CTU

CZECH TECHNICAL UNIVERSITY IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Science

Master's Thesis

Data flow structuralization and visualization in the scope of enterprise integration platform

Bc. Marek Mičkal
Open Informatics, Software Engineering

May 2024
Supervisor: Ing. Karel Frajták, Ph.D.

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Mičkal Marek**	Personal ID number:	**492276**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Software Engineering**		

II. Master’s thesis details

Master’s thesis title in English:

Data flow structuralization and visualization in the scope of enterprise integration platform

Master’s thesis title in Czech:

Strukturalizace a vizualizace datových toků v rámci firemní integrační platformy

Guidelines:

Data integration represents one of the essential components of corporate software solutions infrastructure. Given the challenge of heterogeneous data structures present in various components of such systems, the clarity and traceability of data flow and individual data entities are crucial aspects of these solutions.
The thesis will be developed in collaboration with ON Semiconductor Czech Republic, s.r.o., which, within its Middleware team, provides a company-wide integration platform with approximately 4000 active integration flows, representing approximately 5 TB of daily network traffic.
Familiarize with the technologies underlying the existing integration platform (Spring framework, Apache Camel, Hazelcast, etc.) and the context of data integration in corporate systems. Perform an analysis of data flows within the platform, propose a methodology for transforming data flow definitions into structured formats based on this analysis and research, and apply this methodology appropriately to develop an experimental software solution. As part of designing this methodology, define and conduct an analysis of suitable methods and techniques. The implementation must meet conditions, especially regarding universal usability within the mentioned integration platform and the comprehensibility aspect of the resulting display for users. Test these properties of the resulting implementation on an independent sample of users using a carefully chosen methodology.
Based on the testing, determine the benefits of the implementation.
The implementation part of the master's thesis, within its scope, should utilize appropriate automation tools to map and visualize at least a substantial portion of all integration flows. In line with the requirement for universality, it is necessary to consider aspects of individual groups and subgroups of integrations, including relevant technologies, their specifics, used data formats, protocols, and methods for capturing data flow structure.

Bibliography / sources:

Ibsen, Claus, and Jonathan Anstey. Camel in Action. Simon and Schuster, 2018.
Dong, Xin Luna, and Divesh Srivastava. "Big Data Integration." 2013 IEEE 29th International Conference on Data Engineering (ICDE). IEEE, 2013.
Camposo, Guilherme. Cloud Native Integration with Apache Camel. Apress, 2021.
Popa, Sorin. "Secure Applications Integration with Apache Camel." Journal of Mobile, Embedded and Distributed Systems 7.1 (2015): 24-29
Doan, AnHai, Alon Halevy, and Zachary Ives. Principles of Data Integration. Elsevier Science, 2012.

Name and workplace of master's thesis supervisor:

Ing. Karel Frajták, Ph.D. System Testing IntelLigent Lab FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **01.02.2024** Deadline for master's thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

Ing. Karel Frajták, Ph.D.	Head of department's signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature		Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgement

I would like to express my gratitude to my supervisor, Ing. Karel Frajták, Ph.D., and the external examiner, Ing. Pavel Kostelník, Ph.D., for their professional guidance and support throughout the development of this thesis.

I am also thankful to ON Semiconductor for allowing me to pursue my research at their institution. I am particularly grateful to the entire middleware team for their collaboration and generously sharing their expertise with me, which has been instrumental in my growth and learning.

Lastly, I would like to thank my family and friends for their unwavering support and encouragement throughout my studies.

Declaration

I declare that I have completed the submitted work independently and have listed all the information sources used per the Methodological Guidelines on adhering to ethical principles in the preparation of university final theses.

In Prague 15. 5. 2024

....

Abstrakt /

Diplomová práce se zabývá tím, jak vyvinout strukturalizaci pro datové integrace v rámci specifické integrační platformy a jak na základě této strukturalizace následně vytvořit vhodný model vizualizace, který bude použitelný pro širokou škálu datových integrací implementovaných v rámci platformy. Tato platforma, součást společnosti ON Semiconductor, představuje distribuovanou middleware platformu, zahrnující více než 4000 integrací.

Práce začíná stručným úvodem a vymezením svých cílů, následovaný podrobnou kapitolou metodologie, která popisuje metody a postupy použity během vývoje práce. Následně se práce zabývá diskusí o problémech souvisejících s datovými integracemi, čímž poskytuje základní znalosti nezbytné pro komplexní porozumění dané problematice.

Další kapitola, věnující se analýze problematiky úzce spjaté se zadáním, obsahuje podrobný průzkum, zaměřující se na popis samotné platformy, seznámení s integračním frameworkem Apache Camel a průzkum existujících řešení. Následuje kapitola o implementaci, která poskytuje podrobný popis procesu vývoje, včetně popisu vzniklých komplikací a nalezených řešení.

Poté práce přináší kapitolu o dosažených výsledcích společně s popisem vzniklé vizualizace, její vlastnosti a možnosti, které nabízí. Tato kapitola také shrnuje a hodnotí tyto výsledky především na základě uživatelského testování.

Závěrečná část práce přináší reflektivní závěr práce jako celku, shrnující hlavní zjištění a navrhující možnosti pro budoucí výzkum a zdokonalení existujícího řešení.

Klíčová slova: integrace dat, datové toky, vizualizace datových toků, vizualizace datových integrací, strukturalizace datových integrací, distribuovaný middleware, Apache Camel

Překlad titulu: Strukturalizace a vizualizace datových toků v rámci firemní integrační platformy

Abstract /

This thesis attempts to develop a structuralization of data integrations within a custom integration platform, subsequently developing a suitable visualization model to encompass a wide array of data integrations implemented within the platform. This platform, a component of ON Semiconductor, represents a distributed middleware enterprise platform encompassing over 4000 integrations.

The thesis begins with a brief introduction and outline of its objectives, followed by a detailed methodology chapter delineating the approach employed throughout the thesis development. Subsequently, the thesis delves into a discussion on the issues relevant to data integration, furnishing fundamental knowledge crucial for a comprehensive understanding of the problematics.

Next, the Problem Analysis chapter presents a comprehensive exploration, focusing on describing the platform, outlining the Apache Camel framework, and surveying existing solutions. The Flowwing Implementation chapter provides an in-depth report of the development process, including challenges encountered and solutions devised. It concludes the development part with a chapter detailing the results and presenting the implemented visualizations, their features, and insights from user testing.

The thesis ends with a reflective conclusion, summarizing the essential findings and proposing avenues for future research and enhancement.

Keywords: data integration, data flow, data flow visualization, data integration visualization, data integration structuralization, distributed middleware, Apache Camel

Contents

Tables / Figures

Chapter 1 Introduction

1.1 Motivation

In the daily fabric of professional industries, data integration harmonizes disparate information sources, rendering each data point valuable and essential contributor to the cohesive functioning of diverse work environments, irrespective of industry distinctions. That is why data integration is a crucial part of various industries worldwide.

Making such integrations functional and practical can be a complex process. The data flow begins with obtaining the data from premises, edge devices, the cloud, or streaming data sources. Then, the data must be transformed to its usable form since the format of stored data differs from the intended format of the delivered data. Before the delivery, the whole process must be kept secure and resistant to failures.

Other essential principles of data integration, such as flexibility and scalability, must be achieved to implement a well-functionally data integration framework. Solutions like SOA - Service-oriented architecture, ESB - Enterprise service bus, microservices architecture, middleware, event streaming, enterprise messaging, and more can realize these principles.

Specific platforms such as *Informatica*[1] or *Talend*[2] embody these concepts to offer comprehensive data integration solutions. However, it is worth noting that many companies develop custom integration platforms tailored precisely to their unique needs, allowing them to align their integration strategies closely with their business requirements.

The insights gained from data integrations must be accessible to a broader audience, including non-technical stakeholders. Visualization can help with that. Moreover, visualization helps qualified experts use specific integration platforms smoothly and effectively.

Topics mentioned above will be further discussed in subsequent chapters, followed by a chapter focused on the implementation.

1.2 Goal

The primary objective of this thesis is to develop a structured framework for representing various data integration within corporate software infrastructures. This framework addresses the critical challenge of maintaining clarity amid heterogeneous data environments by providing a standardized approach to visualizing integration flows. By analyzing existing data integration structures, the thesis seeks to create a visual interface that enhances comprehensibility and facilitates efficient navigation within the integration platform.

[1] https://www.informatica.com
[2] https://www.talend.com

Collaborating with *ON Semiconductor Czech Republic, s.r.o.*, this study aims to delve into the technological backbone of the company's integration platform, which boasts around 4000 active integration flows and handles approximately 5 TB of daily network traffic.

This thesis will thoroughly analyze data flows within the existing platform, which is underpinned by technologies such as the *Spring* framework, *Apache Camel*, and *Hazelcast*. Building upon this analysis and supplementary research, the thesis aims to create a structured and standardized framework for representing existing data integrations within the company's platform. This methodological blueprint will not only guide the development of an experimental software solution but will also inform the analytical process to identify and employ the most apt methods and techniques for such a transformation.

The implementation phase of this master's project is assessed with a task to harness the capabilities of automation tools to accurately map and exhibit a significant segment, if not all, of the integration flows. The universality of the software solution is paramount. It must cater to the diverse landscape of integration groups and subgroups, accounting for the nuances of the technologies employed, data formats utilized, and the various protocols and techniques in play.

The envisaged extension to the existing integration platform is set to augment user interaction, providing a visual interface for data flows that enhances comprehensibility and, consequently, could alleviate the workload on the platform's integration team. As the platform expands with new data integrations, this tool is anticipated to strengthen its efficiency and performance.

In its closing, the thesis will evaluate the benefits of the implementation through testing with an independent user sample, applying a meticulously chosen methodology. The findings from this evaluation will present the practical benefits and the impact of the visual and structural enhancements on users' interaction with the data integration platform.

Chapter 2
Methodology

In this chapter, the thesis delves into the methodologies and strategies meticulously selected and applied throughout this thesis. The primary aim is to explain the creation of a visualization representing the data flow within a custom integration platform.

To achieve this, the thesis begins by addressing the overarching challenges associated with data integration. Subsequently, the thesis meticulously details the journey from the initial problem analysis through the stages of implementation and culminates in the presentation of our final results.

This chapter aims to provide a comprehensive overview of the methodological underpinnings that have guided the thesis in achieving its goals, offering readers a clear insight into the depth of the chosen approach.

2.1 Thesis Structure

The thesis follows a structured approach designed to address the complexities of data integration visualization systematically. It begins with an introductory chapter 1 that briefly outlines the problematics and articulates the overarching goal of the thesis. Followed by this methodological chapter.

The next chapter, Data Integration 3, aims to lay the foundational theories essential for understanding the complexities of data integration. While this theoretical chapter offers essential insights, the thesis remains accessible even without a deep dive into this section.

The subsequent portion of the thesis is dedicated to the comprehensive work taken to fulfill the thesis goal. Divided into three chapters, this segment logically explains the process behind crafting the solution. The first of these chapters, 4 Problem Analysis, initiates the discourse by conducting an in-depth analysis of the identified problematics and proposing fundamental ideas for addressing it. Following this, the Implementation chapter 5 delineates the steps taken to translate these conceptualizations into a tangible product. Finally, the Work Results and Evaluation chapter 6 provides a comprehensive overview of the developed visualization's functionalities, aesthetics, and results evaluation. This chapter also underscores the contributions of the visualization, thus serving as a culmination of the procedural segment of the thesis.

The thesis concludes with a dedicated chapter 7 synthesizing the essential findings and discussing potential avenues for future research.

Beyond the main body of the thesis, formal requirements such as literature citations, the thesis assignment, glossary, and additional screenshots of the final visualization are presented in the References and Appendix sections, respectively.

This meticulous structuring should ensure clarity and coherence throughout the thesis, facilitating a comprehensive understanding of the undertaken research.

2.2 Problem Analysis Research

Two main components were crucial in the research phase outlined in the problem analysis. Firstly, to investigate the custom integration platform, consultation with experts, examination of *Confluence* pages, and code reading were conducted. Secondly, research on the *Apache Camel* framework was undertaken, primarily relying on the second edition of the book *Camel in Action* - [1].

2.3 Implementation Methodology

An iterative approach was adopted for the implementation methodology, progressing step by step, with tasks tracked using the *Jira* system. Throughout the development process, consultations with other developers were conducted to ensure the delivery of a suitable product and high-quality code. Knowledge gained during studies was utilized, from developing the architecture to its implementation.

2.4 Methodology Behind the Design

The design process was iteratively developed and adjusted based on feedback received. Initial prototypes were created using the *Figma* tool. The decision was made to pursue a simple design to accommodate the diverse users worldwide. The color scheme was selected to align with the company's branding while creating a visually appealing and fresh design. The reliable structure provided by *Camel Karavan* served as a basis for inspiration, ensuring that the custom *Camel* integration design incorporated established principles from a trusted source. Insights from relevant articles like [2] and [3] were integrated into the methodology to enhance the design process.

2.5 User Testing Methodology

The final testing phase encompassed two distinct user groups, each tailored to the intended audience for the respective visualizations. However, it is essential to note that while these visualizations cater to specific user types, they are designed to provide valuable insights to a wide range of users.

Visualize of *Generic* integration testing was conducted with users who manage these integrations but may not possess extensive technical expertise in data integrations. This testing took the form of task-based scenarios with observation and solicited feedback.

Conversely, the visualization for *Camel* was tested with experts familiar with the logic behind their integrations and integration developers. Participants were allowed to interact with the visualization, provide feedback, and engage in a brief discussion.

Throughout the testing process, careful consideration was given to the subject's work environments, and a well-thought approach was employed, drawing upon insights from the course about psychology in human-computer interaction gained during studies.

Chapter 3
Data Integration

This chapter discusses the whole concept of data integration, the platforms for storing data, and critical concepts necessary for functional data integration. Furthermore, it provides insight into integration solutions and platforms that use such solutions. The latter part of this chapter addresses the modern challenge of big data integration.

To clearly understand the discussed problematics, it is essential to know what data integration is. Data integration is the systematic process of consolidating and transforming data from diverse sources into a unified, cohesive format. The primary goal is to provide users with seamless and standardized access to various data subjects and structures. This practice is driven by the need to meet the informational requirements of various applications and business processes.

3.1 Context of Data Integration

This context serves as a comprehensive introduction, offering a broad overview that signals the forthcoming exploration of foundational aspects and background to the reader. This section will delve into the importance, evolution, use cases, and challenges related to data integration, providing a better understanding of its role in the data landscape.

3.1.1 Importance of Data Integration

In the modern world, data has become the lifeblood of nearly every aspect of our daily lives. From personal activities to business operations, we rely on data to advise decisions, drive innovations, and enhance efficiency. Nowadays, data are primarily stored and processed through electronic platforms. Therefore, effective data management, which includes data integration, has become essential.

Enterprises and organizations, in particular, navigate extensive volumes of data generated from various sources, such as customer interactions, transactions, and operational processes. The challenge lies in collecting this data and, more importantly, in extracting meaningful insights from it. That is why data integration emerges as a critical factor in ensuring that businesses can harness the full potential of their data.

To further clarify the importance of data integration, some vital benefits that showcase its indispensable role in the modern industry landscape follow.

- **Increased reach and impact**
  Participating in a data integration project increases the accessibility of one's data, allowing it to reach a wider audience and have a more significant impact.[4]

- **Clear attribution and credit**
  A data integration system can be structured to maintain clear attribution of data, even when results are compiled from multiple sources, ensuring appropriate credit is given to the data owners.[4]

- **Error reduction and efficiency**
  Manual data gathering involves the risk of human errors. In contrast, data integration minimizes these errors by automating the process, resulting in not only reducing the burden of double-checking but also enhancing overall efficiency.[5]

- **Improvement of data quality**
  The integration process is designed to identify and address data issues automatically. Continuous improvements contribute to more accurate data and analysis, ensuring the organization works with reliable and high quality.[5]

- **Accelerated insight time**
  Data integration facilitates rapid access to information, enabling swift analysis. This agility is vital for timely decision-making.[6]

- **Reduction of data silos**[1]
  Data integration combines information from diverse sources and systems, presenting a unified perspective. This dismantling of data silos enables organizations to eliminate redundancies and inconsistencies that typically result from isolated data sources.[6]

- **Data-driven innovation**
  Integrated data exposes hidden patterns, trends, and opportunities that may remain unnoticed when operating with enterprise data scattered across disparate systems.[6]

3.1.2 Evolution of Data Integration

The evolution of data integration is a testament to the adaptability and ingenuity in the face of growing data complexity and technological advancements. From the early days of manual, labor-intensive processes to the current era dominated by algorithms and artificial intelligence, data integration has continuously evolved to meet the increasing demands for efficient and effective data management solutions.

- **The First Age: The Years of Humans [7]**
  In the earliest days of data integration, the process was predominantly manual, involving significant human effort to integrate data from multiple sources. This period was marked by structured metadata and rule-based systems, where human operators played a crucial role in matching, merging, and harmonizing data sources to achieve a unified data view.[8, 7]

- **The Second Age: The Years of Algorithms Assisting Humans [7]**
  As the digital age progressed, the diversity and volume of data sources expanded. This era saw the advent of algorithms designed to assist humans in the data integration process. Information retrieval techniques, graph theory, and early machine learning approaches provided much-needed support. Conceptual frameworks like match operators and similarity matrices facilitated more structured and efficient data integration processes. Mediators and polystore systems emerged to manage the increasing diversity and volume of data sources, adapting to handle both structured and unstructured data.[8, 7]

[1] Data silo refers to an isolated set of data within a single department from the rest of the organization.

- **The Third Age: One Algorithm to Rule Them All [7]**
  The most recent phase in the evolution of data integration has been defined by the rise of machine learning and deep learning technologies. These technologies have shifted the paradigm from human-centric to algorithm-driven processes, where data from an ever-widening array of sources, including unstructured data, can be integrated with minimal human intervention. Prototypical systems like *ADnEV*[9] and *PoWareMatch*[10] illustrate the potential of combining human insights with algorithmic precision, leveraging deep learning to navigate the complexities of modern data ecosystems. This era envisions a collaborative ecosystem where deep learning algorithms and human insights combine to achieve more accurate and efficient data integration processes.[7]

- **The Future of Data Integration**
  The data integration field is poised for further evolution, driven by the relentless growth in data complexity and the continuous innovation in computational technologies. The integration of advanced AI models, such as deep learning and foundation models, into data integration practices, offers promising avenues for further advancement. Managing data lakes and integrating diverse data types pose ongoing challenges that require innovative solutions, emphasizing the need for data integration systems.[7]

  Additionally, federated learning emerges as a novel paradigm with the potential to revolutionize data integration by facilitating collaborative model training across diverse data ecosystems without compromising data privacy. Integrating federated learning into data integration practices heralds a future where data integration becomes more decentralized, privacy-focused, and inclusive of diverse data types and sources. This progression promises to enhance the capability of data integration systems to deliver actionable insights, driving informed decision-making across various domains.[11]

3.1.3 Challenges in Data Integration

Integrating disparate data sources into a cohesive and functional system presents many challenges. These obstacles originate from semantic and structural discrepancies among various data models and contain broader issues related to data organization, quality, privacy, and infrastructure technology. This section delves into the multifaceted challenges of data integration, shedding light on the semantic and structural hurdles and the logistical and procedural issues organizations face in creating integrated data systems.

3.1.4 Semantic and Structural Discrepancies

The challenge of semantic and structural discrepancies in data integration arises from the diverse nature of data sources, each using different schemas, naming conventions, and structures. The mappings between data sources and the unified schema are categorized into three models:

- **GAV (Global As View)**
  GAV approaches define a global schema as a view of the source schemas. It uses local schemas to describe intermediate schemas. Adjustments in source schemas require updates in the global schema and mappings, leading to potential maintenance issues as the number of data sources grows.[12]

- **LAV (Local As View)**
  Contrastingly, LAV describes each source schema as a view over the global schema, easing the integration of new data sources. This flexibility, however, increases the complexity of query processing, as queries against the global schema must be reformulated into executable queries against the local schemas.[12]

- **GLAV (Global and Local As View)**
  GLAV combines GAV and LAV advantages, offering a flexible framework for data integration by accommodating a variety of source schemas and facilitating easier adjustments to data environment changes. This approach helps balance GAV's rigidity and LAV's query processing complexity.[12]

To further address this challenge, the LSTM unfolding model is used for efficient data matching.

- **LSTM (Long Short-Term Memory) [12]**
  LSTM introduces a dynamic approach to addressing the semantic and structural discrepancies in data integration through advanced data matching techniques. This method stands out for its adaptability and learning capabilities, essential in recognizing and reconciling the diverse representations of data entities across various sources. LSTMs have been successfully applied in various tasks, becoming the most cited neural network of the 20th century.[12]

  A standard LSTM cell comprises several key components: a unit, an input gate, an output gate, and a forget gate. This structure allows the cell to maintain information over varying time periods. The three gates within the cell input, cell output, and cell forge control the information's flow, enabling it to effectively remember values across time intervals.[12]

3.1.5 Deduplication and Data Quality

Deduplication directly addresses eliminating redundant data within an integrated dataset, a process crucial for maintaining data quality and integrity. Deduplication involves identifying and removing duplicate records that might appear when merging datasets from various sources. This task is particularly challenging due to discrepancies in data representation across different systems, such as variations in naming conventions or formats.[12]

There are known methods for matching data, such as blocking approaches. These methods process all records in the datasets, inserting each record into one or more blocks, within which all pairs are considered candidate records, which potentially refer to the same entity. However, they can have a problem in cases like when there are several records with the same record. This comparison between all records additionally creates a large search space.[12]

Therefore, article [12] proposes a new method that reduces the search space by relying on integrating the data in the form of a graph. This makes the search process easier by searching for records that share the same entity. Followed by a cosine similarity algorithm to calculate the similarity between the related entities.[12]

3.1.6 Privacy

Maintaining data privacy during data integration involves navigating complex landscapes, technological constraints, and organizational policies to protect sensitive information from unauthorized access or violations. The *General Data Protection Regulation*

(*GDPR*) sets out key principles for processing personal data, including purpose limitations (collecting data for specific explicit reasons), accuracy, storage limitation, and more. These principles directly impact the organization's IT systems. However, no single product can address all privacy concerns, so organizations must ensure their solutions work together to achieve accurate *GDPR* compliance.[13]

Notable privacy challenges include:

- **Limiting Data Ownership and Access**
  Establishing clear data ownership and access controls helps prevent unauthorized data sharing. Policies should dictate who can access data, under what circumstances, and the protocols for sharing data both internally and externally.[14]

- **Sensitivity of Aggregated Data**
  Special attention should be given to the aggregation of data from multiple sources, which can inadvertently reveal sensitive information.[14]

3.2 Models of Data Integration

This section delves into the crucial milestones in the development of data integration strategies, tracing the path from federated database systems and data warehouses to the advent of mediators and the emergence of polystore systems. Each model represents a critical response to the unique challenges posed by the data landscapes of their time, reflecting shifts in technology, data management needs, and the overarching goal of maximizing the value of integrated data.

3.2.1 Federated Database System (FDBS)

FDBS embodies an integration model that unifies a collection of cooperative yet autonomous database components. This model is built on the principle that each participating database retains its independence while contributing to a collective repository accessible through a federated system. The backbone of FDBS is the federated database management system (FDBMS), which acts as middleware. This system orchestrates the interaction between disparate databases, facilitating a seamless integration process.[8]

The architecture of FDBS 3.1 is structured around five key schema levels, which provide a multi-layered approach to data integration:

- **Local Schema:** Defines the conceptual schema in the native data model of each component database.[8]
- **Component Schema:** Translates the local schema into a canonical or common data model.[8]
- **Export Schema:** Represents a subset of the component schema that's made available for federation, incorporating access control mechanisms.[8]
- **Federated Schema:** Aggregates multiple export schema to form a comprehensive view of the integrated data.[8]
- **External Schema:** Offers customized views of the federated data tailored to specific user requirements or applications.[8]

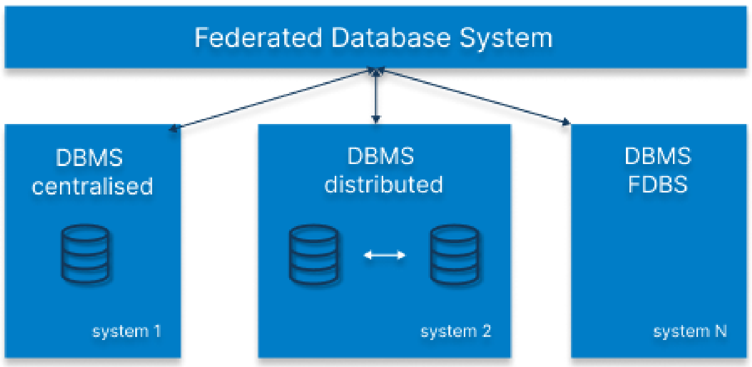


Figure 3.1. FDBS architecture [8]

An essential aspect of FDBS is its provision of specialized processors that facilitate information mapping between the underlying data sources.[8] These processors are crucial for:

- **Transforming** internal command languages into the local query languages of the respective databases.[8]
- **Filtering** operations based on the access controls defined within the export schema, ensuring that only permitted actions are executed on a given component.[8]
- **Constructing** the unified view through query decomposition and data merging.[8]

This layered architecture and the sophisticated processing capabilities of FDBS underscore its effectiveness in managing complex data integration scenarios. By maintaining the autonomy of individual databases while providing a unified access point, FDBS addresses the challenges of integrating diverse data sources in a distributed and heterogeneous computing environment.[8]

3.2.2 Data Warehouse (DW)

A data warehouse integrates diverse data sources into a cohesive and structured repository under a unified schema.[15] Typically represented by a relational database located on the mainframe or cloud of an enterprise.[8] Data warehouses are based on four fundamental characteristics:

- **Subject-Oriented**: They focus on specific subject areas (e.g., sales, customer interactions), allowing for targeted analysis and efficient data organization.[8]
- **Integrated**: Data warehouses consolidate data from diverse sources, ensuring consistency and compatibility across different data types and locations.[8]
- **Nonvolatile**: Once data is entered into a data warehouse, it remains unchanged.[8]
- **Time Variant**: Data within data warehouses is associated with time stamps, enabling tracking over defined periods.[8]

These core attributes underscore the DW's role in facilitating structured, reliable data analysis across various dimensions of business operations.

There have been many different DW architectures over time. The basic generalized architecture of DW, depicted in figure 3.2 can be represented by:

- **Data Sources**
  The origin points from which data is sourced. This includes:
  - **Operational Systems:** Systems used to process the day-to-day transactions of an organization, facilitating the management of business operations. [16]
  - **Flat Files:** Systems of files where transactional data is stored, with each file in the system having a unique name. [16]

- **Central Warehouse**
  The main repository for storing data after they are processed and structured for analysis.

- **Platform**
  Interfaces and tools that enable end-users to perform data analysis, reporting, and mining, are often supported by online analytical processing (OLAP) engines for multidimensional analysis.[8]

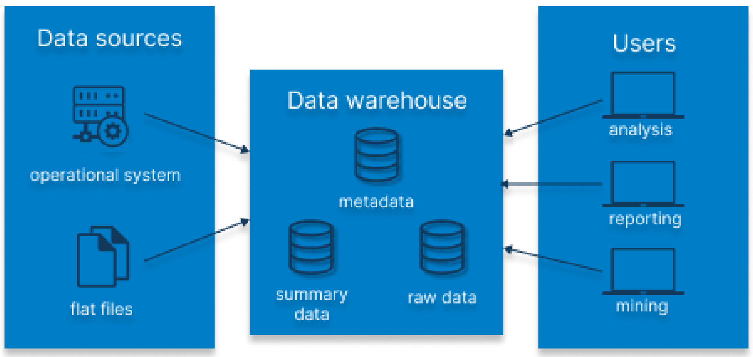


Figure 3.2. DW architecture [8]

The choice of architecture, ranging from virtual warehouses and data marts to comprehensive enterprise warehouses, is tailored to meet an organization's specific analytical needs and business requirements.

Another perspective on the architecture of data warehouses focuses on the methodologies involved in their construction, highlighting two main approaches: the top-down approach and the bottom-up approach. This viewpoint examines the strategies behind building a data warehouse, each with its unique advantages and implications for how data is integrated, processed, and made available for analysis.

- **Top-down Approach**
  This method starts with establishing a centralized data warehouse as the core repository, encompassing the entire organization's data. This central data warehouse then serves as the foundation from which specific data marts are derived, each tailored to particular business functions or requirements. This approach ensures consistency across data marts, providing a unified dimensional view that facilitates comprehensive business intelligence and decision support across the organization. The top-down approach is lauded for its robustness, making it suitable for large-scale enterprises undergoing frequent business changes.[15]

  Figure 3.3 depicts the structure of the top-down architectural approach, where the ETL, further described in section 3.4.1, represents the staging process.

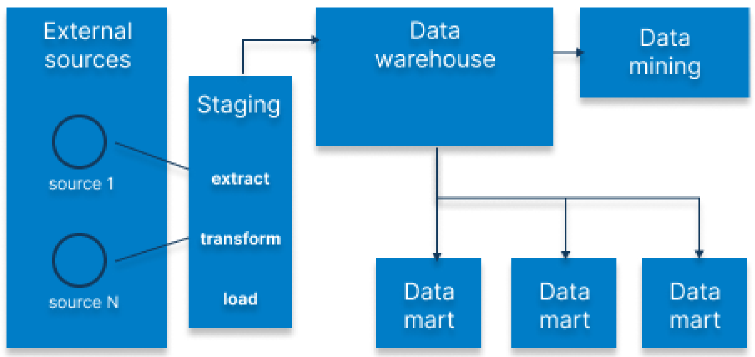


Figure 3.3. DW Top-down Approach [15]

- **Bottom-up Approach**

  This strategy advocates for starting the data warehouse construction by first developing individual data marts focused on specific business areas. These data marts are designed to address immediate analytical needs and provide rapid reporting capabilities. Over time, these data marts are integrated to form a comprehensive data warehouse. The bottom-up approach allows for quicker realization of benefits and offers greater flexibility, making it particularly appealing for organizations looking to build their data warehousing capabilities incrementally.[15]

  Figure 3.4 depicts the structure of the bottom-up architectural approach, where once again the ETL 3.4.1, represents the staging process.

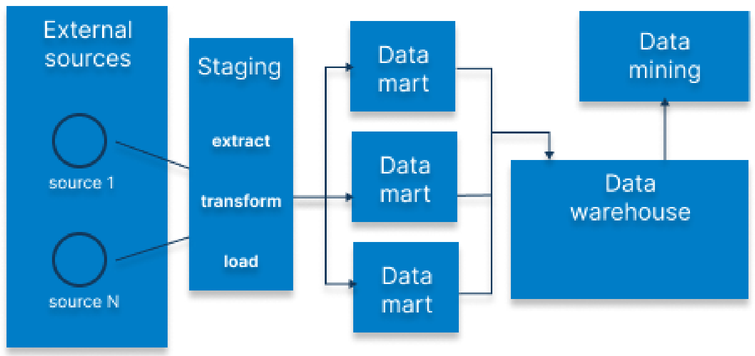


Figure 3.4. DW Bottom-up Approach [15]

The choice between top-down and bottom-up strategies influences DW architecture's design, implementation, and evolution. The top-down approach provides a structured, organization-wide framework for data integration, promoting consistency and scalability. Contrariwise, the bottom-up approach allows for more agile development and can be more responsive to specific business needs, facilitating quicker deployment of analytical capabilities.

3.2.3 Mediators

Mediators provide a virtual view of data stored across different sources, maintaining the data's original locations. They utilize a virtual schema that integrates the schemas from these sources and offer metadata to define the integration schema and the external schemas related to each data resource in the federation. Mediator architecture employs data mapping through wrappers for querying and matching data across sources. This allows a user's query to be translated into multiple queries suitable for different data sources, with the results then consolidated and delivered to the end-user.[8]

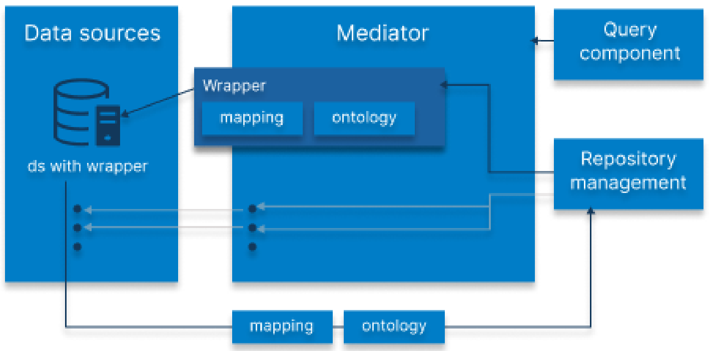


Figure 3.5. Mediator architecture [8]

3.2.4 Polystore Systems

A polystore system is an advanced database framework that provides integrated access to various data stores through a single interface.[8] It offers a solution to the challenges posed by big data diversity, enabling interaction with multiple storage models like *NoSQL*, *RDBMS*, and *HDFS*, sometimes even employing data processing frameworks to facilitate this interaction.[17] These systems are an evolution of federated databases, uniquely equipped to manage and query across various data models housed in separate data stores.[17] This is illustrated in a basic polystore architecture in figure 3.6.

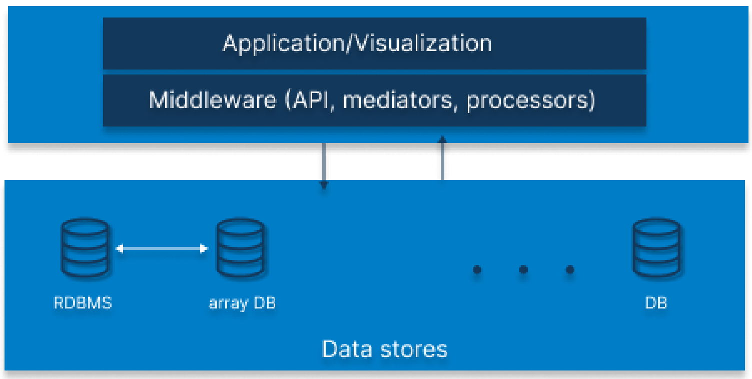

Application/Visualization
Middleware (API, mediators, processors)
RDBMS
array DB
DB
Data stores

Figure 3.6. Polystore architecture [8]

Polystore systems have been the focus of various academic and corporate research endeavors. One notable example is *MIT's BigDAWG* architecture. Similarly, *Microsoft's PolyBase* represents another approach to polystore systems, integrating SQL server parallel data warehouse with *Hadoop*.[8] The *BigDAWG* polystore system addresses the integration of heterogeneous data by balancing location transparency and semantic completeness, accommodating various databases and data models. It offers a middleware solution that provides a uniform query interface while preserving the unique capabilities of each database. Despite its innovations, *BigDAWG*, much like earlier systems, relies on specific adaptations to fully support the semantics of connected databases.[17]

3.2.5 Comparison

FDBS, mediators, and polystore systems share a foundational similarity in their architecture. They are all virtual models based on the concept of data virtualization.

These systems utilize data virtualization techniques, leveraging wrappers for data access and employing query mapping for data retrieval, implying their roots in distributed database systems.[8]

Within these virtual architectures, the primary distinction arises in how they process queries. FDBS utilizes federated query agents (FQA) that can act as intermediaries, storing and executing queries. Mediators take a different approach, using wrappers to map and merge data across queries. Polystore systems employ diverse query processors to accommodate the variety and volume of data.[8]

Another notable difference is the extent of support for the data model. FDBS traditionally supports a single, mainly relational data model. In contrast, mediators and polystore systems are designed to handle multiple data models, reflecting their adaptability to the diverse nature of contemporary data.[8]

Data warehouses stand apart with their physical architecture, centralizing data storage and relying on the Extract-Transform-Load (ETL) process for data integration.[8]

Table 3.1 summarizes the comparison between the mentioned data model architectures.

Features	FDBS	DW	Mediator	Polystore
Architecture	Virtual	Physical	Virtual	Virtual
Distribution	Distributed	Central	Distributed	Central
Automation	Mapping	ETL	Wrappers	Wrappers/ Mapping
Query Processor	FQA	OLAP	Mapping and Merging	Island, HDFS bridge and more
Data Models	Single	Single	Multiple	Multiple

Table 3.1. Comparison of Data Integration Models across different system architectures.[8]

3.3 Data Platform

A data platform is a set of technologies incorporating computational, architectural, and foundational aspects. It transcends a data lake by storing massive, unsorted data and refining and structuring it into actionable information. Creating a data platform involves establishing a data lake and then constructing the platform to serve applications with structured, contextualized data. This indicates a shift from merely collecting data to making it usable and insightful for various applications.

3.3.1 Data Platform Structure

A data platform is structured in layers, where data flows from its source to application use. It begins with the data source layer, producing data in its format through various sources like sensors or automated measurement systems. The acquisition and collection layer follows, involving data capture and initial processing, including ETL processes for cloud applications. Following this, the storage layer holds the raw data, effectively forming the data lake's foundation. The database layer further refines this data into business-relevant information through ingestion processes. Lastly, the application layer utilizes this organized data to address specific business challenges, completing the data's journey from raw input to strategic assets.[18] This structure is depicted in 3.7.

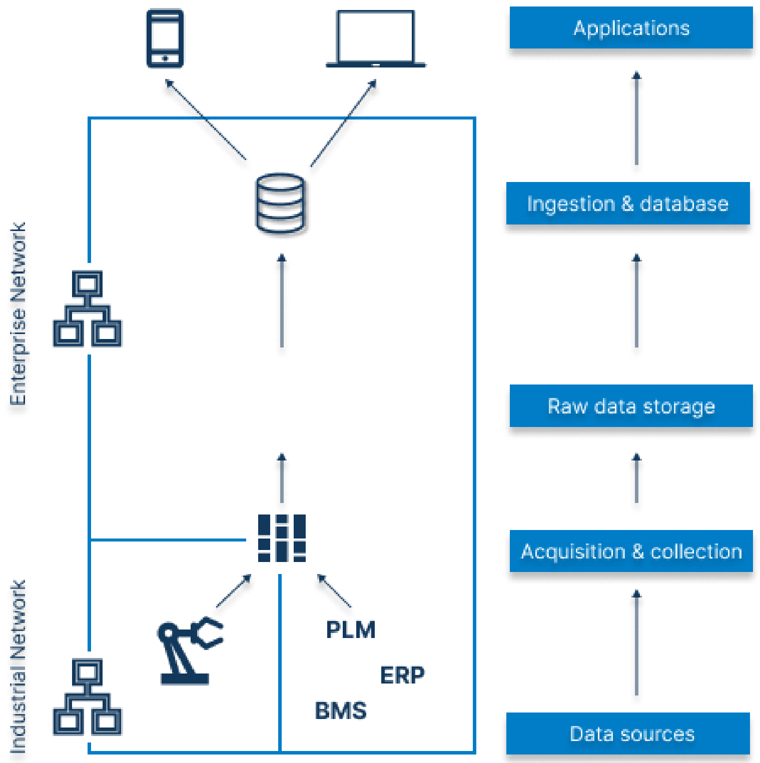


Figure 3.7. Industrial Data Platform [18]

3.3.2 Data modeling

Based on data modeling, data platforms are designed and developed. This subsection showcases the difference between traditional data modeling and big data modeling.

- **Traditional Data Modeling**
  Traditional data modeling has been foundational to managing transactional and analytical processes within organizations. Online Transaction Processing (OLTP) systems utilize entity-relationship models to handle a large number of transactions efficiently, optimize for real-time data manipulation, and adhere to the principles of normalization to avoid redundancy and inconsistencies.[19]

  In parallel, online analytical processing (OLAP) systems focus on data analysis from multiple database applications, prioritizing query performance and the integration of business metrics. Dimensional modeling serves as a foundational technique in data structuring, particularly suited for quantitative data analysis. In practice, dimensional modeling simplifies data by denormalizing it within a central fact table, surrounded by various dimension tables. This setup, commonly called the star schema, facilitates efficient data exploration and analysis across multiple dimensions. Additionally, the snowflake schema represents a variant of dimensional modeling, further normalizing dimensions to elaborate the hierarchical structure within each dimension, allowing for more detailed analytical queries.[19]

  Despite their effectiveness in structured environments, traditional data models, reliant on relational databases, face scalability and performance challenges as data volumes grow. The cost and complexity of scaling these systems, alongside increased query response times, highlight their limitations in the context of big data.[19]

- **Big Data Modeling**

The advent of big data, characterized by its volume, velocity, variety, veracity, and value, necessitates a departure from traditional SQL-based modeling. Big data's complexity and the need for faster data delivery demand a more flexible approach, incorporating procedural or functional languages to handle arbitrary logic beyond SQL's capabilities.[19]

This shift is not just technical but also conceptual, moving towards denormalization and the integration of historical and real-time data to generate insights. Big data modeling rejects the hierarchical, siloed structures of traditional models, instead advocating for a democratic usage pattern where data from multiple sources and of various types is integrated, posing a challenge to the scalability and query performance of dimensional models.[19]

Big data platforms require robust, scalable data modeling methods to accommodate the dynamics of big data, including the need to process data with varying velocities. These methods must balance cost, quality, and performance, ensuring that data can be efficiently organized, stored, and analyzed to drive strategic decisions and competitive advantages.[19]

Table 3.2 compares these two approaches to data modeling.

	Traditional Data Model	**Big Data Model**
Approach	First design then develop	First discover then analyze
Usage Pattern	Top-down, hierarchical	Democratic, distributed
Volume Growth	Manageable volume with steady growth	Massive volume with exponential growth
Main Goal	Business Analytics	Statistical Analysis, Machine Learning

Table 3.2. Comparison of Traditional and Big Data Modeling approaches.[19]

3.4 Key concepts of Data Integration

In the scope of data integration, multiple underlying concepts work in concert to enable the seamless fusion of data across various platforms and systems. The following subsections delve into some of the fundamental principles that are instrumental for a robust data integration strategy. These include the ETL, ELT, and reverse ETL processes, which serve as the backbone for data movement and transformation. Additionally, data quality, APIs, services, batch processing, and scheduling are explored for their pivotal roles in ensuring that data integration processes are efficient, scalable, and aligned with business requirements. While the concepts discussed here are crucial, they represent only a portion of the extensive array of techniques and practices that underpin a successful data integration framework.

3.4.1 ETL, ELT and Reversed ETL

One of the pivotal concepts in data integration involves processing and organizing data according to the specific requirements of data warehouses. This fundamental aspect is crucial for the strategic handling and analysis of large datasets collected from diverse sources. In the subsequent discussion, we introduce the most critical and commonly employed processes in this domain: ETL (Extract, Transform, Load) and ELT (Extract, Load, Transform), alongside their innovative counterpart, reverse ETL. These methodologies serve as the backbone for data integration strategies and also play a vital role in optimizing data flow, enhancing decision-making processes, and enabling a more agile response to business needs.

- **ETL**
  ETL is a traditional method where data is first extracted from the source systems, transformed into a format suitable for the data warehouse, and finally loaded into the data warehouse. This process is ideal for dealing with structured data and legacy systems, where transforming data before loading is necessary to ensure it fits the data warehouse schema.[20]

- **ELT**
  Conversely, ELT modernizes the approach by extracting data from source systems and loading it directly into the data warehouse before transforming it. This method is advantageous for handling large datasets and leveraging the computational power of modern cloud-based data warehouses, allowing for more flexible and timely data processing.[20]

- **Reverse ETL**
  Reverse ETL differs significantly by making the data warehouse the source, from which data is extracted, processed to meet the formatting needs of the destination, and then fed into various applications. This approach distributes data across the company, making it actionable and accessible in the tools used by marketing, sales, support, and other teams.[20]

The main differences between ETL and ELT are captured in table 3.3, and the comparison of those two approaches with their reverse approach is summarized in table 3.4. For a clearer understanding of the methodologies, detailed explanations of each operation are provided below:

- **Extract**
  Data is thoroughly examined and selectively retrieved from the source database during this phase. The aim is to gather as much relevant data as possible with minimal resource expenditure, ensuring the process does not negatively impact the performance or response times of the source system.[20]

- **Transform**
  This step involves cleaning and refining the data, which may include merging it with other datasets or utilizing tools and queries to restore it to a usable state. It encompasses validating records, excluding irrelevant data, and consolidating data sources to ensure consistency and relevance. Standard practices in this stage include organizing, deduplicating, standardizing, decoding, and verifying data integrity.[20]

- **Load**
  In this stage, the processed data is transferred into the data warehouse. This involves compiling the transformed data in a structured format that fits a targeted data repository. Various tools facilitate the integration of extracted and transformed data into the warehouse, either by automating the insertion of each record or by adding records to specific database tables through SQL commands.[20]

Parameter	ETL	ELT
Optimal Use	Structured data, legacy systems, and relational DBs; transforming data before loading to data warehouse.	Quicker, timely data loads, structured and unstructured data, and large datasets; transforming data as per need.
Privacy	Personal Identifiable Information can be eliminated in the Pre-load transformation step.	Major safeguards for privacy are required since data is directly loaded.
Transformations	Secondary server performs the transformations. Pre-cleaning and heavy transformation are optimal.	Higher speed and efficiency is achieved since the database compute performs transformations for load and transform simultaneously.
Maintenance	High maintenance due to the presence of multiple processing servers.	Reduced maintenance burden because of fewer systems.
Expenses	Monetary issues due to separate servers.	Less Monetary overhead because of simplified data stacks.
Compatibility with Data Lake	Not there with ETL.	There with ELT.
Output of Data	Structured.	Can be unstructured or semi-structured.
Amount of Data	Datasets of Small and moderate volume.	Datasets of Large volume.

Table 3.3. Difference Between ETL and ELT[20]

3.4.2 Data Quality

As previously highlighted in section 3.1.5, data quality emerges as a challenge within data integration, but it is also a core concept for quality of data integration. Managing

Parameter	ETL/ELT	Reverse ETL
Synchronization Mode	There can be full or incremental data extraction. UPSERT operation to merge data.	CDC is difficult to apply in reverse ETL as the warehouse typically doesn't provide a transaction log or "updated_at" columns.
Data Transformations	From specific to general. Extracts data from different specific sources to then integrates it into a common destination.	From general to specific, having to conform to each business application API.
Data Quality	Less data quality overhead as the destination is the database/data warehouse.	High data quality overhead as more validation and knowledge of the destination are required.
Failures and Job Re-execution	ETL/ELT jobs are idempotent, meaning that no matter how often you run them, they should produce the same results.	Reverse ETL jobs are not idempotent since the re-execution might result in unwanted side effects as they depend on the business logic of the destination.

Table 3.4. ETL, ELT and Reverse ETL Differences[20]

data quality involves establishing strict standards and mechanisms to ensure accuracy, consistency, and reliability across all data utilized within an organization. Essential to achieving high data quality are:

- **Data Integrity**
  Implementing policies and technical constraints to maintain data consistency and prevent invalid data entries (e.g., ensuring numeric fields like salaries are above zero). Techniques such as functional dependencies and denial constraints can be applied at the application or database level to enforce these rules universally.[14]

- **Testing**
  Data processing often requires numerous adjustments, making the ability to rapidly test hypotheses against the data valuable for maintaining relevance and accuracy in evolving datasets.[14]

- **Data Version Control**
  Managing and documenting data modifications by different organizational actors is crucial for tracing data changes back to their sources, facilitating a more straightforward resolution of issues arising from erroneous data updates.[14]

3.4.3 API and Services

APIs and services are indispensable for data integration due to their ability to streamline the flow of information between different systems and platforms. APIs, acting as connectors, enable real-time data exchange and manipulation. Services, particularly those built on a microservices architecture, support this by encapsulating business logic into discrete units that can communicate over the network, making them highly adaptable and easier to integrate with other services and applications. This combination not only accelerates development cycles but also enhances the robustness and scalability of data ecosystems, making APIs and services critical components in achieving comprehensive and efficient data integration strategies.

3.4.4 Batch Processing and Data Streaming

The integration of batch processing and data streaming allows organizations to leverage the benefits of both methods. While batch processing provides a reliable framework for dealing with large data sets and complex computations, data streaming offers agility and the ability to react to new information instantly. A summary of the comparison of those processes is in table 3.5.[21–22]

Modern data architectures often incorporate a hybrid model combining these approaches, enabling deep analysis of accumulated data while supporting real-time analytics and responses.[21–22]

- **Batch processing**: efficiently handles high-volume, repetitive data jobs by accumulating tasks and executing them collectively during off-peak hours. This approach minimizes human interaction and optimizes computing resources, making it ideal for tasks like weekly billing, inventory processing, and report generation.[21]

- **Data Streaming**: caters to the need for low-latency processing of continuously flowing data. Characterized by its real-time nature, streaming data is pivotal for applications that rely on immediate data analysis and decision-making. It is beneficial in scenarios like monitoring brand sentiment through social media, financial fraud detection, or IoT device management, where timely responses are crucial.[22]

3.4.5 Scheduling

Scheduling in the context of data integration is a critical function that orchestrates the timing and execution of various data processing tasks. It ensures that these tasks occur in the correct sequence and at the right time. It enhances the efficiency, reliability, and scalability of data-driven systems.

3.5 Solutions for Data Integration

This section explores the pivotal architectures and middleware solutions that have shaped the data integration landscape, discussing their core principles, benefits, and the challenges they aim to overcome.

3.5.1 Service-oriented Architecture - SOA

Service-oriented Architecture (SOA) is an architectural pattern where services, representing self-contained and well-defined functions, interact to perform activities or exchange data. Initially utilizing technologies like Object Request Broker or Distributed

Criteria	Batch Processing	Stream Processing
Data Scope	Queries or processing over all or most of the data in the dataset	Queries or processing over data within a rolling time window, or on just the most recent data record
Data Size	Large batches of data	Individual records or micro batches consisting of a few records
Performance	Latencies in minutes to hours	Requires latency in the order of seconds or milliseconds
Analysis	Complex analytics	Simple response functions, aggregates, and rolling metrics

Table 3.5. Comparison of Batch Processing and Data Streaming[21–22]

Component Object Model (DCOM) based on Common Object Request Broker Architecture (CORBA) specifications, SOA has evolved with web services as a pivotal connection technology. These web services employ XML to establish robust connections, ensuring clear communication between service consumers and providers.[23]

The SOA model consists of three primary operations (find, bind, and publish) and three participants (service requestor, service provider, and service broker), irrespective of its internal implementation. This framework allows for dynamic interactions between the entities involved:[23]

- **Service Requestor**: Initiates discovery and invocation of services to deliver comprehensive business solutions capable of operating locally and remotely.[23]
- **Service Provider**: Offers software service interfaces representing business entity services or reusable subsystem service interfaces.[23]
- **Service Broker**: Functions as a repository for the published software interfaces by service providers, facilitating the discovery of services.[23]

These components interact through the operations of publishing (performed by the service provider to make services available to the broker), finding (performed by service requestor to locate necessary services through the service broker), and binding (to establish a connection to the services found).[23]

SOA promotes principles like:

- **Modularity**: Where services are designed for specific functions, promoting focused development and easier maintenance.[24]
- **Flexibility**: Enabling systems to adapt through service recombination, enhancing the ability to meet changing business needs.[24]
- **Scalability**: Supporting growth without significant restructuring, ensuring systems can handle increased loads efficiently.[24]

Key challenges include:

- **Service Versioning and Compatibility**: Maintaining backward compatibility while evolving services, ensuring uninterrupted service use.[24]
- **Service Granularity**: Balancing between too coarse or too fine services, optimizing for efficiency and manageability.[24]
- **Service Management**: Requiring transparent governance for consistency and security, overseeing service lifecycle and policies.[24]
- **Data Consistency Across Services**: Ensuring accurate and synchronized information, maintaining integrity in distributed environments.[24]
- **Service Orchestration**: Coordinating multiple services for complex business processes, enabling seamless integration and functionality.[24]

In SOA, an Enterprise Service Bus (ESB) acts as a central hub that facilitates communication among disparate services, regardless of their underlying technologies. It routes service requests to the correct destination and transforms them to match the service's platform and language requirements.[25]

3.5.2 Microservices

Microservices architecture is an approach that structures applications as collections of loosely coupled, independently deployable services. Each focused on a specific business capability. These services communicate through well-defined APIs, promoting agility, scalability, and maintainability. Microservices often employ individual databases for each service to enhance autonomy, with API gateways acting as intermediaries to route external requests efficiently to the appropriate microservice.[24]

Microservices architecture prioritizes principles like:

- **Service Autonomy**: Each microservice operates independently with its own database, facilitating rapid development and deployment.[24]
- **Decentralized Data Management**: Allows each service to manage its data store, promoting data ownership and reducing data coupling.[24]
- **Technology Heterogeneity**: Enabling the use of the most suitable technology stack for each service's specific needs, fostering efficiency and innovation.[24]

Challenges include:

- **Managing the complexities of distributed systems**, such as service discovery, network latency, and load balancing.[24]
- **Maintaining data consistency across independent services**[24]
- **Orchestrating a cohesive business process from autonomous services**[24]
- **Ensuring comprehensive monitoring and observability** requires advanced solutions in a distributed environment.[24]

Table 3.6 compares service-oriented architecture with microservices architecture.

3.5.3 Middleware

Middleware acts as an intermediary software layer that interfaces between applications and various data sources. It addresses key challenges such as system heterogeneity, interoperability, security, and reliability, positioning itself as a crucial component in the architecture of data platforms. By offering a network-oriented perspective, middleware simplifies the complex process of data acquisition, transformation, and storage.[27]

Features and characteristics of Middlewares:

	SOA	Microservices
Implementation	Different services with shared resources.	Independent and purpose-specific smaller services.
Communication	ESB uses multiple messaging protocols like SOAP, AMQP, and MSMQ.	APIs, Java Message Service, Pub/Sub
Data storage	Shared data storage.	Independent data storage.
Deployment	Challenging. A full rebuild is required for small changes.	Easy to deploy. Each microservice can be containerized.
Reusability	Reusable services through shared common resources.	Every service has its own independent resources. You can reuse microservices through their APIs.
Speed	Slows down as more services are added on.	Consistent speed as traffic grows. (requires scaling)
Governance flexibility	Consistent data governance across all services.	Different data governance policies for each storage.

Table 3.6. Comparison of SOA and Microservices[26]

- **Flexibility**: Middleware offers adaptability to handle conflicting issues due to the communication between applications and things, supporting different forms of flexibility essential for varying software or hardware components.[27]
- **Transparency**: It hides complexities and details from both the application and object sides, allowing them to communicate with minimal knowledge of each other's information. This feature is critical for platform and network transparency, enabling cross-platform operations and resource location transparency.[27]
- **Interoperability**: Enables meaningful data and service exchanges between applications on interconnected networks with differing protocols, data models, and configurations.[27]
- **Reusability**: SOA-based middleware allows for the reuse of software and hardware components, making system design and development more efficient and cost-effective.[27]
- **Maintainability**: Systems, applications, or devices can rapidly return to normal functionality post-failure, requiring well-defined procedures and infrastructures for effective maintainability.[27]
- **Adaptability**: Indicates how middleware should behave against environmental changes, reacting both statically and dynamically to provide durability against long-term system changes.[27]

- **Security and Privacy**: Ensures that data transmission and operations through middleware are secure, addressing confidentiality, integrity, and availability, alongside providing privacy protection from unauthorized access.[27]
- **Connectivity Convergence**: Supports diverse hardware and software components interacting through heterogeneous communication platforms, necessitating convergent connectivity APIs and management.[27]

Middleware designs derive from diverse architectural principles. Establishing a well-articulated framework is crucial in middleware design, as each architecture possesses distinct features. Instead of adhering to rigid categorization, middleware architectures can be distinguished based on various criteria, reflecting their unique attributes.[27] Down below are listed some of those architectural principles, followed by table 3.7 summary of benefits and challenges for each of the principles.

- **Component-based**
  This architecture emphasizes loosely coupled, independent components that work together to complete tasks. Each component addresses a specific problem segment, aiming for minimal dependency, increased reusability, and simplified troubleshooting.[27]

- **Distributed**
  Involves networked software and hardware components collaborating to perform tasks. Characteristics include component concurrency, no global clock, and individual component failures not affecting the entire system. Benefits include fault tolerance, scalability, and flexibility.[27]

- **Service-based**
  Efficient design style, often implemented as stand-alone or through cloud computing services (PaaS). It focuses on providing and utilizing services over a communication protocol, though it may not be cost-effective for homogeneous systems or suitable for applications requiring strict response times or heavy data traffic exchange.[27]

- **Node-based**
  Comprises software components on mobile and sensor networks, processing and communicating data collected from sensors. This setup, involving streams and nodes, is well-suited for mobile devices.[27]

- **Centralized**
  Centralizes services in a specific location, with thin-client devices requesting resources from a central, resource-rich server. It contrasts with distributed architecture, and failure of the central server can disable the network without an immediate backup.[27]

- **Client-server**
  A classic model where one side requests and the other replies. Roles can be flexible, with devices acting as both client and server. It includes thin client models (server handles processing and data, clients provide GUI) and thick client models (servers manage data while clients handle application implementation and GUI), as well as multi-tier models separating main functions like presentation and application processing.[27]

Architecture	Benefits	Challenges
Component-based	Reusability, Abstraction support and Independency	Maintenance, Migration, Complexity and Compatibility
Distributed	Resource sharing, Openness, Scalability, Concurrency, Consistency and Fault tolerance	Interoperability, Security, Manageability, and Maintainability
Service-based	Reusability, Scalability, Availability, and Platform independence	Service discovery, Complex service management, and Service identification
Node-based	Availability and Mobility	Security and Manageability
Centralized	Simplicity, Security, and Manageability	Scalability, Availability, and Portability
Client-server	Servers separation, Resource accessibility, Security, Back-up and Recovery	Congestion, Limited scalability, and Single Point of failure

Table 3.7. Comparison of different possible architecture designs in middlewares[27]

3.6 Notable Integration Platforms

Various platforms offer tailored solutions for data integration to meet diverse organizational needs. Among these, *Talend* and *Informatica* stand out as two of the most popular and widely adopted platforms. However, it is common for companies to develop their custom data integration platforms, aiming to address specific challenges and integrate seamlessly with their unique IT ecosystems.

3.6.1 Talend

Talend stands out as a comprehensive data integration suite, offering solutions encompassing big data analytics and data security. Its *Talend Data Fabric* provides a unified collection of all *Talend* resources, accompanied by top-tier customer support. For those opting out of cloud solutions, *Talend Open Studio* offers a web-based alternative with extensive support for various cloud and on-premise databases and SaaS connectors.[28]

Embracing open-source principles, *Talend* has been at the forefront of data integration software since 2005, providing tools for data management, storage, and business device integration. This open-source tool allows companies to make real-time, data-driven decisions, enhancing data accessibility and ensuring efficient delivery to the intended networks. With its downloadable resources, *Talend* encourages community engagement.[28]

Talend's approach aligns with the evolving needs of marketing and business strategies, emphasizing the importance of scalable and adaptable solutions for future challenges.[28]

3.6.2 Informatica Power Center

Informatica Power Center is a versatile ETL tool designed to work with a variety of legacy database systems. It supports essential data management functions such as governance, monitoring, master data management, and data masking. Suitable for on-premise use, *Power Center* also has a counterpart in the cloud, broadening its application scope.[28]

Power Center enables the construction of corporate data warehouses, providing professionals with the capability to connect to multiple data sources and perform comprehensive data processing. It accommodates a range of platforms and offers cloud-based applications, easing the data integration process for IT professionals. Key features include real-time data integration, B2B data exchange, and analytics, with support for *Salesforce* integration. [28]

The tool's server, crucial for running data processing campaigns, interfaces directly with the source data, managing modifications before loading to the target destination. Noteworthy for its automation and reusability, *Power Center* allows for efficient and swift backend data operations, from deletions to complex adjustments. It comes equipped with functionalities for row-level data operations and handling data across structured, semi-structured, and unstructured formats, alongside preparing data for execution. Furthermore, its metadata capabilities ensure that information about applications and data processes is effectively managed and safeguarded.[28]

3.6.3 Talend and Informatica Comparison

Talend provides both free and commercial data integration tools. It offers the flexibility of developing intuitive *Java* code that can run on any *Java*-compatible platform, making it accessible and versatile. *Talend* also allows for precise custom code writing and offers reusability options for transformations. Its commercial versions support advanced features like project scheduling and parallel computing, whereas these capabilities are limited in the open-source version.[28]

On the other hand, *Informatica* offers its solutions solely on a commercial basis. It is considered a mature and leading enterprise data integration platform, renowned for its robustness and extensive automation capabilities, especially in deployment. *Informatica* facilitates the creation of reusable components and allows parallel processing, where multiple mapping sessions can run concurrently, optimizing resource utilization. Its repository manager provides features for data retention and recovery, which *Talend's* open-source version lacks. *Informatica*, however, does not allow the creation of subfolders for organizing objects, unlike *Talend*, which offers more flexibility in repository organization.[28]

In terms of pricing, *Talend's* free versions and *Informatica* require payment for single- and multi-user licenses.[28]

3.6.4 Other Platforms

- **StreamSets**
  StreamSets offers a DataOps platform optimized for the cloud, providing real-time data integration with a spark-native execution engine and the capability to construct

data pipelines with minimal coding.[28]

- **Blendo**
  Blendo is a prominent ETL and data integration tool that automates the synchronization and extraction of raw data from various sources, directly loading it into databases to accelerate business intelligence processes.[28]

- **Google Cloud Dataflow**
  Google Cloud Dataflow, built on *Apache Beam*, is a fully managed ETL service by *Google* that is compliant with data privacy regulations like *HIPAA* and *GDPR* and suitable for both batch and stream processing.[28]

- **IBM InfoSphere**
  IBM InfoSphere, an on-premise ETL tool, enables connections to diverse legacy database systems and supports data governance, monitoring, and master data management with a focus on data security.[28]

- **AWS Glue**
  AWS Glue is a managed ETL service that enables serverless data integration using *AWS* services, with features like an interactive data catalog, automatic schema discovery, and serverless ETL pipeline creation with *AWS* lambda functions.[28]

Chapter 4
Problem Analysis

4.1 Onsemi Middleware Platform

The *ON Semiconductor Middleware* integration platform, initially a monolithic application established in 2008, has evolved into a sophisticated microservices-based architecture and now operates as a distributed middleware system. The platform spans 37 virtual servers, including 17 dedicated to production environment. It expertly manages approximately 4,000 active integration flows, handling 5 TB of daily network traffic, making it a vital component of *ON Semiconductor's* global infrastructure.

4.1.1 Onsemi Middleware Architecture Overview

The *Onsemi Middleware* architecture is a complex system comprised of several core components that collaborate to ensure seamless execution of interfaces. Below is an overview of the main building blocks:

- **MDW Dashboard**
  The center for user configuration and management, the *MDW Dashboard*, is where all interface configurations are conducted. It centralizes setup processes, eliminating the need for scattered configuration files or system-level adjustments. This consolidation ensures that interface management is streamlined and accessible from a single control point.

- **Runtime Engine**
  The *Runtime Engine* orchestrates the execution of interfaces. It carries out the operations defined within the *MDW Dashboard* configurations and facilitates communication with the *Delivery Engine* for outbound processing.

- **Delivery Engine**
  The *Delivery Engine*, specializing in outbound delivery, transmits files, emails, and HTTP requests and can also trigger cascading other interfaces. This component ensures asynchronous deliveries, handles retries, and manages the distribution tokens associated with each delivery.

- **Scheduler**
  The *Scheduler* initiates interfaces at predefined intervals based on specific preconditions. It supports a range of preconditions for various sources, including FTP, database queries, and JMS queues.

4.1.2 Onsemi Middleware Environments

The *ON Semiconductor Middleware* platform ensures sustainable development and functionality through a structured multi-environment approach:

- **UE DEV**: A development sandbox for internal development and testing, with restricted access due to potential instability.
- **UE INT**: Used for unit testing and interface development.
- **UE UAT**: The staging ground for all user testing, hosting release candidate versions of applications, making it the final validation step before production deployment.
- **UE PRD**: The production environment. It operates with the highest performance for handling all active interfaces and the bulk of data traffic.

4.2 Used Data Integration Frameworks within the Platform

The *ON Semiconductor Middleware* platform primarily supports two frameworks for data integration: the widely adopted *Apache Camel* and the proprietary framework known as *Generic*.

4.2.1 Generic framework

This in-house framework provides less complex integration scenarios, enabling users to create integrations through a wizard within the *Dashboard*. It is designed to be user-friendly, allowing people without programming skills to configure integrations without deep technical expertise. The *Generic* framework supports a variety of integration patterns, including:

- **F2F (File to File)**: Transfers between FTP/SFTP and *Samba* shares.
- **F2D (File to Database)**: File ingestion into databases like *Oracle, DB2, MySQL, MSSQL, PostgreSQL*, or any JDBC-compatible system.
- **D2F (Database to File)**: Database extractions to file systems over FTP/SFTP with *Samba* shares.
- **D2D (Database to Database)**: Direct database-to-database transactions.
- **Pure HTTP requests**

The *Generic* framework is based on configurated properties, where all integration aspects, from connection parameters to specific operational flags, are defined within the wizard.

4.2.2 Apache Camel Framework

For more complex and specialized integration requirements, the platform utilizes *Apache Camel*, engaging developers to build solutions based on Functional Requirement Documents (FRD). *Apache Camel*-based integrations handle complex processes and support various integration patterns defined within *Camel*. The *Onsemi Middleware* development team implements these integrations. A dedicated section follows to provide a deeper understanding of the complexities and capabilities of the *Apache Camel* framework.

4.3 Apache Camel

Apache Camel represents the pinnacle of integration design within modern software development. It is not merely a library or a set of APIs. It is a comprehensive integration framework grounded in *Java* renowned for simplifying the development of integration solutions. By embracing the power of well-established integration patterns, *Apache*

Camel provides a standardized pathway for building robust integrations. It boasts an impressive array of components, over 300, building blocks for creating connections to many endpoints, including databases, message brokers, web applications, and various cloud services like AWS and Azure. What sets *Apache Camel* apart is its capacity to democratize integration, offering tools that pave the way for both traditional coding and low-code/no-code approaches.[29] *Camel* also stands out for implementing Enterprise Integration Patterns (EIPs), enabling developers to solve integration problems efficiently by applying best practices. It can operate standalone or be embedded within *Spring Boot*, *Quarkus*, and cloud environments. *Camel* supports over 50 data formats, effectively translating messages across various formats.[30]

The core feature of *Camel* is its routing and mediation engine. The routing engine functions by maneuvering messages selectively, where the specific configurations of the routes dictate the direction and flow. In the world of *Camel*, these routes are ingeniously constructed using a blend of Enterprise Integration Patterns and a domain-specific language, crafting a robust and flexible routing framework.[1]

4.3.1 The Functional Architecture of Apache Camel

This section delineates the fundamental principles underpinning *Apache Camel's* architecture and operational mechanisms. It aims to provide an insightful overview of the core components and processes that constitute the framework, facilitating a comprehensive understanding of its capabilities and functionalities.

- **Messaging system**
  In the domain of *Apache Camel*, the concept of messaging is abstracted into two primary entities: message and exchange. Messages are the fundamental data carriers within *Camel's* routing mechanism, allowing systems to communicate through predefined messaging channels. On the other hand, an exchange acts as a wrapper for these messages during the routing process, embodying the message exchange patterns (MEPs) and facilitating the interactions between various system components.[1]
  - **Message**
    A message in *Camel* is an entity that encapsulates data intended to be conveyed from one system to another. It comprises a payload, known as the body, headers, and optional attachments. The body is a versatile container capable of holding any content, while headers provide metadata about the message, such as content encoding and authentication details.[1]

  - **Exchange**
    During routing, messages are contained within an exchange, a *Camel* construct that not only carries the message but also details about the interaction pattern, be it one-way (InOnly) or request-response (InOut). Exchanges are enriched with properties and an ID, enabling robust handling of messages and their corresponding responses, if applicable. These properties persist for the duration of the exchange, providing a means to store and access global-level data that may be necessary for various components and operations within the routing process. In the event of errors during the message routing, exceptions are captured within the exchange, ensuring that any issues can be handled gracefully.[1]

- **Routes**
  In the architecture of *Apache Camel*, routes are fundamental abstractions that guide

the flow of messages. Routes, defined as sequences of processors, enable the creation of sophisticated messaging applications by providing a clear pathway for message delivery. By strategically decoupling clients from servers and producers from consumers, routes empower developers to:[1]

- Dynamically determine the destination of messages based on business logic.[1]
- Integrate additional processing steps flexibly within the message flow.[1]
- Develop clients and servers independently, promoting modularity.[1]
- Foster robust design practices by seamlessly connecting systems with distinct functionalities.[1]
- Amplify the inherent capabilities of systems like message brokers and Enterprise Service Buses (ESBs).[1]

Moreover, every route in *Camel* has a unique identifier, which is crucial for activities like logging, debugging, monitoring, and management of the lifecycle of routes, including their initiation and termination. While a route traditionally has one input source, *Camel's* design also accommodates scenarios where multiple inputs converge into a single route.[1]

- **Domain-Specific Language (DSL)**

  Apache Camel enhances the development of routing logic by using domain-specific languages (DSLs), offering a syntax particularly tailored to route construction. The concept of DSL in *Camel* is interpreted with a certain latitude, denoting a fluent *Java* API that encapsulates methods named after Enterprise Integration Patterns (EIPs). DSLs in *Camel* facilitate the wiring of processors and endpoints to forge routes and provide a high level of abstraction, simplifying the construction of applications within the *Camel* ecosystem.[1]

  Camel offers a spectrum of DSLs to accommodate developer preferences and project requirements, including *Java DSL, XML DS, Spring XML, YAML DSL, Rest DSL, Groovy DSL*, and *Kotlin DSL*.[31]

- **Processor**

  In *Apache Camel's* routing architecture, the processor is a key element responsible for manipulating and routing an exchange. Exchanges pass sequentially through processors, each acting as a node that can process, modify, or route the exchange based on predefined logic. The flow of exchanges from one processor to another is determined by the message exchange pattern (MEP) associated with the route, which dictates whether a response is sent back at the end of the processing chain.[1]

- **Components**

  Components are the primary extension mechanism within *Apache Camel*, serving as the backbone for various functionalities, from data transportation to the execution of domain-specific languages (DSLs). They are identified by a unique name within a URI and act as factories for endpoints, facilitating the connection between the *Camel* route and the external system or resource.[1]

- **Enpoints**

  Endpoints are versatile in functionality, acting as factories for producing and consuming message exchanges. They represent the end of a messaging channel where data is either consumed from or sent to and are configured through URIs to define how and where messages are transferred. At runtime, *Camel* resolves these endpoints using their URI notation to manage the flow of messages. [1]

- **Producers**
  The producer embodies the entity responsible for sending messages to endpoints. Tasked with ensuring message compatibility with an endpoint, the producer abstracts the complexities of various transport protocols. This abstraction is pivotal in *Camel*, as it simplifies the developer's task to define the destination endpoint while the producer manages the intricate details of message delivery.[1]

- **Consumers**
  The consumer represents the service that initiates the messaging process. It is the receiver of messages produced by external systems, wrapping them in an exchange for processing within *Camel's* routes. As the source of exchanges in *Camel*, the consumer is responsible for initiating data flow through the framework.[1]

  Camel supports two primary types of consumers:

  - **Event-driven consumer**
    The event-driven consumer remains idle until a message arrives, exemplifying an asynchronous receiver that activates to process incoming data.[1]
  - **Polling consumer**
    Conversely, the polling consumer, known as a synchronous receiver, actively seeks out messages from a source and processes them before polling for more.[1]

4.4 Existing Solutions for Data Flow Visualization

Many integration platforms have solutions designed to simplify and visualize complex data flows and integrations through low-code approaches. These platforms provide visual design environments that enable users to create integrations by dragging and dropping components, reducing the need for extensive coding and making integration processes more accessible. Below are mentioned some of those platforms.

4.4.1 MuleSoft's Anypoint Platform

Anypoint platform is a versatile integration solution that connects various applications, data, and devices within an organization. It is built around *Mule*, an efficient open-source integration engine, and promotes an API-led connectivity strategy that helps businesses seamlessly design, build, and manage their integrations.[32] *Anypoint Studio*, the platform's visual development tool, offers a user-friendly drag-and-drop interface, making creating and orchestrating integration workflows straightforward and efficient. Figure 4.1 illustrates the example of *Anypoint* platform's integration visualization.

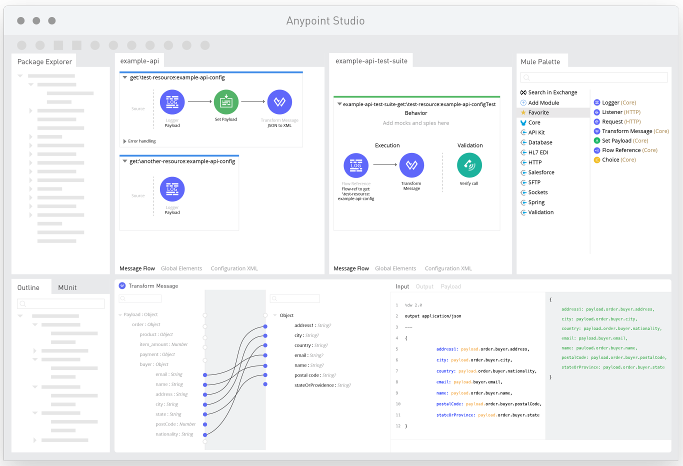


Figure 4.1. Anypoint Studio Integration Example [33]

4.4.2 Oracle Integration Cloud (OIC)

Oracle Integration Cloud (OIC) is a comprehensive integration platform that facilitates low-code application development. It enables users to establish connections between various applications and systems and then utilize a drag-and-drop tool for integration construction. *OIC* provides a mapping tool called the *Oracle Mapper*, which allows users to define how data is transferred between systems. Figure 4.2 represents an example of integration visualization within the platform.

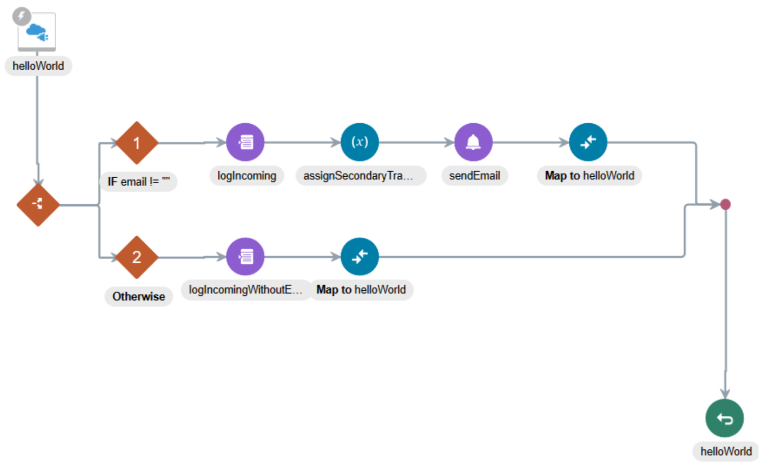


Figure 4.2. OIC Integration Example [34]

4.4.3 Apache Camel Karavan

Karavan stands out in the *Apache Camel* ecosystem as a potent Integration Toolkit. It is designed to make the integration process straightforward. With *Karavan*, users can visually construct and manage integration pipelines/routes, which enhances understanding and control over complex data flows. The toolkit is engineered to work with *Apache Camel* runtimes, facilitating direct packaging, image building, and deployment into Kubernetes environments.[35] An example of data flow in *Camel Karavan* is depicted in Figure 4.3.

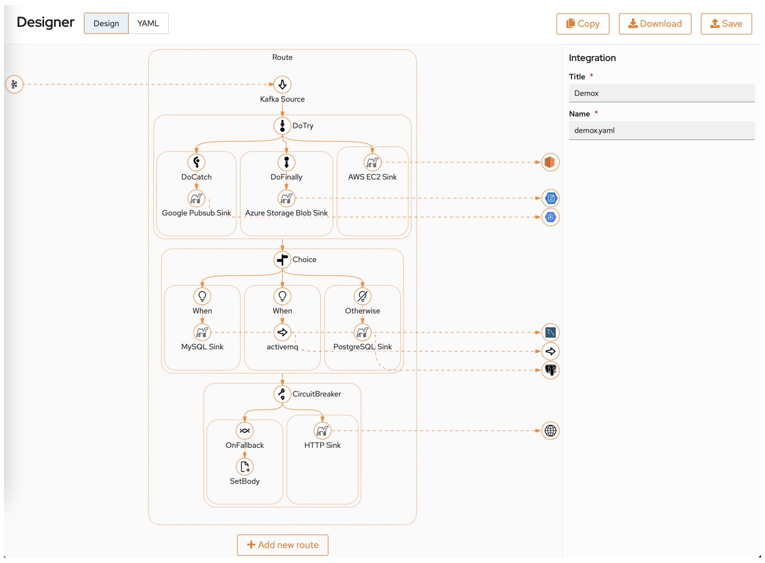


Figure 4.3. Camel Karavan Integration Example [36]

4.4.4 Possible Utilization of Existing Solutions

While these platforms offer significant benefits, none can directly integrate with the *Onsemi Middleware* due to specific technical and operational requirements. Also, these are standalone platforms and not pure solutions for data flow visualization that the thesis aims to achieve. The *Generic* framework's custom nature naturally requires a custom solution for visualization. Even though *Camel Karavan* offers a visualization model, its limitation to YAML configurations does not align with *Onsemi Middleware's* extensive use of *Java DSL*, which enables greater customization and complexity in integration processes. However, *Camel Karavan's* visualization approach could serve as a design inspiration for the visualization of the *Apache Camel* framework within the *Onsemi Middleware Platform*.

4.5 Applicable Techniques for Data Flow Visualization

As previously discussed, existing solutions for data flow visualization do not bring a viable way to visualize integrations within the *Onsemi Middleware* platform. Given the unique requirements and the complex nature of the platform's integration processes, exploring alternative techniques that can be used to fit these needs is essential. This section will identify various approaches and technologies that could be adapted for data flow visualization within the *Onsemi Middleware* platform.

4.5.1 Proposed Technique for the Generic Framework

For the *Generic* framework within the *ON Semiconductor* platform, the approach to data flow visualization is straightforward due to the framework's property configuration nature. The primary task involves comprehensively analyzing the *Generic* framework to understand its underlying structure and the relationships between various components and data flows. Based on this analysis, a suitable backend model can be developed that accurately represents these data flows and their properties. Once established, this backend model must be effectively exposed to the front end, where it can be utilized to create dynamic, interactive visualizations.

4.5.2 Alternative Techniques for the Camel Framework

Given the variability, complexity, and dynamic nature of the *Apache Camel* framework, it is not feasible to build a straightforward model for visualization. Consequently, alternative techniques are required to extract meaningful information from the code to build up a viable structure for effective visualization.

- **Static Code Analysis**
  This method involves parsing the source code of *Camel* routes to directly extract and model the data flows and their properties.

  Obtaining the object structure is the first stage in the static analysis of the source code. The appropriate solution for such obtaining can lead to reduced duration of the entire static analysis process.[37] Some methods for such extraction are described within the paper [37].

  Access to the source code within the platform enables using tools such as *GitHub's Java parser* or other tools to create custom parsing mechanisms for *Camel* interfaces. However, developing a custom parser is a complex task. It requires a deep understanding of the *Camel* framework and its specific integrations. Custom parsers also carry risks of inaccuracies due to the nuances of the code's functional logic and variations in coding styles, which could lead to misleading representations.

- **Dynamic Tracing**
  Dynamic tracing offers real-time insights into the operation of *Camel* routes, capturing live data on system behaviors. This approach involves techniques like:
  - **Logging**
    By capturing logs generated during the execution of *Camel* routes, this approach provides visibility into the functional aspects of integrations, such as route execution, data transformations, and error handling.

    However, its effectiveness hinges on the thoroughness of logging implemented by developers. Without consistent and detailed logging practices across all integrations, the completeness and reliability of visualization based on logs alone could be compromised.[38]
  - **Network Traffic Analysis**
    By monitoring the data packets moving through the system, including the REST API requests and responses analysis, insights into the timing, volume, and direction of data flows and the status and content of HTTP transactions can be obtained. However, they lack a deep understanding of the business logic of integrations.

  Both logging and network traffic analysis can contribute valuable additional information about the integrations. However, neither approach, alone or in combination, can sufficiently create a structured data flow model. They cannot fully capture the complexity and interconnectedness of advanced integration patterns and the business logic embedded within them.

- **AI-driven Analysis**
  Leveraging advanced AI technologies can extract viable information from the code that could be used to build up a model. Such methods include:
  - **Pattern Recognition**
    This method could identify common coding patterns and their variations within *Camel* routes, which can help automate understanding of complex code structures.

- **Semantic Analysis**
  This method could capture the contextual meaning of the code, offering a deeper understanding of the business logic and functionalities embedded within the routes.
  For such analysis, approaches like deep code comment generation can be used to bring valuable insights.[39]

However, while AI-driven analysis promises sophisticated insights, it comes with significant risks:

- **Reliability Concerns**
  AI models depend on the quality and variety of data they are trained on. Errors in model predictions can lead to unreliable visualizations, which misrepresent the actual operations and interactions within the *Camel* routes. Such inaccuracies could reduce trust in the visualization tool, leading to underutilization and limiting its overall contribution to the project.
- **Security Implications**
  Providing AI models with access to source code might pose security risks, especially if sensitive or proprietary information is involved. The need to align with company security policies and protect intellectual property could restrict the feasibility of implementing AI-driven analysis.[40]

Chapter 5
Implementation

This chapter delineates the implementation methodology articulated by the thesis for constructing and visualizing structural representations of data integrations within the enterprise's technological framework. It commences with an exposition of the technologies pivotal to the implementation, laying the groundwork for the intricate development narrative. Detailed within are the distinctive strategies and models applied, with a bifurcated focus on the *Generic* and *Camel* types of integrations, each necessitating a tailored approach due to their inherent specificities. This segmentation allows for an in-depth discussion of the code implementation and the adaptation of patterns that underpin the extended functionality of the platform. The descriptions traverse the applied coding practices and explain the conceptual models that serve as the backbone for the visualization techniques employed. By doing so, this section aims to provide a comprehensive account of how the research findings and theoretical constructs have been translated into a functional and scalable extension, illustrating the interplay between abstract patterns and their concrete implementations within the integration environment.

5.1 Used Technology

The implementation detailed within this thesis serves as an extension of the existing integration platform. Therefore, technological compatibility is crucial. The selection of technologies and development methodologies was carefully aligned with the platform's current ecosystem and operational paradigms, aiming to introduce seamless integration.

5.1.1 Development Approach

To effectively manage the complex process of enhancing a sophisticated integration platform, the project employed a development strategy influenced by Agile methodologies.

This approach allowed for more flexibility and adaptability in addressing the complexities involved. This iterative and incremental model facilitated a dynamic development environment where progress was continuously evaluated and adapted based on ongoing feedback and reflections. Moreover, Agile's emphasis on customer collaboration over contract negotiation ensured that the development process remained responsive to evolving customer needs.[41]

5.1.2 Technologies and Tools used for Implementation

This section outlines the key technologies and tools employed in developing the platform extension.

- **Jira**
  As the cornerstone for task organization and management, *Jira* facilitated the Agile project management process, ensuring tasks were clearly defined, organized, and tracked through to completion.

- **Confluence**
  Confluence served as the knowledge base for the extension. It housed the documentation, including manuals, development guidelines, and architectural decisions.

- **Git**
  Employed for version control, *Git* enabled iterative development and management of the codebase's evolution. This tool was essential in supporting the Agile-based workflow adopted by the project.

- **Spring Boot Framework**
  The main framework of the integration platform. The implementation utilized *Spring Boot's* advanced features, such as custom annotations, dynamic bean configuration, and REST API.

- **Java**
  Java was used as the programming language for backend development.

- **Hazelcast**
  Hazelcast was utilized as an in-memory data grid to store the structures of *Camel* integrations.

- **Swagger**
  Swagger offered a user-friendly interface for documenting the developed API endpoints.

- **React with TypeScript**
  The front end was architected using *React*, enhanced by *TypeScript*. This combination ensured the development of reliable and maintainable UI components, creating extensible visualization.

- **Figma**
  As a design tool, *Figma* played a critical role in the early stages of development by enabling the visualization of components. This facilitated early discussions on functionality and expedited the development process by providing explicit design references.

- **Material-UI (MUI)**
  Integrated with *React*, MUI accelerated the UI development process, providing a uniform look and feel across the platform with its comprehensive suite of pre-designed components.

- **Storybook**
  Used as a development environment tool, *Storybook* streamlined the creation and testing of UI components, allowing for isolated development and mocking of responses without needing a back end.

- **Camel Karavan**
  Camel Karavan was an inspiration for designing the visualization for custom *Camel* integrations.

5.2 Data Flow Visualization for Generic Framework

This section is dedicated to the implementation of data flow visualization for the *Generic* framework within the *ON Semiconductor Middleware* platform. It outlines the complete, straightforward process that began with an in-depth analysis of the *Generic* integrations codebase and its integration with the more extensive middleware system. The section will cover the journey from building the underlying structure to creating a user-friendly visualization tool for *Generic* integrations.

5.2.1 Architectural overview

The flowchart serves as an architectural blueprint for the data flow visualization within the *Generic* framework. It briefly illustrates the sequence from the existing integration framework to the user interface where data flow visualization occurs.

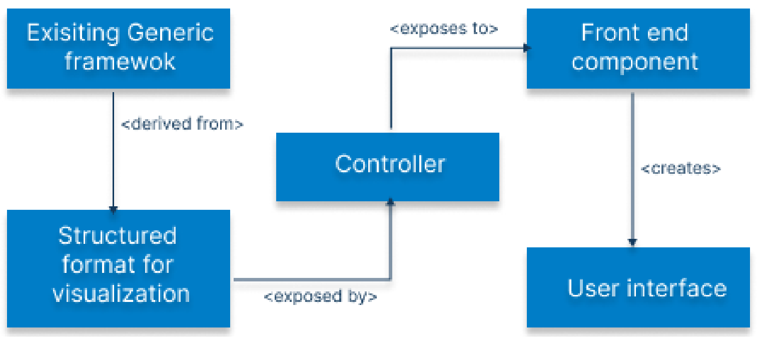


Figure 5.1. Architectural Overview for Generic Data Flow Visualization

The process starts with the existing *Generic* framework, which forms the bedrock of the current integrations. From this foundation, a structured format for visualization is derived, encapsulating the essential information in a format conducive to visual interpretation. The subsequent stage involves exposing this structured format through a RESTful controller, which acts as a channel between the backend logic and the frontend presentation layer.

The frontend component, which interacts with the controller, creates the visual representation of the data flows as part of the user interface. This user interface materializes the conceptual data structures into tangible, user-friendly visual components, culminating in an intuitive and accessible depiction of the integration flows.

5.2.2 Backend Model

The backend model for data flow visualization within the *Generic* framework is a part of the *Dashboard* component of the *Onsemi Middleware* platform. The development began with a simple model focused on a single factory responsible for generating the visual flow components, and all varieties of beans were directly wired into a *Dashboard*'s configuration. As the implementation progressed, the complexity increased to accommodate requirements for a more structured and extensible model. This led to the final architecture depicted in the figure 5.2.

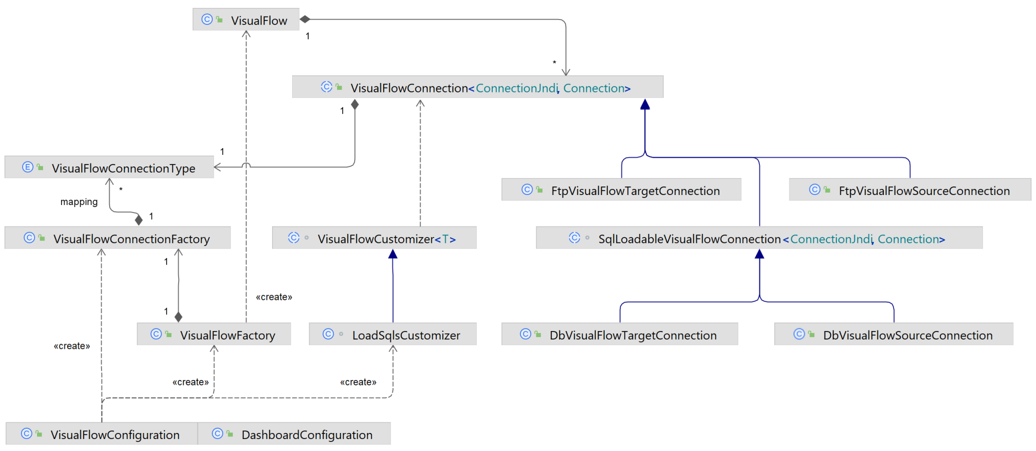

VisualFlow
VisualFlowConnection<ConnectionJndi, Connection>
VisualFlowConnectionType
mapping
VisualFlowConnectionFactory
VisualFlowCustomizer<T>
FtpVisualFlowTargetConnection
FtpVisualFlowSourceConnection
SqlLoadableVisualFlowConnection<ConnectionJndi, Connection>
VisualFlowFactory
LoadSqlsCustomizer
DbVisualFlowTargetConnection
DbVisualFlowSourceConnection
«create»
VisualFlowConfiguration
DashboardConfiguration

Figure 5.2. Generic Backend Model Architecture

- **VisualFlow**

  The `VisualFlow` component serves as the core of the *Generic* framework's visualization process, and it is ultimately exposed via the controller for frontend processing.

  This component is constructed with lists of `VisualFlowConnection` instances, which collectively embody the sources and targets of the data flow. These connections are pivotal in mapping the data journey as it moves through the integration process.

  Furthermore, the `VisualFlow` component also encapsulates delivery actions. The *Delivery Engine*, a distinct component of the *Onsemi Middleware* platform, specifically handles these actions.

- **VisualFlowConnection**

  The `VisualFlowConnection` is defined with *Java* generics to provide flexibility and type safety. Its abstract nature allows it to be concretely implemented by various specific connection types tailored to different data sources and targets. Any class extending `VisualFlowConnection` must specify types for `ConnectionJndi` and `Connection`, which are subclasses of the main beans that define a connection to another system. This design ensures support for the various connections and reduces the casting need.

  The connection can be in the form of `FtpVisualFlowTargetConnection`, `FtpVisualFlowSourceConnection`, or `SqlLoadableVisualFlowConnection` represented by `DbVisualFlowTargetConnection` or `DbVisualFlowSourceConnection`.

  Each specific connection class is further configured with property values predefined in the *Generic* framework, fitting the requirements of the given integration. To enable this, a custom spring annotation with an annotation processor has been created to streamline property injection.

  Additionally, the `SqlLoadableVisualFlowConnection` offers adaptability and the ability to accommodate customization through `VisualFlowCustomizer`.

- **VisualFlowCustomizer**

  The `VisualFlowCustomizer` is a *Generic* class that injects customizable behavior into `VisualFlowConnection` objects. It provides a framework for extending and modifying the functionality of these connections, tailoring them to the specific needs.

  A concrete implementation of the `VisualFlowCustomizer` is the `LoadSqlsCustomizer`, which enriches `VisualFlowConnection` objects with the capability to load SQL statements.

Using the customizer pattern in this context is crucial for maintaining a clean separation of concerns. It allows for the core `VisualFlowConnection` logic to remain uncluttered while still providing the flexibility to introduce new behaviors or modify existing ones without altering the underlying structure of the connections themselves. This pattern ensures that the system is both extensible and adaptable to evolving requirements.

- **VisualFlowConnectionFactory**
  The `VisualFlowConnectionFactory` is an integral part of the backend model, acting as the manufacturing hub for the different `VisualFlowConnection` instances. It utilizes a mapping system to create specific connection types based on the requirements of the visual flow. The factory maintains a mapping structure (depicted below) using generics, which allows the factory to handle many connection types and their respective *Java* class definitions flexibly.

```
Map<VisualFlowConnectionType, Map<Class<? extends
ConnectionJndiBean>, Class<? extends VisualFlowConnection>>>
```

  During the application startup, the post-process bean definition registry initializes this mapping, ensuring that all necessary connections are dynamically created. Following this initialization, the registration of these bean definitions follows. Thus, when the visualization process requires a particular type of data connection, the `VisualFlowConnectionFactory` can promptly produce the required instances.

- **VisualFlowConnectionType**
  The `VisualFlowConnectionType` is an enumeration within the visualization architecture that categorizes the types of connections into distinct source and target types.

- **VisualFlowFactory**
  The `VisualFlowFactory`, as the primary factory, merges all the necessary elements comprising a complete `VisualFlow` component.

  It leverages the `VisualFlowConnectionFactory` to procure the correctly typed `VisualFlowConnection` instances. It contains embedded logic that determines the specific connections needed for any flow based on the context and configuration.

  In addition to managing connections, the `VisualFlowFactory` is also responsible for incorporating the delivery actions into the `VisualFlow`. The *Delivery Engine*, the platform's component, processes these actions as a part of an integration.

- **VisualFlowConfiguration**
  The `VisualFlowConfiguration` is the spring configuration class within the platform's *Dashboard* component, producing the instantiation and configuration of essential elements in the visualization architecture.

  This configuration class is responsible for setting up the `VisualFlowFactory`, `VisualFlowConnectionFactory`, and the `LoadSqlsCustomizer`.

  Once defined, `VisualFlowConfiguration` is imported into the main *Dashboard* configuration of the *Onsemi Middleware* platform. This integration is pivotal as it links the visualization configuration directly with the *Dashboard*, thereby enabling the *Dashboard* to leverage the full suite of visualization tools for representing data flows.

5.2.3 Frontend Model

The iterative development of the frontend visualization for the *Generic* framework was focused on efficiency and usability.Using this iterative design approach brings benefits such as rapid improvements, informed decisions, and continuous user feedback, which leads to increased effectiveness.[3]

The initial step was to design a comprehensive prototype using *Figma*. This step was crucial as it laid the groundwork for the visual aspects of the application, providing a clear blueprint that guided the subsequent coding phase. Finalizing the design elements upfront ensured a more streamlined and focused development process within the *React* framework using *TypeScript*.[2]

The iterative process cycled through the design, development, testing, evaluation, and refactoring phases. This methodical approach facilitated the continuous improvement of the visualization tool, ensuring that each iteration brought it closer to the end goal of a user-friendly and effective UI. The testing phase involved consultations with fellow developers to gather feedback and ensure technical feasibility. As the interface matured, brief user testing was incorporated, providing valuable insights into the user experience and driving further refinements.

Architecturally, the frontend module was composed of modular *React* components that reflected the structure and design of the *Figma* prototype. The emphasis was on creating reusable components that could be efficiently organized and repurposed across the platform, promoting consistency and reducing redundancy.

The list below summarizes the iterative development process in three phases, capturing the main changes throughout the project. The following sections further describe these phases. They outline the frontend model's evolution, focusing on its development rather than delving into the functionalities in detail. These details are described in the final model in the following thesis chapter, which focuses on work results. Each phase has its model, depicted in the following figures 5.3, 5.4 and 5.5.

1. **Initial Prototyping**

   - **Objective:** Create an initial prototype to brainstorm and gather feedback on the visualization's appearance and functionality.
   - **Outcomes:**
     Established the overall structure of the visualization prototype.
     Discussions and received feedback for further refinement.

2. **Refinement, Codebase and Testing**

   - **Objective:** Enhance the visualization prototype, implement a structured frontend codebase, and gather feedback from users and developers.
   - **Outcomes:**
     Developed functional components in Storybook for the visualization.
     Conducted testing sessions with users and developers to gather feedback on the prototype.
     Initiated in-depth discussions based on the feedback received for further refinement.

3. **Finalization and Enhancements**

   - **Objective:** Finalize the visualization prototype based on feedback and incorporate additional enhancements to ensure usability and effectiveness.
   - **Outcomes:**
     Finalized the frontend visualization prototype based on feedback from users and developers.
     Additional enhancements were incorporated to improve usability and effectiveness.
     Implemented the finalized frontend visualization in an applicable form for integration within the platform.

- **Phase I - Initial Prototyping**
  The initial prototype presented a simplistic yet functional structure, offering a foundational layout for data flow visualization. This prototype was pivotal for brainstorming sessions and dialogues to collect insights on the eventual design and requirements for the visualization tool. This prototype was not meant for the development stage. It went straight to the testing stage, where discussions were held among developers of the *Onsemi Middleware* platform to gather feedback and determine essential features.

  The feedback indicated that while the foundational structure was adequate, the interaction method needed refinement. The consensus suggested displaying just the core information with an option to show additional information by some button, enhancing user engagement and information accessibility. Moreover, the prototype was revised to include visual representations for database connections, recognizing their distinct nature, especially the presence of SQL statements. This adjustment aimed to ensure that the visualization prototype caters to the different characteristics of each connection type. Additionally, it brought to attention the significance of the delivery action performed by the *Delivery Engine* within the data flow visualization. It was suggested that this element should not be omitted to ensure a comprehensive representation of the integration process. However, it was collectively decided to prioritize the development of a functional and streamlined prototype first.

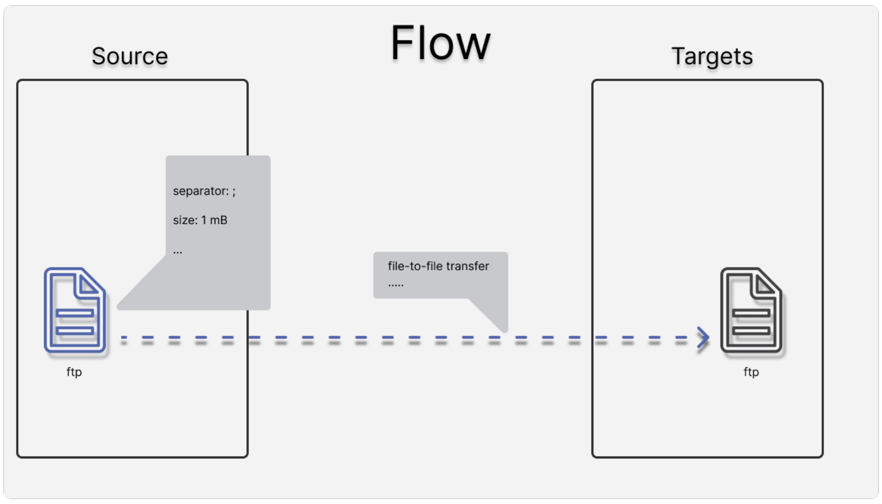


Figure 5.3. Figma prototype for Generic, phase I

Phase II - Refinement, Codebase and Testing

This version introduced a refined design to enhance visual appeal and functionality. The new prototype featured an on-click card next to each connection icon, summarizing the essential information with a detailed modal window for additional data. For database connections, the prototype showcased a specialized card listing all SQL statements, with the option to expand and view each in detail with syntax highlighting.

This enriched prototype laid the groundwork for the frontend model structure, which was developed using *Storybook* for efficiency. Data obtained from the backend controller were used as mock for visualization purposes within the *Storybook* environment.

The testing phase began with feedback from developers, who suggested eliminating the boxes around sources and targets for a cleaner interface. They recommended integrating the feature as a new tab within the interface section of the *Dashboard*, making the close button redundant. Furthermore, it was advised that editing features should be visibly deactivated or omitted, as this functionality was beyond the scope of the visualization tool and reserved instead for potential future development.

Two independent users also tested the visualization. The aim was to capture the users' natural interaction with the module and their independent input on the visualization's features without prior guidelines or instructions.

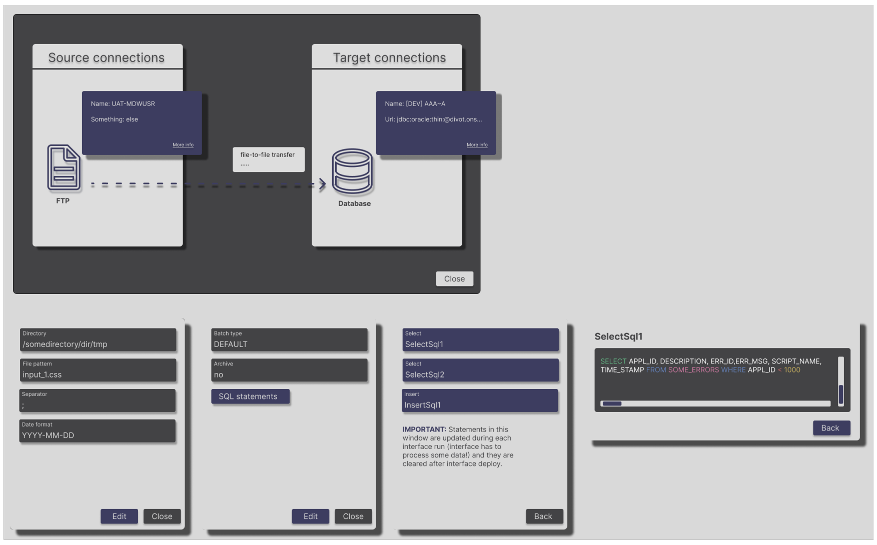
Taking an impulsive approach, the first participant navigated quickly through the various components. They expressed a positive view of the visualization, appreciating its novelty and the refreshing aspect it added to the platform. When prompted to locate the information they commonly sought, the participant could find the details important to them, such as the directory for the FTP and the SQL statement for the database. They showed interest in understanding other information presented in the visualization, but these eventually did not serve him any use.

The second participant, initially uncertain without specific instructions, asked about the intended actions to be taken. After being encouraged to explore the interface freely, it was observed that the user's cursor did not change upon hovering over interactive elements, which was noted for further enhancement. The participant appeared to be perplexed by the information displayed and did not engage with it meaningfully. Moreover, one of their integrations hinged on delivery actions, rendering the current visualization inadequate for their needs. This highlighted the importance of representing delivery actions within the visualization, which was vital for users with similar integration setups.

The evaluation of this user testing yielded some actionable insights, notably the necessity of cursor indication changes on hover to enhance user interaction cues. However, the more significant realization was the need to reassess how detail is displayed within the visualization.

The feedback was gathered from a limited and specific group of users, suggesting that extended user testing might be time-consuming and potentially contentious due to the diversity of the user base's needs. Some users with a technical orientation demand detailed insights, while others prefer a straightforward overview that simply maps the data flow from source to target by names alone. This note is not merely speculative but was corroborated by insights from the integration Architect of the *Onsemi Middleware* platform.

This distinction among users' expectations has implications for designing a visualization tool. It implies that the future development of this tool must strike a balance between offering depth of detail and maintaining clarity for those who seek simplicity. Therefore, another discussion with developers was held based on the last model developed.



Name: UAT-MDWUSR
Something: else

Name: [DEV] AAA~A
Url: jdbc:oracle:thin:@divot.ons...

file-to-file transfer
.....

Directory: /somedirectory/dir/tmp
File pattern: input_1.css
Separator: ;
Date format: YYYY-MM-DD

Batch type: DEFAULT
Archive: no

Select: SelectSql1
Select: SelectSql2
Insert: InsertSql1

IMPORTANT: Statements in this window are updated during each interface run (interface has to process some data!) and they are cleared after interface deploy.

SelectSql1

```
SELECT APPL_ID, DESCRIPTION, ERR_ID,ERR_MSG, SCRIPT_NAME,
TIME_STAMP FROM SOME_ERRORS WHERE APPL_ID < 1000
```

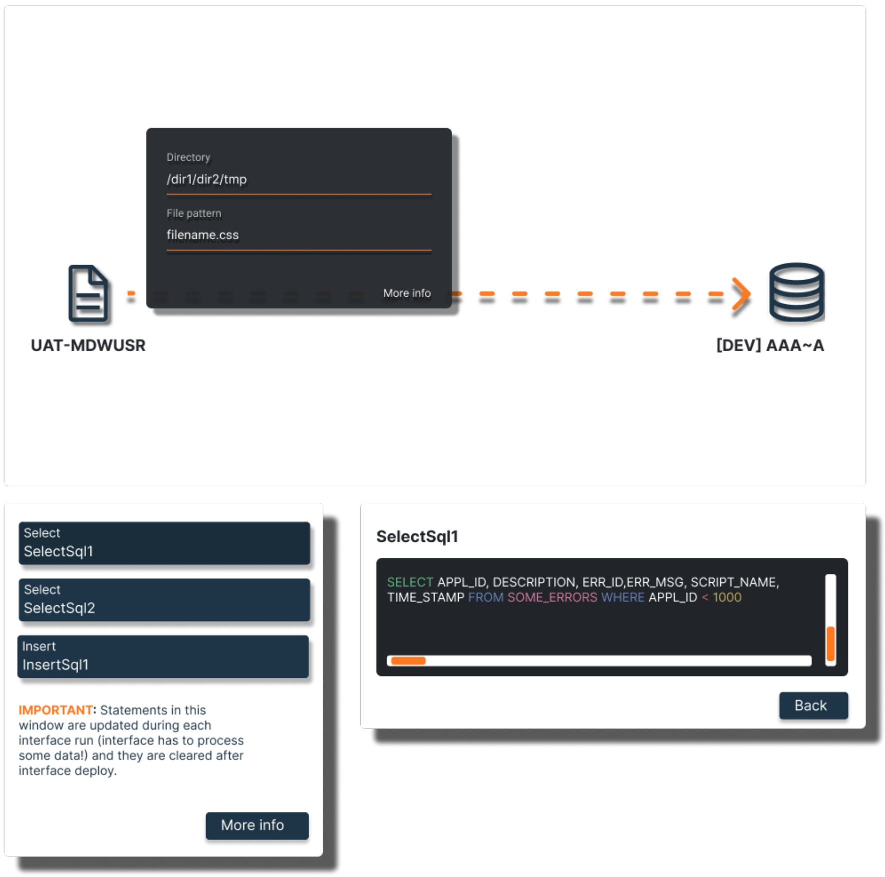
Figure 5.4. Figma prototype for Generic, phase II

- **Phase III - Finalization and Enhancements**
The last prototype marks the final form of the visualization. This prototype was additionally enhanced with minor adjustments and similar components. These enhancements did not necessitate the creation of a new Figma prototype and were included directly in the implementation. Additionally, the whole color scheme was shifted to fit the company's scheme.

An in-depth evaluation of its predecessor formed the design of this model. Discussions with developers indicated that navigating the previous model required excessive clicks to access key information. A new concept for displaying additional information was proposed. This concept involved introducing forms linked to specific connections within the flow. These forms are designed to present detailed information about the connections and are formatted to accommodate future features, enabling users to edit this information directly.

The development stage of this prototype introduces new changes. Names of connections are visibly displayed beneath their icons, offering an immediate understanding of the flow's structure at a glance. The FTP connection has been optimized to reveal two crucial pieces of information with a single click. Alternately, clicking on a database connection now triggers a modal with SQL statements right ahead. Both these windows offer the option to display newly designed forms tailored for that specific connection. Example of such form is depicted for database connection in figure 5.6. The model newly supports the delivery actions that mirror the approach of the FTP connection component. They present essential information upfront and provide a customized form for each action. For users who desire deeper information, the sufficient properties of *Generic* integration are now organized in a separate card.

These modifications have culminated in the final product for the *Generic* framework. The results and implications of this prototype are further discussed and evaluated in the subsequent chapter 6.


Directory
/dir1/dir2/tmp
File pattern
filename.css

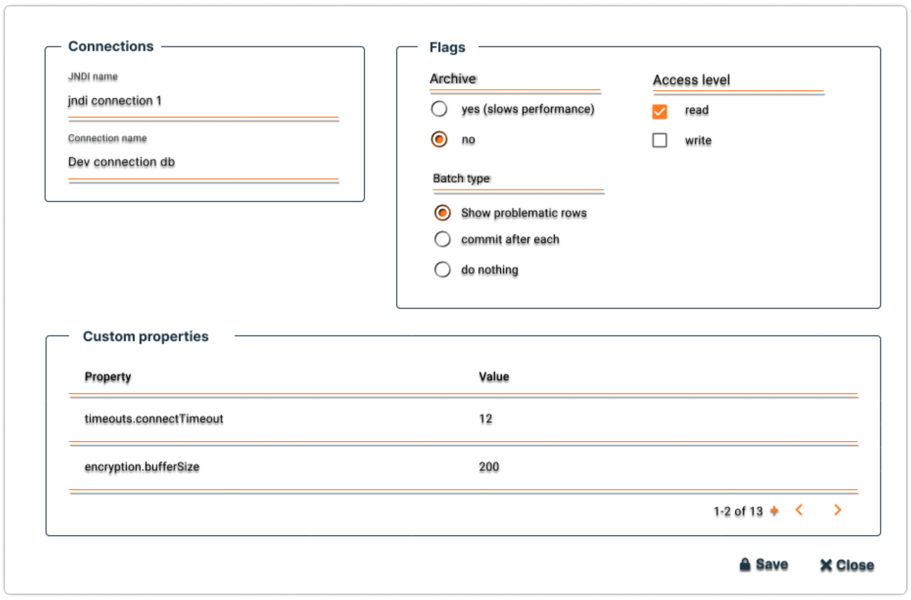
Select
SelectSql1
Select
SelectSql2
Insert
InsertSql1

IMPORTANT: Statements in this window are updated during each interface run (interface has to process some data!) and they are cleared after interface deploy.

SelectSql1

```
SELECT APPL_ID, DESCRIPTION, ERR_ID,ERR_MSG, SCRIPT_NAME,
TIME_STAMP FROM SOME_ERRORS WHERE APPL_ID < 1000
```

Figure 5.5. Figma prototype for Generic, phase III


Connections

JNDI name: jndi connection 1

Connection name: Dev connection db

Flags

Archive: ( ) yes (slows performance) [✓ no]

Access level: [✓ read] write

Batch type: [✓ Show problematic rows] ( ) commit after each ( ) do nothing

Custom properties

Property	Value
timeouts.connectTimeout	12
encryption.bufferSize	200

1-2 of 13

Figure 5.6. Figma form prototype for Generic, phase III

5.3 Data Flow Visualization for Camel Framework

The development process for the structuralization and visualization of the *Camel* framework integrations presented a multifaceted challenge, markedly distinct from the more straightforward path navigated for the *Generic* framework. Due to *Camel* integrations' dynamic and heterogeneous nature, establishing a robust method for capturing their intricate structure was critical yet proved to be a complex undertaking. This thesis section details the complicated process of developing a system capable of interpreting and visualizing the various *Camel* integrations. The subsequent creation of a corresponding frontend model also entailed navigating many possibilities presented by *Camel's* comprehensive set of components.

The entirety of this development process, replete with its inherent challenges and innovative solutions, is documented in the subsequent discourse, encapsulating the essence of rendering custom *Camel* integrations into a visually coherent and functional form.

5.3.1 Backend Model

When embarking on the development of the backend model for the *Camel* framework, the process commenced with an exploration of static code analysis. This technique was applied to the codebase of several *Camel* integration implementations to determine the feasibility of constructing a structured model. The tool for trying out the static analysis was the *Java* parser provided by `com.github.javaparser`. Initially, the results were auspicious when applied to individual integration instances, indicating the potential for extracting a coherent structure.

However, as the static analysis was extended across various implementations, it became evident that crafting a universal parser capable of accommodating all the potential variations in implementation would be a very complex task that could still be incapable of handling all the possible implementations. This realization necessitated the pursuit of an alternative path.

Search for alternatives unearthed the possibility of leveraging the legacy native XML representations of *Camel* routes. Initially, these XML representations were crafted to define *Camel* routes, not to extract information from running *Camel* code. Nonetheless, they offered an unexplored avenue for structuring the *Camel* integrations at a foundational level.

Given that these XML-defined routes encompass the entirety of the data flows within *Camel* integrations, the essence of what this thesis aims to visualize can be satisfied.

With this new approach, a backend architecture was developed to utilize the legacy XML. This architecture aimed to interpret the XML route definitions into a structured model that could serve as the backbone for the desired data flow visualization within the *Camel* framework. This approach promised a less demanding and more reliable path to achieving the thesis's goals and set the stage for the following complex process.

The architecture for the backend model within the platform and the logic behind it are depicted in figure 5.7 and described in the text below.

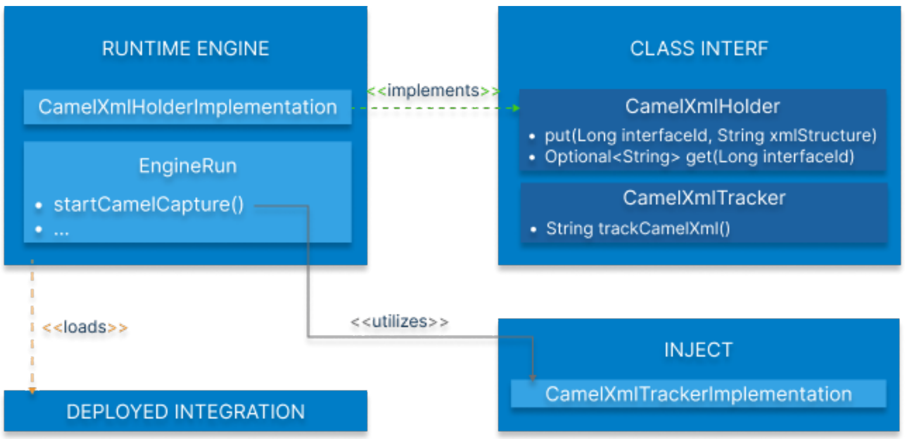


Figure 5.7. Backend model architecture for capturing Camel integration structure

Architecture Description

As discussed in previous section 4.1.1, the *Runtime Engine* is one of the platform's primary applications. Additionally, the platform incorporates custom libraries, namely the Class Interf library and the Inject library, which are part of the process behind capturing the *Camel* structure.

- **Class Interf Library**
  The Class Interf library prescribes the *Java* interfaces. This library defines two primary interfaces for *Camel* structure capture: the `CamelXmlTracker` and the `CamelXmlHolder`. The `CamelXmlTracker` interface encapsulates the logic to extract the XML structure from running integrations. The `CamelXmlHolder` interface manages the storage of the tracked XMLs and ensures their persistence and accessibility.

- **The Inject Library**
  The Inject library provides an implementation of the logic required for capturing the XML structure of running *Camel* integrations. This implementation is represented by Spring Bean, constructed based on a newly created configuration with Spring conditionals. These conditionals are the presence of integrationContext and the identification of the integration as a *Camel* integration, ensuring that the bean is wired at the appropriate time and only when necessary.

  To prevent errors during various integration runs, all additional *Camel* dependencies required for this implementation should used with the option of scope provided. This approach avoids direct wiring of *Camel* to the *Runtime Engine*, which could lead to errors during other integration runs. By using the provided scope, these dependencies are available only at compile time. Subsequently, these dependencies should be provided within the *Deployed integration* that the *Runtime Engine* executes.

- **Runtime Engine**
  The *Runtime Engine* houses the implementation of the `CamelXmlHolder` class, with the logic for storing the extracted structure. Additionally, it controls the entire execution of integrations. This management is handled within the EngineRun class, where the new feature of capturing the structure is triggered. The EngineRun class utilizes the Tracker implementation from the Inject library to facilitate this functionality.

- **Deployed Integration**
  The *Deployed integration* represents a container that holds the integration along with all its dependencies. This container is loaded by the *Runtime Engine*, enabling the execution of the integration within the platform.

Even though the architecture and strategy were precisely designed, numerous challenges emerged throughout the development process. The notable challenges encountered and their respective resolutions are outlined below.

- **Spring contexts**
  The initial challenge arose due to the absence of the `CamelXmlHolder` bean within the integration spring context. In the *Onsemi Middleware's* platform architecture, integrations operate within their own integration spring context. However, accessing the `CamelXmlHolder` bean within this context must be retrieved from the application context.

  Furthermore, since the `CamelXmlHolder` bean creates a *Hazelcast* map to store XMLs, it needs to be instantiated as a singleton within the application context.

  To address this challenge and ensure the class is accessible within the integration spring context, a solution in the form of custom `@DefaultBeans` annotation was applied. The `@DefaultBeans` annotation acts as a plugin system, allowing for the creation of bean definitions for each `DefaultBean` passed from the parent context. Through this approach, the `CamelXmlHolder` bean was instantiated within the application context and made accessible within the integration spring context, meeting the requirements for singleton instantiation and context accessibility.

- **Dependencies**
  The next challenge emerged from Maven dependencies. As previously mentioned in the architecture, dependencies necessary for extracting routes in XML format were utilized within the Inject library with a provided scope. However, this approach led to an exception, likely due to the *Runtime Engine's* inability to correctly locate the appropriate provided class within the *Deployed integration* container.

  This issue likely arised from class identification based not only on the class itself but also on the classLoader through which it was loaded. Consequently, the *Runtime Engine* failed to identify the correct class when later provided by the *Deployed integration* container.

  The implementation of the `CamelXmlTracker` was reworked to utilize *Java* reflection to resolve this issue. *Java* reflection is an API used to modify the behavior of methods, classes, and interfaces at runtime. While reflection is not considered a best practice, it serves as a viable solution for addressing dependency issues within custom platforms.

- **Effecting integration run**
  The next challenge arose during the integration run process. Capturing the routes necessitated performing this capture during the interface run. The routes are stored in a field called routes in the integration's *Camel* context, which is populated at a specific time. To avoid polling this field, another field containing the route definitions, based on which the routes are dynamically created during the *Camel* integration run, was utilized instead.

  However, extracting information from the route definitions resulted in the definitions deleting themselves after being used, leading to the failure of the entire integration run.

This issue was resolved by implementing a simple solution, creating a deep copy of the route definitions and working with the copy instead. This ensured that the original definitions remained intact and accessible throughout the integration run process.

- **DSL usage**
  Another challenge appeared due to insufficient information in the extracted XML structure. This issue arised from the use of the *Java DSL* for *Camel* integration. Custom usage of this DSL allows for the introduction of new logic, such as creating custom classes to store endpoints.

  However, this custom approach led to a problem during the structure extraction: the parser could not read these custom classes, which excluded vital information about the integration.

  Additional parsing for such endpoints was introduced to enhance the exported XML. This enhancement ensured that crucial information about the endpoints was sufficiently captured and included in the extracted XML structure, thereby improving the overall completeness.

- **Usage of multiple Camel versions**
  While testing the capture process, a new challenge arose concerning using different versions of *Camel*. Within the platform, multiple versions of *Camel* are employed for integration purposes, and these versions may have variations in their APIs used for extracting the XML structure.

  Therefore, logic was implemented to determine the appropriate version and the appropriate use of reflections to select the corresponding API for valid extraction. This approach ensured compatibility with different *Camel* versions across various integrations within the platform.

  Moreover, this challenge highlighted the importance of injecting correct dependencies for *Camel* integrations within the platform. As the APIs varied between different dependencies, ensuring that the correct dependencies were added to all existing interfaces in the appropriate versions became necessary. This task could be managed through an appropriate database update of the existing integrations to ensure compatibility across the platform.

- **Services**
  During testing, another challenge emerged related to integrations representing services. By nature, the services start and do not end until they are stopped. This forces the capture to start right before the service starts.

  Therefore, modifications were made to the `CamelXmlTracker` to support running in its thread. With this enhancement, the Tracker was configured to continuously poll for the *Camel* context of the interface to be loaded. Once the context was detected, the Tracker would execute the capture process, ensuring that the integration structure was captured appropriately, even for services.

5.3.2 Frontend Model

This section describes the development of the frontend part, which commenced after the backend model successfully captured the integration structure for various integrations. No *Figma* prototype was necessary as the desired visualization structure was based on the open-source *Camel Karavan*. Although *Camel Karavan* did not contain the same components as those in the captured XML structure, its overall visualization structure served the same purpose and thus inspired the frontend development.

While there were multiple options for working with the XML structure obtained from the backend, such as creating a *React* component for each element and then assembling the visualization, the nesting structure of the XML suggested a more straightforward approach. Thus, the chosen method involved traversing the XML and dynamically building the components. A classic Depth-First Search (DFS) algorithm was chosen for this traversal.

To handle the diverse components efficiently, an optimal implementation led to a component generator. This generator dynamically creates components based on provided parameters, with the option for specialized behavior if needed. These unique behaviors include extracting additional information from the component's body, setting different styling for its child components, visually interconnecting with other components, or displaying extra information beyond just properties. For instance, the display of extra information involves making requests to the backend to parse and decode SQL statements or to resolve information about delivery actions. This dynamic approach ensures that the front end can adapt to various integration scenarios and can be easily extended by new components.

To effectively capture the entire flow of integration within *Camel*, a straightforward layout was chosen. This decision stems from the complexity of *Camel* integrations, often composed of multiple routes. Each route is positioned adjacent to one another, allowing for a clear visualization of their interconnections. Moreover, each route has its configuration. This configuration remains consistent throughout the routes within a specific integration. In the visualization, this configuration is extracted from the routes and positioned at the beginning on the left for logical purposes. The following chapter 6 provides further details for a comprehensive understanding of the visualization and its features.

From a technological perspective, *React* and *TypeScript*, along with *Storybook*, were once again leveraged, mirroring the approach used to visualize the *Generic* template. When selecting icons for components, careful consideration was given to their respective functions. Icons were sourced primarily from the *Karavan* project and *Google Fonts* page. The open-source *React X-arrows* library was employed for visual interconnections between components. In order to ensure consistency, some elements of the *Generic* visualization were reused.

5.3.3 Interconnection between the Models

This section delves into the interconnection between the frontend and backend modules, providing insight into how the entire visualization is created within the *Onsemi Middleware* platform. A diagram 5.8 is provided below to explain this connection.

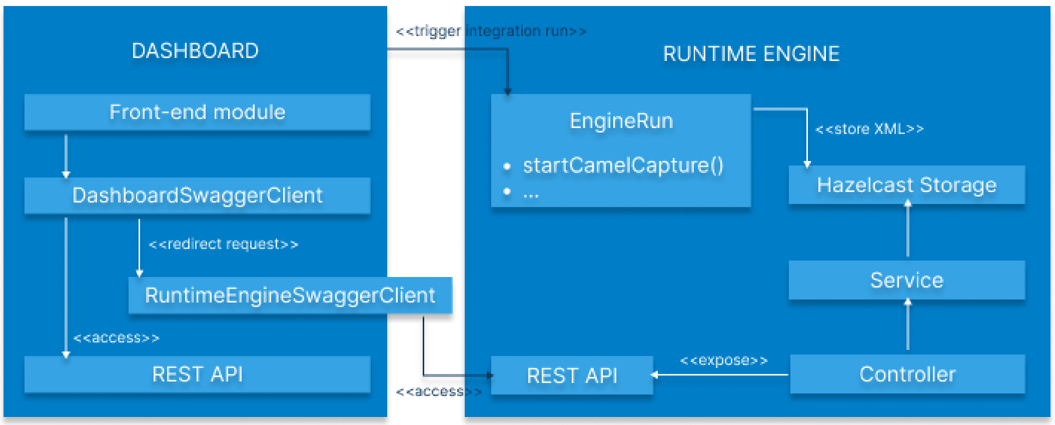


Figure 5.8. Overall Architecture of Capturing XML Structures for Camel integration

This diagram simplifies the depiction by focusing solely on the two main applications of the platform. Including libraries and additional applications would introduce unnecessary complexity for what it aims to describe.

The process begins with the *Dashboard* application triggering the execution of a specific integration on the *Runtime Engine*. The *Runtime Engine* excludes the entire execution process here, including the capture of the XML structure, as detailed in the previous backend model section 5.3.1. The captured structure is stored in a *Hazelcast* map and can be accessed through the service when the controller needs to access it. The controller exposes an REST API that is responsible for accessing this stored information.

Additionally, the *Dashboard* encapsulates the frontend module, which is responsible for the final visualization based on the obtained structure from the *Runtime Engine's* API. This interaction occurs through a redirected request facilitated by the `DashboardSwaggerClient` and passed to the `RuntimeEngineSwaggerClient`. The frontend module also utilizes the `DashboardSwaggerClient` to send additional requests to the backend to resolve more information.

To clarify, while the term *redirect request* is used, the actual process is more nuanced. Direct API requests from the *Dashboard's* frontend to other applications are not feasible due to browsers blocking cross-domain requests. Instead, clients are managed on the backend, replicating necessary endpoints through the application. In this case, the *Dashboard* has its own REST API for obtaining the *Camel* XML structure. During this request, the RuntimeEngineClient is employed to send a request to the *Runtime Engine*. Subsequently, the response is handled in the backend and passed as a response to the original request.

Chapter 6
Work Results and Evaluation

This chapter provides a comprehensive summary of the work results, offering insights into the final visualizations for the *Generic* and *Camel* frameworks. It focuses exclusively on presenting the final visualizations and their features, omitting the detailed results of the backend model, which are discussed in the Implementation chapter 5.

The chapter also incorporates feedback from platform developers and integration specialists through dialogue sessions to ensure a thorough evaluation of the contributions. Additionally, testing and reviews conducted by a select group of users further enrich the assessment of the delivered solutions.

6.1 Data Flow Visualization of the Generic Framework

This section provides an overview of the final visualization for the *Generic* framework, offering screenshots accompanied by brief descriptions to illustrate the key features and layout of the visualization.

6.1.1 Initial Screen

The initial screenshot 6.1 offers an overview of the entire integration flow, which is incorporated into the existing platform as a tab.. While dark information cards are not visible by default, users will typically encounter a similar interface where connection names are listed under corresponding icons, making it easy to identify each connection type, such as FTP or database connections. For delivery actions like FTP delivery, specific connection names are included, while icons for email and HTTP actions are self-explanatory.

In the provided example, the integration progresses from an FTP connection to a database, incorporating email file delivery and HTTP requests. This should be understood by the user at first glance.

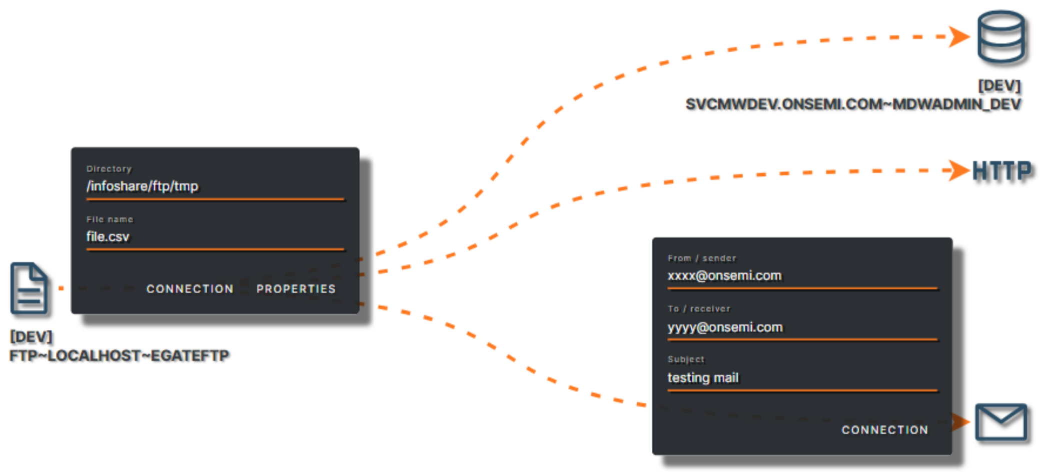

Directory
/infoshare/ftp/tmp
File name
file.csv
From / sender
xxxx@onsemi.com
To / receiver
yyyy@onsemi.com
Subject
testing mail

Figure 6.1. Example of Data Flow Visualization of Generic Framework

6.1.2 Key Infromation Card

The main information cards serve as key elements within the visualization, offering users quick access to essential details without navigating through multiple components. These cards are available for all delivery actions and FTP connections, providing users with relevant information at a glance. However, the database connection behaves differently, as it primarily lists SQL queries, which are considered the primary information for its component.

In the provided example 6.1, these cards are depicted as dark gray panels visible next to the FTP connection or email delivery icons. Upon clicking on the respective icon, users trigger the display of the corresponding card.

Each card is precisely designed to present users with the most sought-after information for that particular component. For FTP connections or deliveries, this includes details such as directory paths and file patterns, while for email deliveries, it includes sender details, recipient addresses, and email subjects. Similarly, for HTTP requests, the target URL is prominently displayed.

These cards offer a streamlined viewing experience by presenting crucial information directly on the visualization canvas, eliminating the need to search for key information within many details. The card can be closed only by clicking on the icon again, a deliberate design choice aimed at allowing users to view the entire flow with details opened from multiple connections.

Moreover, each card features additional buttons that allow users to access more detailed information if desired. The 'Properties' button enables users to view properties associated with the *Generic* framework configuration, while the 'Connection' button directs users to a comprehensive form listing all relevant connection information.

6.1.3 SQL Card

The SQL card, accessible for database connections, opens as a modal in the center of the screen upon clicking the respective component icon. This design choice was made to accommodate the more extensive content of the card without distracting from the overall flow visualization.

Within the card, all SQL statements performed on the present database are listed along with their names, allowing users to select and view details about specific statements by clicking on them. Additionally, the card incorporates short informational statements.

At the bottom of the card, three action buttons are provided. The first allows users to close the card, while the second provides access to the properties associated with the *Generic* framework configuration. The third option enables users to access a form containing all relevant details about the database connection.

An example of an SQL card is presented in figure 6.2.

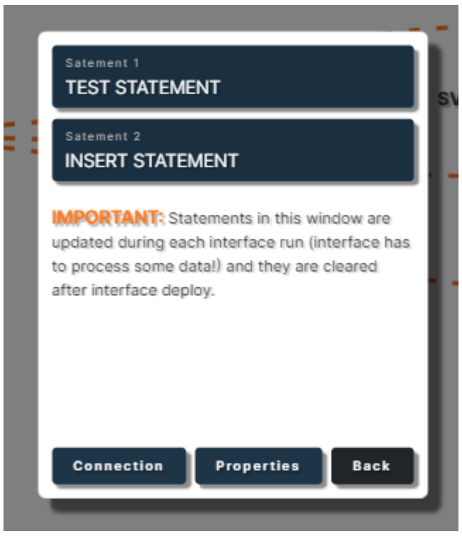
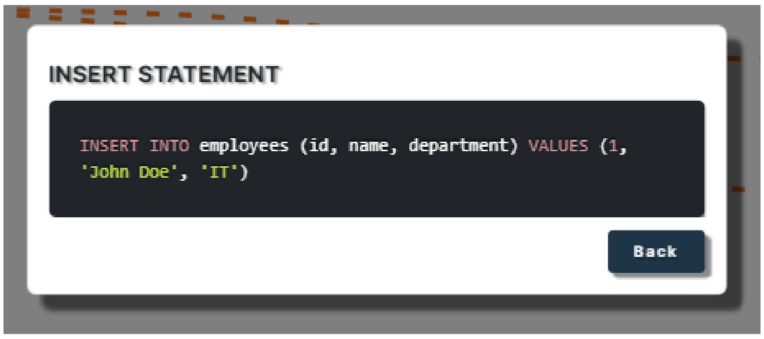

Satement 1
TEST STATEMENT
Satement 2
INSERT STATEMENT
IMPORTANT: Statements in this window are updated during each interface run (interface has to process some data!) and they are cleared after interface deploy.

Figure 6.2. Example of SQL Card Visualization

6.1.4 SQL Statement Card

The SQL Statement Card is opened after clicking on a specific SQL statement in the previous tab. It provides users with the exact SQL statement visualized with highlights to make the statement more readable.

An example is presented in figure 6.3.


INSERT STATEMENT

```
INSERT INTO employees (id, name, department) VALUES (1,
'John Doe', 'IT')
```

Figure 6.3. Example of SQLStatement Card Visualization

6.1.5 Connection Forms

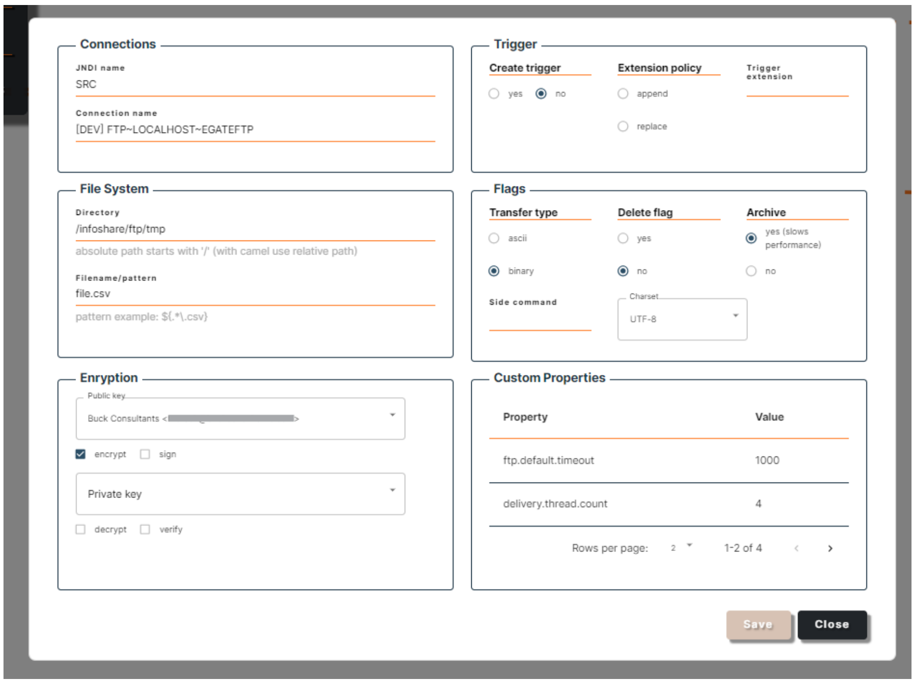
The custom connection forms, accessible by clicking the connection buttons associated with each connection and Delivery Action, provide users with a detailed overview of the respective connection properties. These modal windows also appear in the middle of the screen. While some components of the forms are shared among different connections, they have been reused.

An example of the FTP form is depicted below 6.4. The form design consolidates related information to enhance user orientation throughout the form. Each form encapsulates all the essential information about the connection, including details related to the file system, encryption, custom properties, and more. While the fields of all forms are not explicitly mentioned here, they will be included in the user manual in the form of the *Confluence* page.

It is important to note that these forms are developed solely for visualization purposes in the context of this thesis, and therefore, users cannot save the information they edit.

However, they have been designed in a way that allows for future expansion to enable setting connection details directly within the visualization.

As a result, the save button is disabled, and the other button closes the form.



Connections

JNDI name: SRC

Connection name: [DEV] FTP~LOCALHOST~EGATEFTP

Trigger

Create trigger	Extension policy	Trigger extension
yes [✓ no]	append / replace	

File System

Directory: /infoshare/ftp/tmp

absolute path starts with '/' (with camel use relative path)

Filename/pattern: file.csv

pattern example: ${.*\.csv}

Flags

Transfer type	Delete flag	Archive
ascii [✓ binary]	yes [✓ no]	[✓ yes (slows performance)] no

Side command:

Charset: UTF-8

Enryption

Public key: Buck Consultants <[redacted]>

[✓ encrypt] sign

Private key

decrypt verify

Custom Properties

Property	Value
ftp.default.timeout	1000
delivery.thread.count	4

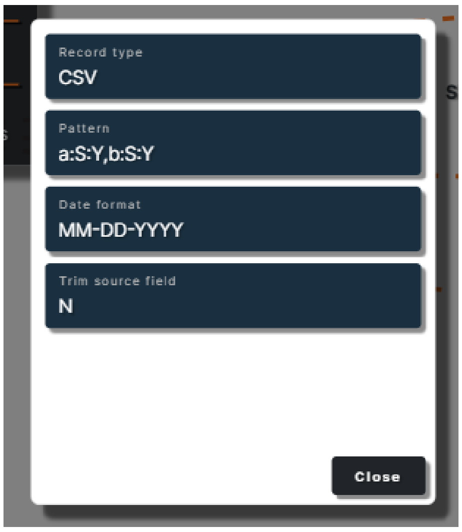
Rows per page: 2 1-2 of 4

Figure 6.4. Example of FTP Conenction Form

6.1.6 Properties Card

The properties card, as mentioned earlier, displays the properties related to the *Generic* framework configuration. This card is available for FTP and database connections. While it may not provide significant value to many users, it can be helpful for some.

An example of such a card for an FTP connection is shown below 6.5.



Record type: CSV

Pattern: a:S:Y,b:S:Y

Date format: MM-DD-YYYY

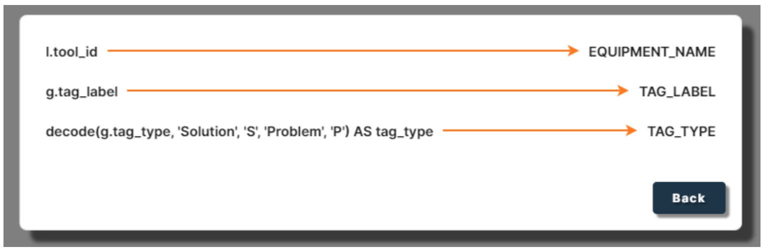
Trim source field: N

Figure 6.5. Example of Properties Card

6.1.7 Database to Database Mapping Card

In the case of integrating data from database to database, the visualization provides options to display mapping on columns based on the SQL statements. This card opens by clicking the head of the arrow.

An example of a mapping card is in the figure 6.6.



l.tool_id → EQUIPMENT_NAME
g.tag_label → TAG_LABEL
decode(g.tag_type, 'Solution', 'S', 'Problem', 'P') AS tag_type → TAG_TYPE

Figure 6.6. Example of Mapping Card

6.2 Data Flow Visualization of the Camel Framework

The core feature of the *Camel* framework visualization is the representation of the flow structure. Due to the diverse components in various integrations and the underlying complexity of the *Camel* framework, this description provides a concise overview. A synoptic integration example is depicted in figure 6.7 to highlight the key features, with additional examples available in the appendix.

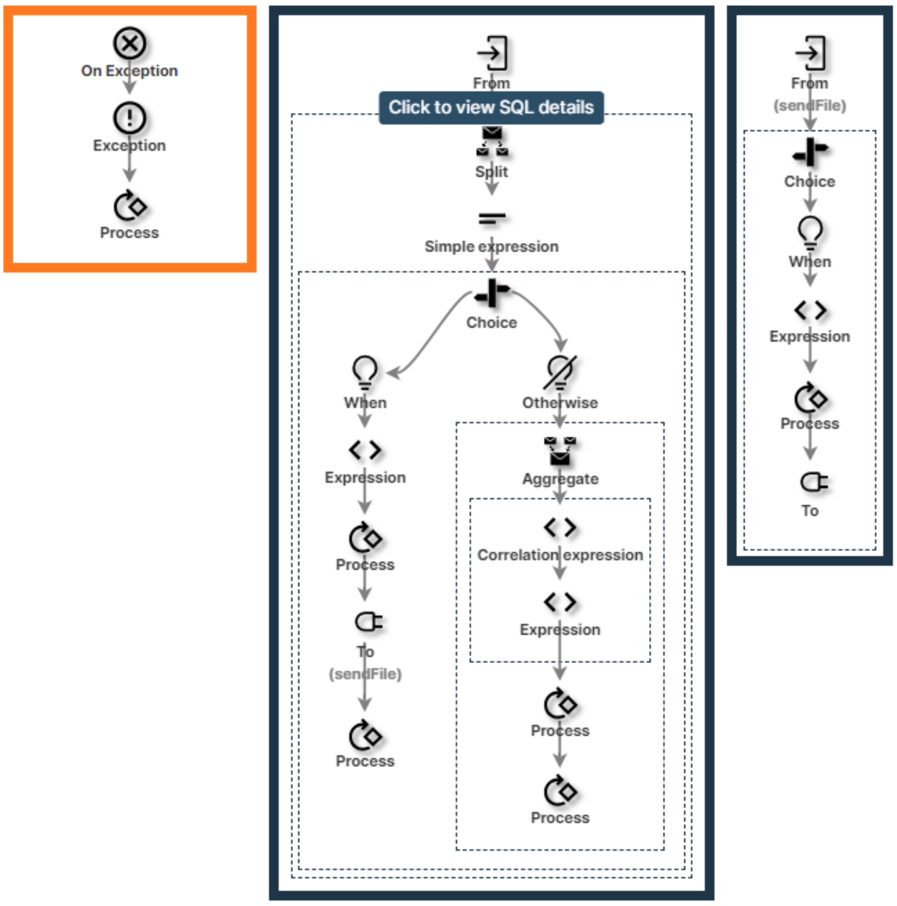


Figure 6.7. Example of Camel Integration Flow

6.2.1 Structure

The visualization primarily consists of routes framed in blue and optional configurations, commonly exception handling configurations, framed in orange at the beginning.

These routes are filled with *Camel* components that are further framed by their context. Each component can be hovered over to display a tooltip providing relevant information. This tooltip informs the user of the available actions upon clicking the component, such as showing properties, SQL details, delivery events, or indicating when no additional information is available. An example of such tooltip is depicted in the example 6.7.

The configurations are also formed from components, but they usually represent some sort of error handler. In the example provided, the configuration consists of an error handler, enabling users to examine the type of exception thrown. Another screenshot 6.8 shows configurations may consist of multiple handlers and include components like logs.

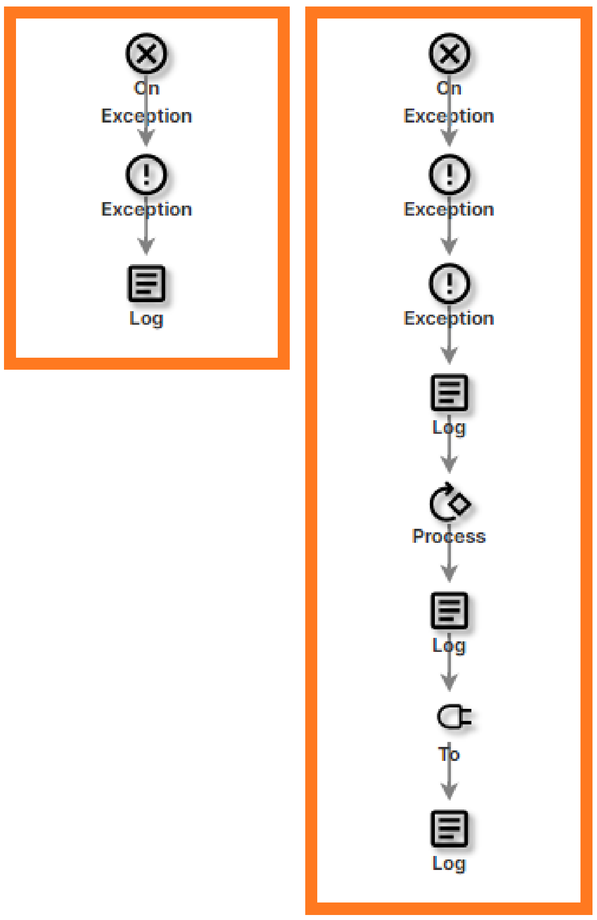
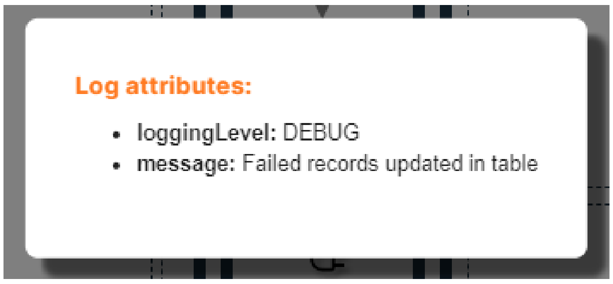


Figure 6.8. Example of Camel Integration Exception Handling Configuration

6.2.2 Properties

An example of how the properties display for components that allow this option is demonstrated in figure 6.9 on the mentioned log component from 6.8.



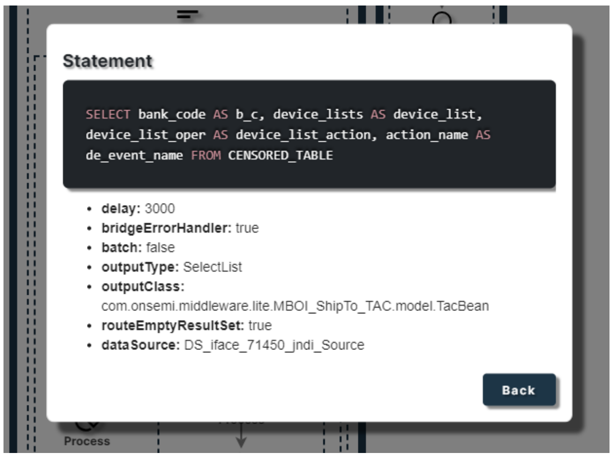
Log attributes:

- **loggingLevel:** DEBUG
- **message:** Failed records updated in table

Figure 6.9. Example of Properties Details within Camel Visualization

6.2.3 Database Endpoints

A window similar to that used in the *Generic* framework visualization is employed for displaying SQL details, with additional properties provided, as shown in figure 6.10.



Statement

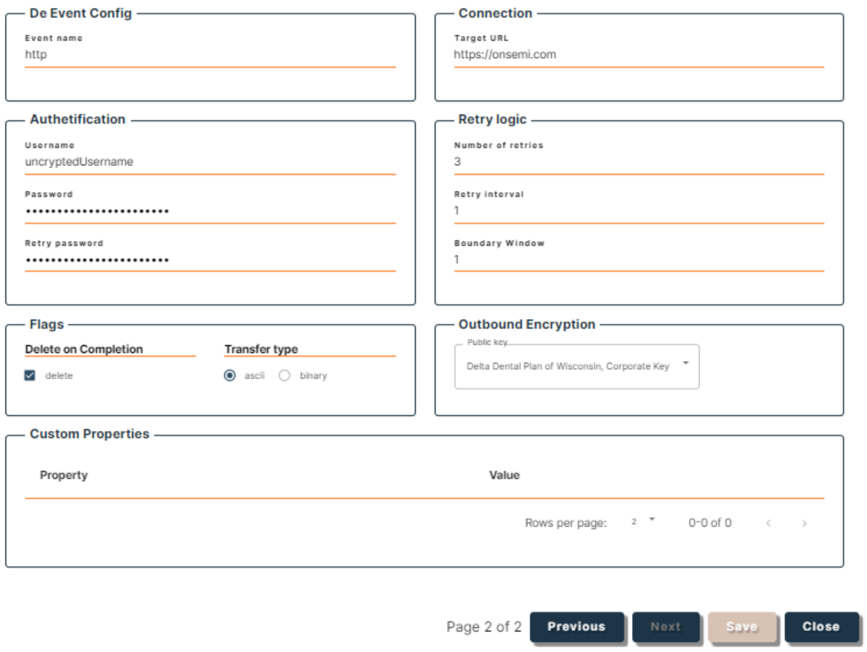
```
SELECT bank_code AS b_c, device_lists AS device_list,
device_list_oper AS device_list_action, action_name AS
de_event_name FROM CENSORED_TABLE
```

- **delay:** 3000
- **bridgeErrorHandler:** true
- **batch:** false
- **outputType:** SelectList
- **outputClass:** com.onsemi.middleware.lite.MBOI_ShipTo_TAC.model.TacBean
- **routeEmptyResultSet:** true
- **dataSource:** DS_iface_71450_jndi_Source

Figure 6.10. Example of SQL Card for Camel Visualization

6.2.4 Delivery Event Endpoints

Similarly, the option to display a delivery event closely resembles the functionality of displaying the FTP Connection form depicted in figure 6.4. However, in this case, the event may be associated with multiple delivery actions, offering users the option to navigate through the actions. Figure 6.11 showcases example of such form.



De Event Config

Event name	http

Connection

Target URL	https://onsemi.com

Authetification

Username	uncryptedUsername
Password	••••••••••••••••••••••
Retry password	••••••••••••••••••••••

Retry logic

Number of retries	3
Retry interval	1
Boundary Window	1

Flags

Delete on Completion: ✓ delete

Transfer type: [✓ ascii] binary

Outbound Encryption

Public key	Delta Dental Plan of Wisconsin, Corporate Key

Custom Properties

Property	Value

Rows per page: 2 0-0 of 0

Page 2 of 2

Figure 6.11. Example of Delvery Event Form for Camel Visualization

6.3 User Testing

The user testing phase was initiated to comprehensively evaluate the resulting visualization, assessing its strengths, weaknesses, and areas for improvement. This endeavor aimed not only to discern the contribution of the visualization but also to propose future developments or simple adjustments that could enhance the overall user experience.

6.3.1 Methodology

The chosen methodology for testing the *Generic* visualization and the *Camel* visualization varied slightly due to their distinct natures. However, despite these differences, each testing session followed a similar approach. Users were informed at the outset that they were not the subject of testing and that their capabilities of answering any question or completing a given task were not measured. They were encouraged to freely share their thoughts and opinions throughout the session, including any negative feedback, with the assurance that this feedback would not affect the thesis rating. This conversation aimed to ensure that users felt comfortable sharing their honest impressions.

Testing took place in the natural environment of the company, typically in a small meeting setting, with a sweet treat of appreciation offered as gratitude for their participation to maintain a relaxed atmosphere. This choice follows the commonly known recommendations for user testing, also mentioned in [42]. Each session lasted between 20 to 45 minutes, with testing for the *Camel* visualization often being longer. After each session, users were asked to summarize their thoughts and feelings about the visualization and ask any additional questions they wanted.

The chosen methodology aligns with the Informal Walkthrough method, emphasizing intuitive user interaction. This approach involves conducting test sessions within real-world usage contexts, incorporating task-free exploration periods and optional predefined scenarios, and it necessitates the availability of a functional prototype.[42]

6.3.2 Generic Visualization Testing

For the testing of the *Generic* visualization, five users of the platform were selected based on recommendations from the integration architect, who possesses extensive experience with such users. The users were chosen in this manner to align with the recommendations mentioned in [42], which suggest selecting real and possible users. Therefore, four real users and one possible user were selected. The [42] also mentions that about 80% of usability issues can be discovered with five users. Each user was presented with two visualizations of their custom integration to simulate their future usage. The testing approach involved allowing the users to freely explore the first integration without providing any prior information about the visualization. Their interactions and comments were observed and documented. In the second integration, users were given a basic description of the present components, similar to the one in the user manual, that will be accessible to all users. Users were assigned two simple tasks during this phase to locate specific information within the integration.

A brief summary of each user's session follows.

- **Participant I.**
  The participant began briefly contemplating their observations, mentioning their familiarity with the integration by name and their ability to discern the connections. This indicated a solid understanding on the part of the participant. They explored the integration more thoroughly by interacting with the icons and arrows. Although they encountered some readability issues, the participant attributed this to forgetting their glasses and speculated that using a laptop screen instead of their usual monitor might have contributed to the problem. This suggested a need to verify the clarity of the visualization across various resolutions.

  The participant did not focus much on the property card but dedicated time to studying the information within the forms. They shared common challenges experienced when using such integrations and proposed enhancements to address these issues within the visualization. One suggestion was to incorporate an existence check for the directory field in the FTP connection (as the participant owned only such FTP integrations) because the integration fails when the specified directory does not exist, requiring them to search for this information within the logs, which they found inconvenient. Another proposal was to introduce a test button to identify problems with a connection in case of integration failure.

  During the second part of the testing, the participant swiftly located the requested information despite clicking through some cards multiple times.

  In their final feedback, the participant expressed appreciation for the visualization, noting its value in immediately displaying connections without the need to navigate through tabs as usual, which sometimes loaded slowly. They highlighted the potential time-saving aspect of this feature. They reiterated the usefulness of the option to test the connection, emphasizing its benefits for themselves and their team.

- **Participant II.**
  The second participant initially struggled with being given the freedom to explore and immediately sought guidance on what to do. They were encouraged to interact with various elements, read the provided information, etc. The participant noted some design preferences throughout the exploration, such as preferring a classic **X** instead of a close button. However, they did not express many opinions and instead focused more on recalling and understanding the integration, inquiring about certain

aspects of the integration logic. They experimented with text copying and right-clicks and appreciated the colored visualization of the SQL statements.

During the task phase, they successfully located the intended information without encountering any difficulties.

This participant also displayed curiosity about the *Camel* visualization, indicating a willingness to learn about it despite lacking prior knowledge. They were keen to grasp the useful information that could be gleaned from such a visualization.

Their feedback highlighted the ease and speed of finding information in the visualization, expressing appreciation for this aspect. They suggested incorporating the name, considering their use of certain conventions, might enhance their experience. However, this acknowledgment is not directly relevant to the testing, as the integration's name would be visible to users on the platform.

Participant III.

The third participant exhibited a higher level of technical expertise. They navigated through the visualization swiftly, examining each element by clicking and right-clicking on various components. The participant observed several unexpected behaviors in the visualization, such as the inability to close certain cards by clicking outside of them. They remarked that the connection button felt more like an activation of an action rather than a typical button, expressing a preference for showing information cards on hover instead of click, which they found to be more intuitive. Additionally, they provided detailed feedback on the connection forms, expressing confusion over some naming conventions and space distribution for certain elements. They encountered difficulty with the save button initially, as they did not immediately recognize it as disabled, strongly recommending a change in its appearance.

This participant also did not encounter any problems when given the task of finding some information. Only the naming conventions confused them a little.

Despite this, the user had a clear vision with many notes on enhancing the visualization overall. They noted that they like how fast they can orient within the integration and that this overall integration readability is their main point.

Participant IV.

This participant took a deliberate approach to exploring the visualization, appearing to reflect each component, which they later shared during the session. Their method led to more of a discussion-based testing approach rather than mere observation. The participant encountered some difficulty in closing certain cards, particularly the modal window for the SQL card. They speculated about the overall structure and suggested improvements they would like to see, unaware that opening the cards would provide the desired information. Overall, this user appeared enthusiastic about the visualization, highlighting many aspects they appreciated. They particularly praised the overall structure, the design, and the visualization of SQL statements.

A minor complication occurred during task completion and was attributed to the participant mishearing instructions. However, during the task, they suggested that displaying the file delivered by email could be more valuable than the email subject. Additionally, they gently proposed that the form for listing custom properties offer the option to display all properties without paging through them.

In their final feedback, this participant expressed excitement about the future use of the feature.

- **Participant V.**
  The last participant also opted to take a methodical approach to explore the visualization, carefully considering each step. However, despite reminders to do so, they did not express many opinions or share their thought process. Therefore, there was not much to describe because pushing them out of their comfort zone is not good practice.

  After exploring, the participant did not struggle to find any requested information within the second integration.

  Overall, it seemed this participant was looking for information that was not present in the visualization. Their final feedback mentioned missing the mapping in their database-to-database integration. Except that, this participant did not share more feedback.

6.3.3 Camel Visualization Testing

Two integration developers were selected as participants for the testing of *Camel* visualizations. Similar to the testing of the *Generic* visualization, users were encouraged to interact with the visualization freely. However, in this case, the testing sessions were more structured as discussions rather than purely observational.

The testing process tended to be longer due to the complexity of the tested integrations and the increased number of questions posed by the users throughout the testing sessions.

Below, key points from each discussion are highlighted. Due to the nature of the testing, which involved more discussion about the *Camel* integration than the final product, only select aspects are described.

- **Participant I.**
  The first participant began by analyzing the overall structure of the visualized integration, aiming to map the entire process to what is happening in the code of that integration. They noted multiple ways some components can be defined in code and how they map to the visualization. Here, they suggested that they would prefer some components to be embedded within clickable details rather than in the entire flow. The participant struggled with the logic behind the orange frame for the visualization, but after an explanation of its purpose, they acknowledged it. They noted that they struggled with the arrows rendering over the text and suggested providing visibility when two routes are interconnected. They recommended adding more specified names for components that represent endpoints to see the main flow of the whole integration right away. The participant clicked through all components to see which component was capable of visualizing what, noting that some information did not bring anything but acknowledging that this was due to the dynamic nature of *Camel*.

  In the end, they noted that they liked this tool for getting some initial information about the integration without looking for and exploring the source code. Despite that, they still prefer the code because they are used to it, and they can fully understand what's going on within the integration. They also noted that the visualization of large complex integrations gets confusing but stated that this is not the problem of the visualization, as such code is also confusing. They noted that they also saw some complicated integration in the *OIC* platform, which was confusing, clarifying the visualization problem for complex integrations.

- **Participant II.**
  The second participant also took the approach of slowly exploring the visualization.

Throughout the exploration, they expressed that they really liked the visualization for SQL endpoints. With the delivery event endpoints, they noted that they did not spot that it was a delivery event endpoint in the tooltip. Therefore, they suggested adding a description or a different icon for SQL and delivery event endpoints. Participant took multiple pauses throughout the exploration, thinking about possible adjustments and what could be embedded in the code to show more information in the visualization. Here, the participant came up with the idea of utilizing custom naming for the components' identifiers. They noted that using this logic during development could bring some additional information to the visualization. They also thought about the convention in the query naming, but here, they concluded that this information likely would not get into the visualization.

During further exploration, the participant suggested renaming some parts of the form to fit the old convention. They also proposed a feature that highlights the route based on a click on the interconnection from the previous route to provide faster orientation within the structure. In the final feedback, the participant stated they liked the visualization, and they would use this feature as the first tool for analysis so they do not need to download the whole code repository. They also noted it could help find mistakes within the integration faster in some cases, but they need some time to get used to this feature and understand its capabilities by using it. The participant added that it would be great if the visualization could display more details about some components, as in the case where lambda is used, and the expressions are not readable, but if they were, it would be really helpful.

6.4 Evaluation and Contribution

In this section, the thesis comprehensively evaluates the results obtained from the *Generic* framework and *Camel* framework visualizations, as assessed through user testing sessions. The findings are summarized to highlight each visualization's strengths, weaknesses, and overall contribution. Through this evaluation, the thesis aims to draw insights into the effectiveness of the visualizations in meeting user needs and identify areas for further enhancement.

6.4.1 Generic Framework Visualization Evaluation

Based on the testing sessions for the *Generic* visualization, the pivotal findings follow:

- **User Engagement and Understanding**
  Participants demonstrated varying levels of engagement and understanding while exploring the visualization. Some quickly grasped the connections and flow of the integration, while others encountered challenges with readability and navigating the interface.

- **Feedback on the Design**
  Participants provided valuable feedback on the interface design, highlighting areas for improvement such as readability, button clarity, and the intuitiveness of certain features like closing modal windows.

- **Value in Simplifying Workflows**
  Participants appreciated the visualization's ability to provide a clear overview of integration flows. They acknowledged its potential to save time and simplify work-

flows by providing quick access to information about integrations.

- **Suggestions for Enhancement**
  Participants provided valuable insights into potential enhancements for the visualization. While some suggestions were more like personal preferences, such as implementing hover-over effects instead of on-click interactions, others highlighted crucial improvements. These included enhancing readability for certain buttons, improving the clarity of closing modal windows, and incorporating a mapping feature for database-to-database integrations. The mapping feature was deemed crucial, so it was incorporated into the thesis development.

- **Overall Positive Feedback**
  Despite some challenges, participants generally had a positive impression of the visualization, noting its potential to enhance user interaction and efficiency within the integration platform.

6.4.2 Generic Framework Visualization Conclusion

The testing sessions provided valuable insights into the strengths and areas for improvement of the *Generic* visualization. While participants appreciated its potential to streamline workflows and provide a clear overview of integration flows, there were suggestions for enhancing the interface design and usability. Overall, the feedback aligns with the goal of the visualization, which is to help users navigate integration flows and provide an understandable visualization of the data pathways. These insights will inform further refinements to the visualization, ensuring it effectively meets the needs and expectations of users within the integration platform.

6.4.3 Camel Framework Visualization Evaluation

Based on the testing sessions for the *Camel* visualization, the pivotal findings follow:

- **Potential Use**
  The visualization tool demonstrated its value by providing quick insights into *Camel* integration structures without requiring users to delve into source code. Users appreciated its utility for initial analysis and error detection, particularly for gaining a high-level understanding of integrations.

- **Proposed Enhancements**
  Suggestions for improvement included enhancing component visibility, adding descriptions or icons for different endpoint types, and providing customization options such as custom naming for identifiers. Users also proposed features like highlighting routes based on interconnections.

- **User Familiarization**
  One user explicitly noted the need for familiarization with the tool, which was reflected in the slow exploration of both participants during the visualization exploration. This suggests a learning curve for users to fully utilize its capabilities.

6.4.4 Camel Framework Visualization Conclusion

The visualization tool effectively aligns with its goal of aiding navigation within *Camel* integration structures. Users found it valuable for gaining initial insights and detecting errors without extensively investigating the code. However, it became evident that while visualization is a valuable tool, it does not fully replace the necessity of code investigation for comprehensive understanding. A new idea emerged from the testing to enhance the tool's functionality by integrating static code analysis. By extracting and embedding additional information from the integration code, such as custom naming conventions and detailed component descriptions, the visualization could better capture the complexities of integrations and further streamline analysis processes.

Chapter 7
Conclusion

The primary objectives of this thesis were successfully achieved. A structured framework for representing various data integrations within corporate software infrastructures was developed, addressing the critical challenge of maintaining clarity amid heterogeneous data environments. This framework was realized by creating structural representations for two main groups of integrations: those based on the *Generic* framework and those utilizing the *Camel* framework. A standardized visualization of the data flow was also created for each structuralization, ensuring extensibility and ease of understanding. Furthermore, the entire process and implementation were thoroughly documented, facilitating future extensions and modifications. Through user testing, it was confirmed that the final product provides a clear visualization of data flows and significantly aids users in navigating their integrations more efficiently. The thesis also presented general information about the world of data integration, allowing readers to understand the discussed problematics.

During the problem analysis, it became evident that while existing solutions for the visualization of data flows exist, they are often embedded within specific platforms. Thus, the development of a new, tailored solution was deemed necessary. The analysis revealed expected challenges, particularly in the case of the more intricate *Apache Camel* framework, which necessitated a more sophisticated approach. Several challenges emerged throughout the implementation process, including integrating the new features with the existing logic within the platform.

The user testing proved valuable, providing essential insights into the contribution of the new visualization features. While the *Camel* framework visualization may not fully replace code examination, its primary purpose was to visualize the structure of data flows within *Camel* integrations. In this regard, it successfully met its objective, providing users with a clear overview of integration structures. However, further enhancements may be necessary to fully harness its potential, particularly in facilitating deeper insights and analysis. For the *Generic* framework visualization, promising results were observed, as it facilitated easy navigation within integrations, potentially reducing queries to the integration development team and improving understanding among users.

As a result, the contribution of the less complex visualization for the *Generic* framework appears more significant, enhancing the platform's usability. Conversely, the contribution of the more complex visualization for the *Camel* framework is diminished, mainly due to its dependence on understanding the *Apache Camel* framework, restricting its usage to developers rather than standard platform users.

In the end, the thesis developed two new frameworks for the platform in an extensible manner, allowing for various ways to enhance their features. Some suggestions for extensions are discussed below.

7.1 Future Work

Building upon the user testing feedback, the following enhancements are proposed for the future development of the system:

7.1.1 Advancing the Generic Framework Visualization:

Expanding on the proposed solution, adding a testing button for connections within the *Generic* visualization component offers users a straightforward means to verify connection accessibility. This feature addresses the need for real-time connection status checks, necessitating the development of backend logic capable of accurately testing various connection types. This logic already exists, but it is unreliable, which was also why this feature was not implemented in the first place.

Moreover, the existing structure for the connection form lays the groundwork for seamless integration with the backend, enabling users to make modifications directly. Implementing additional validation checks, such as directory existence verification, could enhance data integrity and user confidence in the system.

7.1.2 Advancing the Camel Framework Visualization:

Further exploration of the *Camel* framework's capabilities reveals opportunities for enhanced developer insight and integration understanding. Leveraging static code analysis to extract essential information, or piece of code itself, directly from the codebase empowers developers to gain deeper insights into integration components.

Furthermore, exploring experimental approaches, such as employing AI algorithms for code analysis, offers a promising avenue for comprehensive integration insights. However, it is crucial to develop custom AI solutions aligned with stringent company security policies to mitigate potential risks.

Additionally, analyzing outgoing and ongoing requests associated with integrations offers valuable real-time information, particularly in dynamic environments. This feature aids developers in understanding integration behavior. However, this leads to more real-time visualization than structural visualization.

7.1.3 Scaling User Testing:

Expanding the scope of user testing to encompass a broader audience is essential for gathering diverse perspectives and comprehensive feedback. Engaging a larger user base ensures that proposed enhancements align with various use cases and user requirements. Incorporating feedback from a diverse pool of users facilitates iterative refinement of features, ultimately enhancing the overall usability and effectiveness of the system.

References

[1] C. Ibsen, and J. Anstey. *Camel in Action*. Manning, 2018. ISBN 9781638352808.

[2] Miftah Santoso. Implementation Of UI/UX Concepts And Techniques In Web Layout Design With Figma. 2024, Vol 6 279. DOI 10.47233/jteksis.v6i2.1223.

[3] James Lewis, and Jeff Sauro. *USABILITY AND USER EXPERIENCE: DESIGN AND EVALUATION*. In: 2021. 972-1015. ISBN 9781119636083.

[4] A.H. Doan, A. Halevy, and Z. Ives. *Principles of Data Integration*. Elsevier Science, 2012. ISBN 9780123914798.

[5] Dataversity Digital LLC. *The Fundamentals of Data Integration*. `https://www.dataversity.net/the-fundamentals-of-data-integration/`. Accessed: 2024-01-21.

[6] *Data integration*. `https://www.ibm.com/cloud/learn/data-integration`. Accessed: 2024-01-21.

[7] Roee Shraga, and Avigdor Gal. One Algorithm to Rule Them All: On the Changing Roles of Humans in Data Integration. *Computer*. 2023, 56 (4), 102-109. DOI 10.1109/MC.2023.3240449.

[8] Shashank Shrestha, and Subhash Bhalla. A Survey on the Evolution of Models of Data Integration. *International Journal of Knowledge Based Computer Systems*. 2020, 8 (1 and 2), 11–16.

[9] Roee Shraga, Avigdor Gal, and Haggai Roitman. ADnEV: cross-domain schema matching using deep similarity matrix adjustment and evaluation. *Proceedings of the VLDB Endowment*. 2020, 13 1401-1415. DOI 10.14778/3397230.3397237.

[10] Roee Shraga, and Avigdor Gal. *PoWareMatch: a Quality-aware Deep Learning Approach to Improve Human Schema Matching*. 2021.

[11] Jan-Philipp Awick, Gerrit Schumann, and Jorge Marx Gómez. *Exploring Federated Learning for Data Integration: A Structured Literature Review*. In: *2023 International Conference on Big Data, Knowledge and Control Systems Engineering (BdKCSE)*. 2023. 1-8.

[12] El Abassi Merieme, Amnai Mohamed, Choukri Ali, Yossef Fakhri, and Gherabi Noreddine. *A survey on the challenges of data integration*. In: *2022 9th International Conference on Wireless Networks and Mobile Communications (WINCOM)*. 2022. 1-6.

[13] Mouna Rhahla, Sahar Allegue, and Takoua Abdellatif. Guidelines for GDPR compliance in Big Data systems. *Journal of Information Security and Applications*. 2021, 61 102896. DOI https://doi.org/10.1016/j.jisa.2021.102896.

[14] ElKindi Rezig, Mike Cafarella, and Vijay Gadepally. Technical Report: An Overview of Data Integration and Preparation. 2020.

[15] GeeksforGeeks. *Data Warehouse Architecture and Database Management System Topics*.

`https://www.geeksforgeeks.org/data-warehouse-architecture/`. Available online: Accessed on 20 March 2024.

[16] JavaTpoint. *Data Warehouse Architecture.* `https://www.javatpoint.com/data-warehouse-architecture`. Accessed: 2024-03-22.

[17] Leonardo Azevedo, Renan Souza, Elton de F. S. Soares, Raphael Thiago, Julio Tesolin, Ann Oliveira, and Marcio Moreno. *A Polystore Architecture Using Knowledge Graphs to Support Queries on Heterogeneous Data Stores.*

[18] Sylvain Lacroix, Emeric Ostermeyer, Julien Le Duigou, Florent Bornard, Sylvain Rival, Marie-France Mary, and Benoit Eynard. Lessons learnt in industrial data platform integration. *Procedia Computer Science.* 2023, 217 1660-1669. DOI https://doi.org/10.1016/j.procs.2022.12.366. 4th International Conference on Industry 4.0 and Smart Manufacturing.

[19] Jayesh Patel. *An Effective and Scalable Data Modeling for Enterprise Big Data Platform.* In: *2019 IEEE International Conference on Big Data (Big Data).* 2019. 2691-2697.

[20] Bharat Singhal, and Alok Aggarwal. *ETL, ELT and Reverse ETL: A business case Study.* In: *2022 Second International Conference on Advanced Technologies in Intelligent Control, Environment, Computing and Communication Engineering (ICATIECE).* 2022. 1-4.

[21] Amazon Web Services, Inc.. *What is Batch Processing?* `https://aws.amazon.com/what-is/batch-processing/`. Accessed: 2024-04-02.

[22] Amazon Web Services, Inc.. *What is Streaming Data?* `https://aws.amazon.com/what-is/streaming-data/`. Accessed: 2024-04-02.

[23] Sultan T. Alanazi, Nibras Abdullah, Mohammad Anbar, and Ola A. Al-Wesabi. *Evaluation Approaches of Service Oriented Architecture (SOA) - A Survey.* In: *2019 2nd International Conference on Computer Applications and Information Security (ICCAIS).* 2019. 1-6.

[24] Albert Stec. *Microservices vs. Service-Oriented Architecture.* `https://www.baeldung.com/cs/microservices-soa-differences`. Accessed: 2024-04-02.

[25] Amazon Web Services, Inc.. *What is SOA (Service-Oriented Architecture)?* `https://aws.amazon.com/what-is/service-oriented-architecture/`. Accessed: 2024-04-02.

[26] Amazon Web Services, Inc.. *What's the Difference Between SOA and Microservices?* `https://aws.amazon.com/compare/the-difference-between-soa-microservices/`. Accessed: 2024-04-02.

[27] Amirhossein Farahzadi, Pooyan Shams, Javad Rezazadeh, and Reza Farahbakhsh. Middleware technologies for cloud of things: a survey. *Digital Communications and Networks.* 2018, 4 (3), 176-188. DOI https://doi.org/10.1016/j.dcan.2017.04.005.

[28] J Sreemathy, R Brindha, M Selva Nagalakshmi, N Suvekha, N Karthick Ragul, and M Praveennandha. *Overview of ETL Tools and Talend-Data Integration.* In: *2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS).* 2021. 1650-1654.

[29] Guilherme Camposo. *Cloud Native Integration with Apache Camel: Building Agile and Scalable Integrations for Kubernetes Platforms.* 2021. ISBN 978-1-4842-7210-7.

[30] Apache Camel. *The Apache Software Foundation.* `https://camel.apache.org/`. Accessed: 2024-04-08.

[31] Apache Camel. *The Apache Software Foundation.* `https://camel.apache.org/manual/dsl.html`. Accessed: 2024-04-10.

[32] Rajender Reddy Vangala. INTEROPERABILITY NEXUS: AN IN-DEPTH EXAMINATION OF CROSS-PLATFORM INTEGRATION FRAMEWORKS, UNVEILING COMPARATIVE ARCHITECTURES AND PERFORMANCE DYNAMICS. *INTERNATIONAL JOURNAL OF ELECTRONICS AND COMMUNICATION ENGINEERING AND TECHNOLOGY*. 2020, 11 57-72.

[33] Inc. Salesforce. *Anypoint Platform.* `https://www.mulesoft.com/platform/studio`. Accessed: 2024-04-16.

[34] Oracle. *Using Integrations in Oracle Integration Generation 2.* `https://docs.oracle.com/en/cloud/paas/integration-cloud/integrations-user/img/hello_world.png`. Accessed: 2024-04-15.

[35] Marat Gubaidullin. *KARAVAN INTRODUCTION IN 4 MINUTES.* `https://camel.apache.org/blog/2023/01/karavan-intro/`. Accessed: 2024-04-15.

[36] Marat Gubaidullin. *KARAVAN DESIGNER PREVIEW RELEASE 0.0.10.* `https://camel.apache.org/blog/2022/02/camel-karavan-0.0.10/`. Accessed: 2024-04-15.

[37] Rafał Wojszczyk, Aneta Hapka, and Tomasz Królikowski. Performance analysis of extracting object structure from source code. *Procedia Computer Science.* 2023, 225 4065-4073. DOI https://doi.org/10.1016/j.procs.2023.10.402. 27th International Conference on Knowledge Based and Intelligent Information and Engineering Sytems (KES 2023).

[38] Prabhat Pokharel. *Information Extraction Using Named Entity Recognition from Log Messages.* 2018.

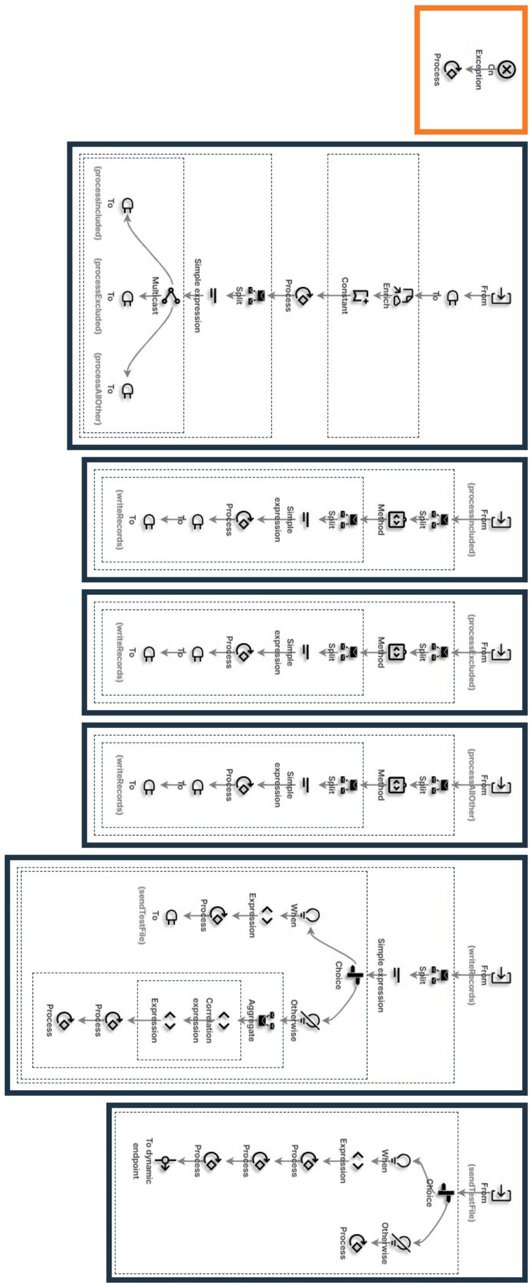
[39] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering.* 2020, 25 DOI 10.1007/s10664-019-09730-9.

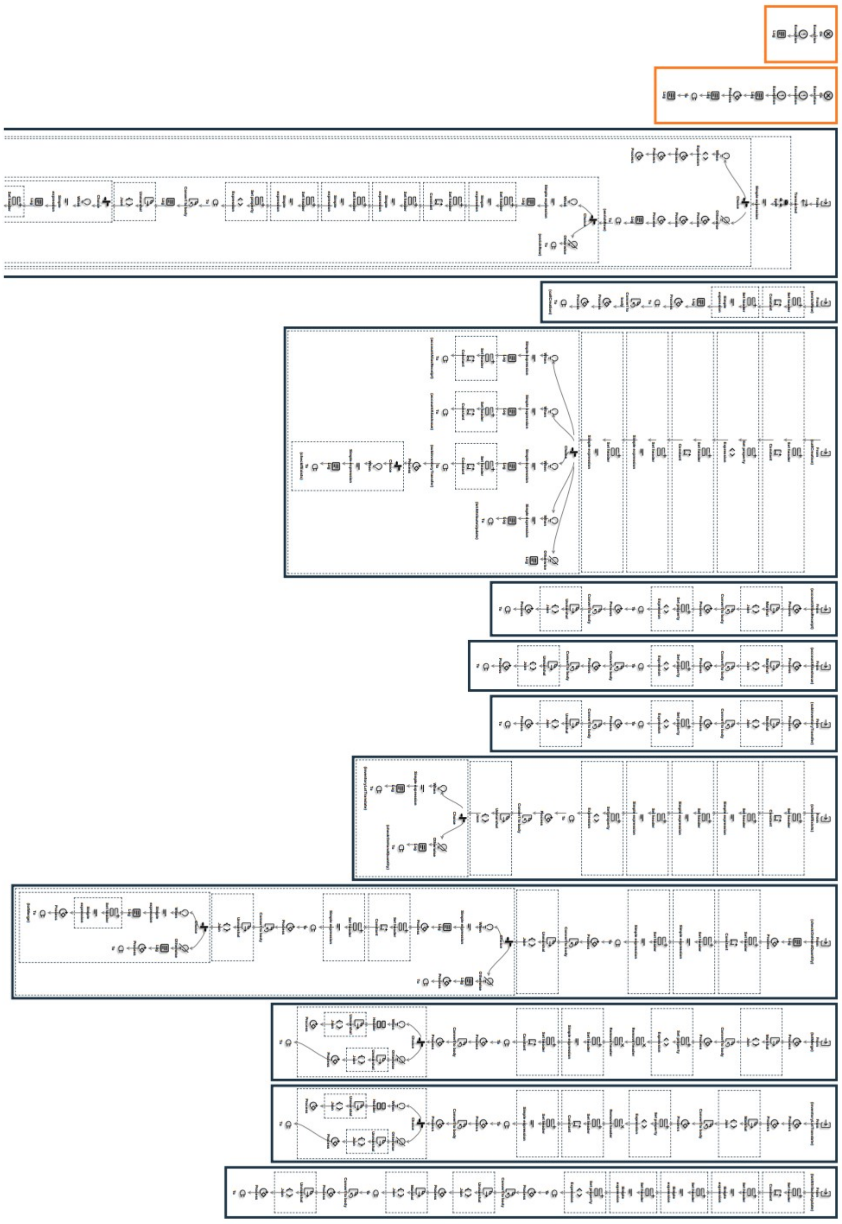
[40] David Elliott, and Eldon Soifer. AI Technologies, Privacy, and Security. *Frontiers in Artificial Intelligence.* 2022, 5 826737. DOI 10.3389/frai.2022.826737.

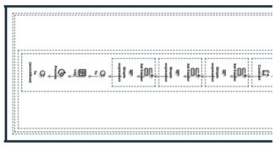
[41] Samar Al-Saqqa, Samer Sawalha, and Hiba Abdel-Nabi. Agile Software Development: Methodologies and Trends. *International Journal of Interactive Mobile Technologies (iJIM).* 2020, 14 246. DOI 10.3991/ijim.v14i11.13269.

[42] Sirpa Riihiaho. *Usability Testing.* In: *The Wiley Handbook of Human Computer Interaction.* John Wiley Sons, Ltd, 2018. 14. ISBN 9781118976005. `https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118976005.ch14`.

Appendix A
Additional Camel Integration Visualizations







Appendix B
Abbreviations and symbols

B.1 Abbreviations

AWS	Amazon Web Services
API	Application Programming Interface
B2B	Business-to-business
BMS	Building Management System
ERP	Enterprise Resource Planning
ESB	Enterprise Service Bus
ETL	Extract-Transform-Load process
FDBMS	Federated Database Management System
FDBS	Federated Database System
FQA	Federated Query Agents
HDFS	Hadoop Distributed File System
IBM	International Business Machines
JMS	Java Messaging Service
MIT	Massachusetts Institute of Technology
NoSQL	Not Only SQL
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
PaaS	Platform as a Service
PLM	Product Lifecycle Management
RDBMS	Relational Database Management System
SOA	Service-oriented architecture
ADnEV	Cross-domain schema matching using deep similarity matrix adjustment and evaluation
BigDAWG	Open source project from researchers within the Intel Science and Technology Center for Big Data
PoWareMatch	Quality-aware Deep Learning Approach