser Jensen (2021). *pygrametl*. Accessed January 11, 2021. URL: https://chrthomsen.github.io/pygrametl/.

Thomsen, Christian and Torben Bach Pedersen (2009). "pygrametl". In: *Proceeding of the ACM twelfth international workshop on Data warehousing and OLAP - DOLAP '09*. ACM Press. DOI: 10.1145/1651291.1651301.

– (2011). "Easy and effective parallel programmable ETL". In: *Proceedings of the ACM 14th international workshop on Data Warehousing and OLAP - DOLAP '11*. ACM Press. DOI: 10.1145/2064676.2064684.

Thuy, Pham Thu Thu, Young-Koo Lee, and Sungyoung Lee (July 2012). "Semantic and structural similarities between XML Schemas for integration of ubiquitous healthcare data". In: *Personal and Ubiquitous Computing* 17.7, pp. 1331–1339. DOI: 10.1007/s00779-012-0567-5.

Triantos, K. (2016). "Human resources analytics at Viggo: warehousing solutions for CSV data". MA thesis. Eindhoven University of Technology.

Vassiliadis, Panos, Alkis Simitsis, and Spiros Skiadopoulos (2002). "Conceptual modeling for ETL processes". In: *Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP - DOLAP '02*. ACM Press. DOI: 10.1145/583890.583893.

Vernon, Vaughn (Feb. 2013). *Implementing Domain-Driven Design*. Pearson ITP.

Waldo, Jim et al. (1997). "A note on distributed computing". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, pp. 49–64. DOI: 10.1007/3-540-62852-5_6.

Weng, Gezhi, Upinder S. Bhalla, and Ravi Iyengar (1999). "Complexity in Biological Signaling Systems". In: *Science* 284.5411, pp. 92–96. ISSN: 0036-8075. DOI: 10.1126/science.284.5411.92.

WHO (n.d.[a]). *Anatomical Therapeutic Chemical (ATC) Classification*. Accessed January 7, 2021. URL: https://www.who.int/tools/atc-ddd-toolkit/atc-classification.

– (n.d.[b]). *International Classification of Primary Care, Second edition (ICPC-2)*. Accessed January 8, 2021. URL: https://www.who.int/classifications/icd/adaptations/icpc2/en/.

– (n.d.[c]). *International Nonproprietary Names Programme and Classification of Medical Products*. Accessed January 7, 2021. URL: https://www.who.int/teams/health-product-and-policy-standards/inn.

– (n.d.[d]). *International Statistical Classification of Diseases and Related Health Problems (ICD)*. Accessed January 7, 2021. URL: https://www.who.int/standards/classifications/classification-of-diseases.

Winter, Alfred et al. (July 2018). "Smart Medical Information Technology for Healthcare (SMITH)". In: *Methods of Information in Medicine* 57.S 01, e92–e105. DOI: 10.3414/me18-02-0004.

Xavier, Cristiano Rogério Lopes (2012). "Extração, Transformação e Carregamento Ágil de Dados". MA thesis. Universidade Portucalense.

Zhang, Hansi et al. (Dec. 2020). "An ontology-based documentation of data discovery and integration process in cancer outcomes research". In: *BMC Medical Informatics and Decision Making* 20.S4. DOI: 10.1186/s12911-020-01270-3.

Appendix A

Technologies Selection

In Section 2.4 the main technological tools and respective alternatives for the creation of a data integration platform have been listed and characterized. Here, AHP is applied to evaluate and pick which ones of these technologies are to be used.

The first selection that needs to be made is with regard to which data processing tool to use in ETLF. The second choice to be done is to identify which WMS tool to use, if any, with MIP to better manage integration processes. The main results of these AHP selections are detailed and explained in Sections A.1 and A.2. Evaluation criteria was selected considering the features provided by each framework (Section 2.4), the purpose of the given tools, and the requirements for the project (Section 3.3.2).

AHP

As already mentioned, AHP is the process used to make the best selection among the available options.

> *"[AHP] is a general theory of measurement. [...] It has found its widest applications in multicriteria decision making, planning and resource allocation and in conflict resolution [...]. In its general form the AHP is a nonlinear framework for carrying out both deductive and inductive thinking without use of the syllogism by taking several factors into consideration simultaneously and allowing for dependence and for feedback, and making numerical tradeoffs to arrive at a synthesis or conclusion"* (Saaty 1987).

Next follows a list of the adopted AHP steps to make the most suitable selection among all the available alternatives:

1. Create evaluation criteria based on the project requirements and desired features;
2. Compare criteria importance;
3. Calculate relative priority (also known as "priority vector" or "own vector");
4. Construct a comparison matrix for each criterion, considering each of the selected alternatives;
5. Get composite priority for alternatives;
6. Select best priority.

In the first step, to compare criteria importance, Saaty's fundamental scale (Figure A.1) is used.

Intensity of importance on an absolute scale	Definition	Explanation
1	Equal importance	Two activities contribute equally to the objective
3	Moderate importance of one over another	Experience and judgment strongly favor one activity over another
5	Essential or strong importance	Experience and judgment strongly favor one activity over another
7	Very strong importance	An activity is strongly favored and its dominance demonstrated in practice
9	Extreme importance	The evidence favoring one activity over another is of the highest possible order of affirmation
2,4,6,8	Intermediate values between the two adjacent judgments	When compromise is needed
Reciprocals	If activity i has one of the above numbers assigned to it when compared with activity j, then j has the reciprocal value when compared with i	
Rationals	Ratios arising from the scale	If consistency were to be forced by obtaining n numerical values to span the matrix

FIGURE A.1: The fundamental scale (Saaty 1987).

The consistency of all subjective matrices needs to be evaluated to know if the given priorities are random or not, and, therefore, valid or not. To do so, the *Consistency Ratio (CR)* has to be calculated with the following formula: $CR = CI/RI$. If the resulting value is bigger than 10% this means that it is *not consistent*. *Consistency Index (CI)* can be determined with the expression $CI = (\lambda_{max} - n)/(n-1)$, and the *Random Consistency Index (RI)* can be fetched from Figure A.2 that shows the RI values for square matrices of order n. λ_{max} can be calculated through the formula $Ax = \lambda_{max}x$ where A is the comparison matrix and x is the priority vector.

n	1	2	3	4	5	6	7	8	9	10
Random consistency index (R.I.)	0	0	0.58	0.90	1.12	1.24	1.32	1.41	1.45	1.49

FIGURE A.2: RI values for square matrices of order n (Saaty 1987).

In step number 5, the priorities' comparison matrix for each criterion considering all selected alternatives has to be multiplied by the priority vector x to determine the composite priority of all alternatives.

A.1 Data Processing Tool Selection

The selection of a data processing tool is quite important taking into account the purpose of the to-be-developed integration framework. Figure A.3 shows the options available and the criteria to evaluate it. The decision was between *Apache Spark*, *Pandas*, *Petl*, and *Python built-in Dictionaries*. Evaluation criteria includes complex transformations, data validation, simplicity, stability, and throughput. Data validation is regarded as the most important criterion, because data integrity is a critical requirement for the success of the current project. Next follows complex transformations, where developers should be able to use common transformation functions like merging, joining, grouping, etc. Added to that, this criterion also weights in the ability to efficiently transform data in any desirable way (e.g., row by bow, table by table). Stability is also quite important as data transformation needs to be a reliable process. It includes a general perception

of the quality and level of maintenance of the tool itself. Simplicity incorporates two sub-criteria: ease of use and added complexity to the overall solution. Throughput is not seen as a major drawback due to the fact that the number of entries needed to be parsed —approximately one hundred thousand— is not substantial and the machines where the process will run can easily deal with it.

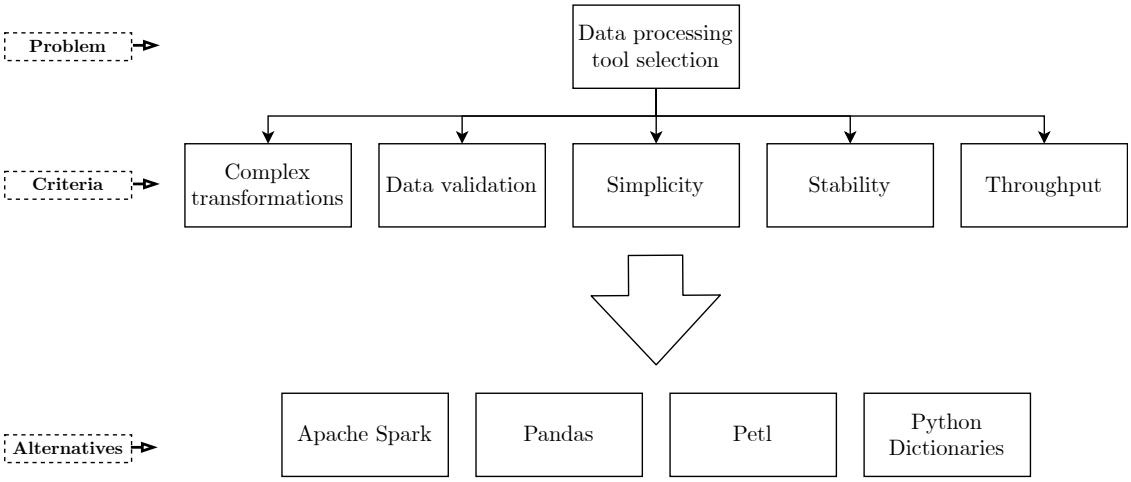

FIGURE A.3: AHP decision tree for the data processing tools.

Table A.1 shows a pair-to-pair importance comparison and the resulting relative priority of each defined criterion.

TABLE A.1: Criteria importance comparison for data processing tools.

	Complex transf.	**Data val.**	**Simplicity**	**Stability**	**Throughput**	**Relative priority (x)**
Complex transf.	1	1/3	4	3	7	0,2759
Data val.	3	1	4	3	7	0,4253
Simplicity	1/4	1/4	1	1/3	5	0,1001
Stability	1/3	1/3	3	1	5	0,1622
Throughput	1/7	1/7	1/5	1/5	1	0,0365

Now it's required to evaluate the consistency of the matrix. Equations (A.1), (A.2), and (A.3) demonstrate the process to acquire CR. The final calculation shows that the given comparison matrix has an inconsistency of 8,93%, being less than the 10% mark, and thus demonstrating that it is valid.

$$Ax = \lambda_{max}x \Leftrightarrow \begin{bmatrix} 1 & 1/3 & 4 & 3 & 7 \\ 3 & 1 & 4 & 3 & 7 \\ 1/4 & 1/4 & 1 & 1/3 & 5 \\ 1/3 & 1/3 & 3 & 1 & 5 \\ 1/7 & 1/7 & 1/5 & 1/5 & 1 \end{bmatrix} \begin{bmatrix} 0,2759 \\ 0,4253 \\ 0,1001 \\ 0,1622 \\ 0,0365 \end{bmatrix} = \lambda_{max} \begin{bmatrix} 0,2759 \\ 0,4253 \\ 0,1001 \\ 0,1622 \\ 0,0365 \end{bmatrix} \Leftrightarrow \quad \text{(A.1)}$$

$$\lambda_{max} = 5,4002$$

$$CI = \frac{\lambda_{max} - n}{n-1} \Leftrightarrow CI = \frac{5,4002 - 5}{5-1} \Leftrightarrow CI = 0,1001 \tag{A.2}$$

$$CR = \frac{CI}{RI} \Leftrightarrow CR = \frac{0,1001}{1,12} \Leftrightarrow CR = 0,0893 \tag{A.3}$$

The next step is to compare all alternatives regarding each criterion (Tables A.2, A.3, A.4, A.5, and A.6).

Table A.2: Data processing tools comparison regarding complex transformations.

	Spark	**Pandas**	**Petl**	**Dict.**	**Relative weight**
Spark	1	1/5	1/3	1/2	0,0840
Pandas	5	1	3	3	0,5083
Petl	3	1/3	1	3	0,2664
Dict.	2	1/3	1/3	1	0,1413

Equations (A.4), (A.5), and (A.6) validate that the comparison of the tools in Table A.2 is consistent with a CR of 4,91%.

$$Ax = \lambda_{max}x \Leftrightarrow \begin{bmatrix} 1 & 1/5 & 1/3 & 1/2 \\ 5 & 1 & 3 & 3 \\ 3 & 1/3 & 1 & 3 \\ 2 & 1/3 & 1/3 & 1 \end{bmatrix} \begin{bmatrix} 0,0840 \\ 0,5083 \\ 0,2664 \\ 0,1413 \end{bmatrix} = \lambda_{max} \begin{bmatrix} 0,0840 \\ 0,5083 \\ 0,2664 \\ 0,1413 \end{bmatrix} \Leftrightarrow \tag{A.4}$$

$$\lambda_{max} = 4,1326$$

$$CI = \frac{\lambda_{max} - n}{n-1} \Leftrightarrow CI = \frac{4,1326 - 4}{4-1} \Leftrightarrow CI = 0,0442 \tag{A.5}$$

$$CR = \frac{CI}{RI} \Leftrightarrow CR = \frac{0,0442}{0,90} \Leftrightarrow CR = 0,0491 \tag{A.6}$$

Table A.3: Data processing tools comparison regarding data validation.

	Spark	**Pandas**	**Petl**	**Dict.**	**Relative weight**
Spark	1	1/7	1/9	1/9	0,0373
Pandas	7	1	1/3	1/3	0,1705
Petl	9	3	1	1	0,3961
Dict.	9	3	1	1	0,3961

Equations (A.7), (A.8), and (A.9) validate that the comparison of the tools in Table A.3 is consistent with a CR of 3,43%.

$$Ax = \lambda_{max}x \;\Leftrightarrow\; \begin{bmatrix} 1 & 1/7 & 1/9 & 1/9 \\ 7 & 1 & 1/3 & 1/3 \\ 9 & 3 & 1 & 1 \\ 9 & 3 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0,0373 \\ 0,1705 \\ 0,3961 \\ 0,3961 \end{bmatrix} = \lambda_{max} \begin{bmatrix} 0,0373 \\ 0,1705 \\ 0,3961 \\ 0,3961 \end{bmatrix} \;\Leftrightarrow \tag{A.7}$$

$$\lambda_{max} = 4,0927$$

$$CI = \frac{\lambda_{max} - n}{n-1} \;\Leftrightarrow\; CI = \frac{4,0927 - 4}{4-1} \;\Leftrightarrow\; CI = 0,0309 \tag{A.8}$$

$$CR = \frac{CI}{RI} \;\Leftrightarrow\; CR = \frac{0,0309}{0,90} \;\Leftrightarrow\; CR = 0,0343 \tag{A.9}$$

Table A.4: Data processing tools comparison regarding simplicity.

	Spark	**Pandas**	**Petl**	**Dict.**	**Relative weight**
Spark	1	1/9	1/9	1/9	0,0355
Pandas	9	1	1/2	1/3	0,1987
Petl	9	2	1	1/2	0,2957
Dict.	9	3	2	1	0,4701

Equations (A.10), (A.11), and (A.12) validate that the comparison of the tools in Table A.4 is consistent with a CR of 5,41%.

$$Ax = \lambda_{max}x \;\Leftrightarrow\; \begin{bmatrix} 1 & 1/9 & 1/9 & 1/9 \\ 9 & 1 & 1/2 & 1/3 \\ 9 & 2 & 1 & 1/2 \\ 9 & 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} 0,0355 \\ 0,1987 \\ 0,2957 \\ 0,4701 \end{bmatrix} = \lambda_{max} \begin{bmatrix} 0,0355 \\ 0,1987 \\ 0,2957 \\ 0,4701 \end{bmatrix} \;\Leftrightarrow \tag{A.10}$$

$$\lambda_{max} = 4,1460$$

$$CI = \frac{\lambda_{max} - n}{n-1} \;\Leftrightarrow\; CI = \frac{4,1460 - 4}{4-1} \;\Leftrightarrow\; CI = 0,0487 \tag{A.11}$$

$$CR = \frac{CI}{RI} \;\Leftrightarrow\; CR = \frac{0,0487}{0,90} \;\Leftrightarrow\; CR = 0,0541 \tag{A.12}$$

TABLE A.5: Data processing tools comparison regarding stability.

	Spark	**Pandas**	**Petl**	**Dict.**	**Relative weight**
Spark	1	1	5	1/3	0,2117
Pandas	1	1	5	1/3	0,2117
Petl	1/5	1/5	1	1/7	0,0529
Dict.	3	3	7	1	0,5238

Equations (A.13), (A.14), and (A.15) validate that the comparison of the tools in Table A.5 is consistent with a CR of 2,74%.

$$Ax = \lambda_{max}x \;\Leftrightarrow\; \begin{bmatrix} 1 & 1 & 5 & 1/3 \\ 1 & 1 & 5 & 1/3 \\ 1/5 & 1/5 & 1 & 1/7 \\ 3 & 3 & 7 & 1 \end{bmatrix} \begin{bmatrix} 0,2117 \\ 0,2117 \\ 0,0529 \\ 0,5238 \end{bmatrix} = \lambda_{max} \begin{bmatrix} 0,2117 \\ 0,2117 \\ 0,0529 \\ 0,5238 \end{bmatrix} \;\Leftrightarrow \tag{A.13}$$

$\lambda_{max} = 4,0738$

$$CI = \frac{\lambda_{max} - n}{n-1} \;\Leftrightarrow\; CI = \frac{4,0738 - 4}{4-1} \;\Leftrightarrow\; CI = 0,0246 \tag{A.14}$$

$$CR = \frac{CI}{RI} \;\Leftrightarrow\; CR = \frac{0,0246}{0,90} \;\Leftrightarrow\; CR = 0,0274 \tag{A.15}$$

TABLE A.6: Data processing tools comparison regarding throughput.

	Spark	**Pandas**	**Petl**	**Dict.**	**Relative weight**
Spark	1	9	5	3	0,5577
Pandas	1/9	1	1/5	1/7	0,0417
Petl	1/5	5	1	1/3	0,1330
Dict.	1/3	7	3	1	0,2676

Equations (A.16), (A.17), and (A.18) validate that the comparison of the tools in Table A.6 is consistent with a CR of 6,47%.

$$Ax = \lambda_{max}x \;\Leftrightarrow\; \begin{bmatrix} 1 & 9 & 5 & 3 \\ 1/9 & 1 & 1/5 & 1/7 \\ 1/5 & 5 & 1 & 1/3 \\ 1/3 & 7 & 3 & 1 \end{bmatrix} \begin{bmatrix} 0,5577 \\ 0,0417 \\ 0,1330 \\ 0,2676 \end{bmatrix} = \lambda_{max} \begin{bmatrix} 0,5577 \\ 0,0417 \\ 0,1330 \\ 0,2676 \end{bmatrix} \;\Leftrightarrow \tag{A.16}$$

$\lambda_{max} = 4,1747$

$$CI = \frac{\lambda_{max} - n}{n-1} \Leftrightarrow CI = \frac{4,1747 - 4}{4-1} \Leftrightarrow CI = 0,0582 \tag{A.17}$$

$$CR = \frac{CI}{RI} \Leftrightarrow CR = \frac{0,0582}{0,90} \Leftrightarrow CR = 0,0647 \tag{A.18}$$

Now, the final step is to determine the composite priority of the alternatives (equation (A.19)). Considering the defined criteria and their respective importance, *Python Dictionaries* appears as the most suitable option to use as a data processing tool.

$$\begin{bmatrix} 0,0840 & 0,0373 & 0,0355 & 0,2117 & 0,5577 \\ 0,5083 & 0,1705 & 0,1987 & 0,2117 & 0,0417 \\ 0,2664 & 0,3961 & 0,2957 & 0,0529 & 0,1330 \\ 0,1413 & 0,3961 & 0,4701 & 0,5238 & 0,2676 \end{bmatrix} \times \begin{bmatrix} 0,2759 \\ 0,4253 \\ 0,1001 \\ 0,1622 \\ 0,0365 \end{bmatrix} = \begin{bmatrix} 0,0973 \\ 0,2685 \\ 0,2850 \\ 0,3492 \end{bmatrix} \tag{A.19}$$

A.2 Workflow Management Tool Selection

The decision here is between *Apache Airflow*, *Luigi*, and *none* of them, as Figure A.4 shows. Both Airflow and Luigi offer quite similar features but their purposes are slightly different. While Airflow is a workflow scheduler, Luigi is used to manage data pipelines. Using none of them is the other option as using these frameworks might increase MIP's complexity unnecessarily. With that in mind, the selected criteria to evaluate them are logging and scheduling capabilities, and the simplicity they provide to the developer. Simplicity is, by far, the most important criterion. As a platform used to build medication integration processes, MIP must be easy to use and, therefore, must not deal with structures/tools that complicate it too much. This enables for an increase in the developers' performance, and easier software maintenance for the years to come. Although the scheduling is important for the requirements of the project, Python scripting can very easily achieve this task without requiring external tools.

Table A.7 shows a pair-to-pair importance comparison and the resulting relative priority of each criteria.

Table A.7: Criteria importance comparison for WMS frameworks.

	Logging	**Scheduling**	**Simplicity**	**Relative priority (x)**
Logging	1	3	1/5	0,1932
Scheduling	1/3	1	1/7	0,0833
Simplicity	5	7	1	0,7235

Now it's required to evaluate the consistency of the matrix. Equations (A.20), (A.21), and (A.22) demonstrate the process to acquire CR. The final calculation shows that the given comparison matrix has an inconsistency of 5,67%, being less than the 10% mark, and, thus, demonstrating that it is valid.

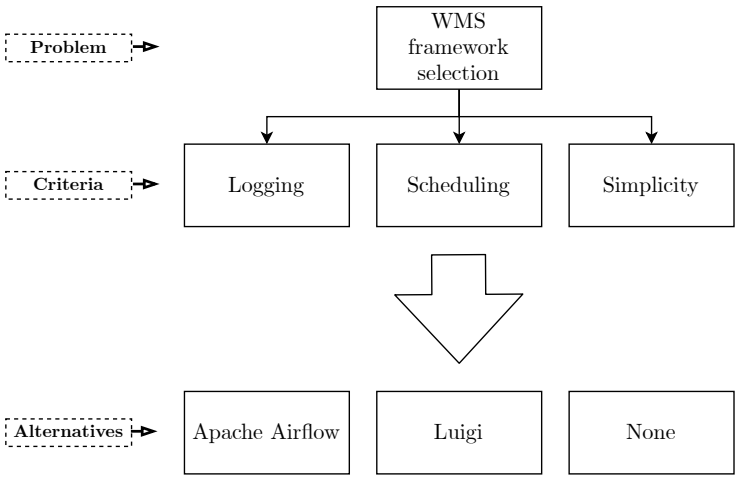

FIGURE A.4: AHP decision tree for WMS tools.

$$Ax = \lambda_{max}x \;\Leftrightarrow\; \begin{bmatrix} 1 & 3 & 1/5 \\ 1/3 & 1 & 1/7 \\ 5 & 7 & 1 \end{bmatrix} \begin{bmatrix} 0,1932 \\ 0,0833 \\ 0,7235 \end{bmatrix} = \lambda_{max} \begin{bmatrix} 0,1932 \\ 0,0833 \\ 0,7235 \end{bmatrix} \;\Leftrightarrow \tag{A.20}$$

$$\lambda_{max} = 3,0658$$

$$CI = \frac{\lambda_{max} - n}{n-1} \;\Leftrightarrow\; CI = \frac{3,0658 - 3}{3-1} \;\Leftrightarrow\; CI = 0,0329 \tag{A.21}$$

$$CR = \frac{CI}{RI} \;\Leftrightarrow\; CR = \frac{0,0329}{0,58} \;\Leftrightarrow\; CR = 0,0567 \tag{A.22}$$

The next step is to compare all alternatives regarding each criterion (Tables A.8, A.9, and A.10).

TABLE A.8: WMS frameworks comparison regarding logging.

	Airflow	**Luigi**	**None**	**Relative weight**
Airflow	1	5	7	0,7235
Luigi	1/5	1	3	0,1932
None	1/7	1/3	1	0,0833

Equations (A.23), (A.24), and (A.25) validate that the comparison of the frameworks in Table A.8 is consistent with a CR of 5,67%.

$$Ax = \lambda_{max}x \Leftrightarrow \begin{bmatrix} 1 & 5 & 7 \\ 1/5 & 1 & 3 \\ 1/7 & 1/3 & 1 \end{bmatrix} \begin{bmatrix} 0,7235 \\ 0,1932 \\ 0,0833 \end{bmatrix} = \lambda_{max} \begin{bmatrix} 0,7235 \\ 0,1932 \\ 0,0833 \end{bmatrix} \Leftrightarrow \tag{A.23}$$

$\lambda_{max} = 3,0658$

$$CI = \frac{\lambda_{max} - n}{n-1} \Leftrightarrow CI = \frac{3,0658 - 3}{3-1} \Leftrightarrow CI = 0,0329 \tag{A.24}$$

$$CR = \frac{CI}{RI} \Leftrightarrow CR = \frac{0,0329}{0,58} \Leftrightarrow CR = 0,0567 \tag{A.25}$$

Table A.9: WMS frameworks comparison regarding scheduling.

	Airflow	**Luigi**	**None**	**Relative weight**
Airflow	1	9	3	0,6689
Luigi	1/9	1	1/5	0,0637
None	1/3	5	1	0,2674

Equations (A.26), (A.27), and (A.28) validate that the comparison of the frameworks in Table A.9 is consistent with a CR of 2,52%.

$$Ax = \lambda_{max}x \Leftrightarrow \begin{bmatrix} 1 & 9 & 3 \\ 1/9 & 1 & 1/5 \\ 1/3 & 5 & 1 \end{bmatrix} \begin{bmatrix} 0,6689 \\ 0,0637 \\ 0,2674 \end{bmatrix} = \lambda_{max} \begin{bmatrix} 0,6689 \\ 0,0637 \\ 0,2674 \end{bmatrix} \Leftrightarrow \tag{A.26}$$

$\lambda_{max} = 3,0292$

$$CI = \frac{\lambda_{max} - n}{n-1} \Leftrightarrow CI = \frac{3,0292 - 3}{3-1} \Leftrightarrow CI = 0,0146 \tag{A.27}$$

$$CR = \frac{CI}{RI} \Leftrightarrow CR = \frac{0,0146}{0,58} \Leftrightarrow CR = 0,0252 \tag{A.28}$$

Equations (A.29), (A.30), and (A.31) validate that the comparison of the frameworks in Table A.10 is consistent with a CR of 5,67%.

TABLE A.10: WMS frameworks comparison regarding simplicity.

	Airflow	**Luigi**	**None**	**Relative weight**
Airflow	1	1/3	1/7	0,0833
Luigi	3	1	1/5	0,1932
None	7	5	1	0,7235

$$Ax = \lambda_{max}x \;\Leftrightarrow\; \begin{bmatrix} 1 & 1/3 & 1/7 \\ 3 & 1 & 1/5 \\ 7 & 5 & 1 \end{bmatrix} \begin{bmatrix} 0,0833 \\ 0,1932 \\ 0,7235 \end{bmatrix} = \lambda_{max} \begin{bmatrix} 0,0833 \\ 0,1932 \\ 0,7235 \end{bmatrix} \;\Leftrightarrow \tag{A.29}$$

$$\lambda_{max} = 3,0658$$

$$CI = \frac{\lambda_{max} - n}{n-1} \;\Leftrightarrow\; CI = \frac{3,0658 - 3}{3-1} \;\Leftrightarrow\; CI = 0,0329 \tag{A.30}$$

$$CR = \frac{CI}{RI} \;\Leftrightarrow\; CR = \frac{0,0329}{0,58} \;\Leftrightarrow\; CR = 0,0567 \tag{A.31}$$

Now, the final step is to determine the composite priority of the alternatives (equation (A.32)). Considering the defined criteria and their respective importance, *None* appears as the most suitable option.

$$\begin{bmatrix} 0,7235 & 0,6689 & 0,0833 \\ 0,1932 & 0,0637 & 0,1932 \\ 0,0833 & 0,2674 & 0,7235 \end{bmatrix} \times \begin{bmatrix} 0,1932 \\ 0,0833 \\ 0,7235 \end{bmatrix} = \begin{bmatrix} 0,2558 \\ 0,1824 \\ 0,5618 \end{bmatrix} \tag{A.32}$$

Appendix B

Evaluation Criteria

In this Appendix, QEF evaluation criteria for MIP and ETL processes built using MIP are described to be evaluated later.

B.1 Medication Integration Platform

Table B.1 describes how each QEF requirement of MIP is to be evaluated. Some assessment criteria are subjective in nature, and the only ones that don't require any sort of manual interaction are A1 and SEM4. SEM2 and SEM3 are somewhat dependent on personal preference, which might affect their final evaluation outcome. Nonetheless, if the solution is satisfactory, the results will tend to be positive.

B.2 Integration Process

Table B.2 describes how all QEF requirements for ETL processes built using MIP are to be evaluated. Considering the provided assessment criteria, it can be stated that this evaluation is objective in nature as none of its elements takes into account any personal preference. All the metrics described here can be assessed through unit and integration software testing. DI1, DI3, DI4, and DI5 are the most complicated to test because they require extensive domain knowledge. P1 and P2 will require tests to be performed in a production-like environment where the machine has identical processor speed and memory. Technologies that allow virtualization at the operating system level, such as Docker, might come in handy here.

TABLE B.1: Metric evaluation of QEF requirements for MIP.

		wf_k %				
Req.	**Metric evaluation**	**0**	**33**	**67**	**100**	**Formula**
ETL1	The platform provides a way for different data structures and communication APIs to be used.	No	—	(*)	Yes	—
ETL2	The platform contains domain specific rules for medication transformation.	No	1 of t_m	2 of t_m	All of t_m	—
ETL3	The platform contains modules for data persistence in SQLite databases.	No	—	—	Yes	—
O1	Developers are able to either schedule or execute integration processes on demand.	No	—	—	Yes	—
O2	Provides an approach to automatically send a notification whenever a critical failure occurs.	No	—	—	Yes	—
A1	End-to-end tests are successful in both Unix and Windows systems, while using the same memory and processor speed.	No	—	—	Yes	—
A2	Processes make use of N configurations.	—	—	—	—	n/N
SEM1	New ServiceConnectors can be added without modifying any other component.	No	—	—	Yes	—
SEM2	Components can be easily extended with custom behaviors without requiring modification of existent behaviors.	No	—	—	Yes	—
SEM3	MIP's approach follows a modular design and a clear separation of concerns, allowing component modification without adjusting other unrelated ones.	No	—	—	Yes	—
SEM4	Processes do not need any manual interaction to be executed from start to finish.	No	—	—	Yes	—
EU1	SEM1, SEM2, and SEM3 are completely fulfilled.	No	—	—	Yes	
EU2	The connector component is isolated from the other components.	No	—	—	Yes	—
R1	The ALERT® medication model is present and can be reused in other processes.	No	—	—	Yes	—

Legend:
wf_k – requirement fulfillment k
(*) – Yes, but only relevant data is not filtered
t_m – medication transformation functions = transform + enrich + validate
n – actual value

TABLE B.2: Metric evaluation of QEF requirements for medication ETL processes.

		wf_k %		
Req.	**Metric evaluation**	**0**	**50**	**100**
F1	Web services are correctly accessed.	No	—	Yes
F2	Data is properly filtered, enriched, and transformed according to the domain rules.	No	—	Yes
F3	Data is correctly persisted in the database.	No	—	Yes
O1	The process is executed right on schedule.	No	—	Yes
O2	A notification is sent whenever a critical failure occurs.	No	—	Yes
DI1	All data in the database is complete according to domain rules (i.e., Entity sizes are equal in internal and external entities, all values are present for required attributes).	No	—	Yes
DI2	There are no duplicated entries in the database.	No	—	Yes
DI3	All data in the database must follow the domain rules.	No	—	Yes
DI4	Data directly extracted from external systems must be the same on the client.	No	—	Yes
DI5	All invalid data in the database is marked with an invalid state.	No	—	Yes
R1	Network failures are properly handled.	No	—	Yes
R2	Server errors are properly handled.	No	—	Yes
R3	Timeouts are properly handled.	No	—	Yes
R4	There are no runtime exceptions.	No	—	Yes
R5	The process is capable of executing the next scheduled run after a failure occurs.	No	—	Yes
S1	Only authorized personnel has access to the database.	No	—	Yes
L1	Errors are properly added to the logs.	No	—	Yes
L2	Information regarding each executed process is properly added to the logs.	No	—	Yes
P1	Ideally, the process should take less than 30 minutes. More than 1 hour is too much.	$t > 1h$	$t > 30m$ $t <= 1h$	$t <= 30m$
P2	The process uses no more than 1 GB of memory.	No	—	Yes

Legend:
wf_k – *requirement fulfillment k*
t – *time*