MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Design and Implementation of Apache Camel Karavan in the Integration Platform

Master's Thesis

BC. MICHAL ŘEZNÍK

Brno, Fall 2024

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Design and Implementation of Apache Camel Karavan in the Integration Platform

Master's Thesis

BC. MICHAL ŘEZNÍK

Advisor: RNDr. Jaromír Plhák, Ph.D.

Department of Machine Learning and Data Processing

Brno, Fall 2024


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Michal Řezník

Advisor: RNDr. Jaromír Plhák, Ph.D.

Acknowledgements

I would like to express my gratitude to all those who made the completion of this thesis possible. I am especially thankful to my supervisor, Prof. RNDr. Jaromír Plhák, Ph.D., for his support and advice throughout the duration of this project.

Special thanks go to my family and girlfriend for her support and encouragement during my studies.

Abstract

This thesis focuses on the design and implementation of an Apache Camel Karavan as part of the company's middleware platform. The platform, which handles thousands of daily integrations, faces challenges related to scalability, maintainability, and usability as it evolves. The primary goal of this work is to empower users to independently implement and manage integrations through Karavan, thereby reducing reliance on the middleware team and improving overall efficiency. The thesis begins with an analysis of the current platform, identifying key limitations and the need for an upgrade. Based on these findings, a solution is proposed to enhance integration management, monitoring capabilities, and maintainability. The implementation involves integrating Apache Camel Karavan into a Spring Boot environment, designing custom Kamelets for common integration patterns, and addressing critical aspects such as configuration management, security, and performance optimization. Testing strategies were employed to ensure the solution meets the platform's long-term needs. The results demonstrate improvements in efficiency, user autonomy, and scalability. In conclusion, this thesis presents a robust solution that not only addresses current limitations but also provides a scalable framework for future growth, ultimately enhancing the platform's reliability and user satisfaction.

Keywords

Apache Camel, Apache Camel Karavan, Integration Platform, Middleware, Java, System Integration, Data Integration, User Interface, Microservices, Visualization

Contents

List of Tables

List of Figures

1 Introduction

In the digital era, where systems in large companies rely on seamless interaction with external applications and data sources, middleware has become an indispensable component. It acts as the glue between diverse technologies, ensuring smooth and reliable data exchange. As organizations adopt cloud computing, IoT, and continue to rely on legacy systems, their middleware must evolve to meet increasing demands for flexibility, scalability, and integration.

Onsemi, a global semiconductor company, addresses this need by operating its own middleware platform. This platform plays a critical role in supporting business processes, enabling integration across internal tools and external systems. It facilitates digital transformation, cloud adoption, and automation, particularly in complex operational scenarios such as mergers and acquisitions.

Despite the platform's technological maturity, incorporating microservices, CI/CD pipelines via GitLab, and Snowflake for analytics, its integration workflows still face significant limitations. Users often depend on platform administrators or developers to create and manage integrations, especially when working with complex logic. This leads to repeated communication, manual coding, testing bottlenecks, and inefficiencies that delay delivery and increase maintenance overhead.

The motivation for this thesis arises from the need to modernize these integration practices and reduce dependency on technical staff. Integration tasks currently handled manually by developers, such as data transformation, API orchestration, and connection setup, could be simplified and partially automated to empower users. A more intuitive and visual approach can help bridge the gap between business and IT, and free up developer capacity for more critical tasks.

One of Onsemi's strategic goals for 2025 is to introduce a graphical user interface (GUI) that simplifies the creation and configuration of integration routes. This initiative is driven not only by the desire for technical innovation, but also by pressure from management to streamline the workload of the middleware team, whose responsibilities span a wide range of integration scenarios. Context switching, urgent support requests, and unclear user input further reduce team

efficiency. Providing users with a self-service tool would significantly reduce this burden.

This thesis addresses these challenges by integrating Apache Camel Karavan, a visual integration tool that allows users to design, configure, and manage routes through a graphical interface without writing code. Karavan uses reusable components known as Kamelets, which encapsulate common integration logic and make complex operations easier to manage. By extending this concept with custom Kamelets tailored to Onsemi's needs, the platform can support advanced scenarios while still being accessible to less technical users.

The solution is implemented within a Spring Boot environment, ensuring compatibility with Onsemi's cloud native and microservices based infrastructure. Visualization is used as a guiding principle, aiming to reduce manual coding, shorten deployment times, and improve communication between developers and business stakeholders. The resulting platform increases adaptability, enhances user autonomy, and aligns better with long-term business goals.

Ultimately, this thesis presents a vision of middleware that prioritizes accessibility, agility, and efficiency through visualization. By replacing opaque and code-heavy processes with intuitive visual design, it transforms integration into a strategic asset. The work underscores the importance of aligning with modern trends to empower users, simplify complexity, and build systems that scale effortlessly. These principles are critical for success in an era where technological adaptability defines competitive advantage.

2 System Architecture Overview

Modern IT platforms form the cornerstone of digital transformation, enabling organizations to navigate the complexities of interconnected systems while meeting the demands for scalability, reliability, and adaptability. These platforms are designed to handle a broad spectrum of functions, from managing data flows to orchestrating complex processes across distributed environments. At the heart of these platforms lies a set of foundational technologies that ensure seamless integration and interoperability, making them indispensable in today's business and technological landscape.

Platforms are further enhanced by technologies that complement middleware. Containerization, for instance, allows applications to run consistently across different environments by packaging them with their dependencies into lightweight, portable units. Orchestration tools such as Kubernetes automate the deployment and scaling of these containers, ensuring efficient resource utilization and high availability. Similarly, the adoption of serverless computing has introduced new paradigms where middleware integrates seamlessly with event-driven architectures, allowing developers to focus on application logic without worrying about infrastructure management.

The combination of these technologies creates a cohesive platform that is capable of meeting the demands of modern IT environments. Whether in traditional enterprises or cutting-edge digital startups, the role of middleware and related technologies cannot be overstated. These solutions enable organizations to build robust, scalable, and agile systems that are prepared to meet the challenges of an ever-changing technological landscape.

2.1 Middleware in the IT Ecosystem

Middleware [1] plays a particularly critical role within these platforms, acting as an intermediary that connects disparate systems, applications, and services. By providing a unified communication layer, middleware abstracts the complexities of different protocols, data formats, and interaction models, allowing developers to focus on business-specific logic rather than infrastructure-related challenges.

Historically, middleware evolved as a response to the growing need for interoperability in distributed computing environments, which began to emerge in the late 1980s and early 1990s. Initially, these systems were built to support simple data exchange and messaging. However, with the rise of enterprise software solutions like ERP and CRM systems, the scope of middleware expanded significantly to include transaction management, orchestration, and advanced message processing capabilities.

In the contemporary landscape, middleware solutions have become even more sophisticated, reflecting the transition from monolithic architectures to distributed microservices and cloud-native [2]. These technologies are integral to addressing challenges such as system interoperability, scalability, and fault tolerance. Middleware platforms ensure that applications built on different stacks, utilizing diverse data models, and operating across varied environments can seamlessly interact with one another. This capability is particularly important in hybrid cloud scenarios, where legacy on-premises systems must coexist and interact with modern cloud-based applications.

Another key aspect of middleware is its ability to support complex workflows by providing advanced orchestration and automation capabilities. This involves coordinating multiple services and ensuring that they operate in a coordinated manner to achieve business goals. Middleware solutions also incorporate monitoring and analytics features that enable organizations to gain insights into system performance, identify bottlenecks, and optimize operations.

The evolution of middleware has also been deeply influenced by the increasing reliance on APIs for system integration. Middleware platforms today often incorporate API management capabilities, enabling organizations to design, secure, and monitor their APIs effectively [3]. This shift reflects a broader trend toward service-oriented architectures and microservices, where APIs serve as the primary mechanism for interaction between services.

2.2 Key Technologies Used in the Onsemi Integration Platform

The middleware platform under development serves as a cornerstone for managing and building integrations in a dynamic enterprise environment. Selecting the right technologies is critical for ensuring system scalability, maintainability, and performance. Choosing the wrong technology can lead to technical debt, increased operational complexity, and difficulties in adapting to evolving business requirements. A well-chosen technology stack enhances development efficiency, improves system resilience, and supports long-term growth.

Based on microservices architecture, the platform integrates technologies like Java, React, TypeScript, Apache Camel, Kafka, Snowflake, Maven, and GitLab Pipelines to deliver a scalable, resilient, and easy-to-use environment for developers.

2.2.1 Modular Design with Microservices

The platform's foundation is a microservices architecture [4], ensuring modularity and flexibility. Each service is designed to perform a specific function independently, enabling teams to develop, test, and deploy components in isolation. The backend, implemented in Java, offers a robust framework for handling complex integration logic and transactional workflows. The frontend, built with React and TypeScript, provides an intuitive interface for defining, monitoring, and managing integrations.

This modular approach facilitates independent scaling of services to meet fluctuating demands, supported by containerization technologies and orchestration tools like Kubernetes. It also simplifies maintenance and upgrades, as individual components can be updated without disrupting the entire system [5]. Figure 2.1 visually compares the monolithic and microservices architectures, where monolithic systems encapsulate all functionality within a single deployable unit, often leading to scalability and maintenance challenges as the system grows.

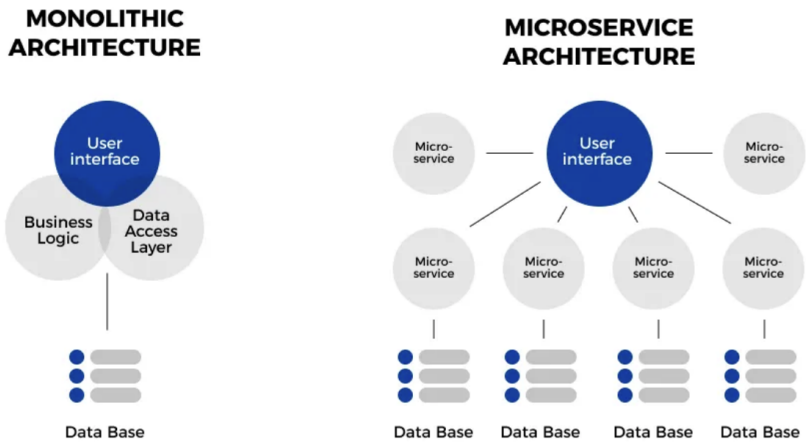

Figure 2.1: Illustration of the differences between monolithic and microservices architectures. Source: [6]

2.2.2 Data Analytics with Snowflake

To complement the integration layer, the platform incorporates a modern data analytics stack. One of the key components is Snowflake, a cloud-native data warehouse designed for performance and scalability.

Processed data flows seamlessly into Snowflake, where it is stored and made available for advanced analytics. Snowflake's architecture ensures that data processing remains fast and scalable, providing insights that help inform business decisions [7]. Developers can use integration pipelines to extract, transform, and load (ETL) data efficiently, bridging the gap between operational systems and analytical workloads.

2.2.3 Integration Architecture with Apache Tools

The platform utilizes two powerful tools from the Apache ecosystem—Apache Camel and Apache Kafka—to support seamless data integration and message-driven communication. Together, they form

a robust foundation for routing, transformation, and asynchronous data exchange across diverse systems.

At the core of the integration layer lies Apache Camel, a flexible framework that enables developers to connect various systems using Enterprise Integration Patterns (EIPs). With support for a Domain-Specific Language (DSL) in Java, XML, or YAML[1], Camel simplifies the implementation of complex integration logic and improves developer productivity [8].

Apache Kafka complements Camel by providing a high-throughput, distributed messaging system suitable for event-driven architectures. It enables asynchronous communication between decoupled services, supports real-time event streaming with low latency, and ensures reliable message delivery.

The synergy between Apache Camel and Kafka allows for powerful integration scenarios. For instance, Camel can consume messages from Kafka topics, process them through predefined workflows, and then forward the results to other systems, such as REST APIs or databases. This combination enables real-time processing, flexible orchestration, and reliable communication across the integration platform.

2.3 Apache Camel Karavan

Apache Camel Karavan is a tool designed to simplify the creation and management of integration routes using the Apache Camel framework (see Section 2.2.3). It has a visual interface that provides users the ability to design integration flows by dragging and dropping components. It provides a more intuitive way to create complex data integrations without the need to know DSL. This helps users who prefer a visual approach to design and configure integration routes.

The tool is composed of several modules that work together to provide a seamless development experience. The first key module of Karavan is the Visual Route Designer, which offers a user-friendly interface for creating integrations by connecting components such as Kafka, HTTP, and FTP [9]. The next module is the Code Generator,

1. Domain-Specific Language (DSL): A programming language designed for a specific application domain. In Apache Camel, DSL refers to the syntax used to define integration routes.

which translates the designed routes from the Visual Route Designer into Camel DSL code in YAML or XML format. This ensures that the integration created in the designer accurately matches the actual implementation. Another essential module is the Integration Runtime, responsible for execution and deployment. It allows users to deploy their routes to Kubernetes while also supporting scalability, efficient resource management, and integration with DevOps practices.

Another key aspect is the Component Library module, which provides a collection of preconfigured components that can be used to build integration routes efficiently. This library includes a wide range of protocols and systems, such as messaging platforms, databases, APIs, and more. Additionally, it contains various Kamelets, which are reusable patterns designed to simplify common integration scenarios. By abstracting complex configurations into ready-to-use building blocks, the library enables developers to quickly connect to cloud systems, various platforms, and even mediate SaaS[2] applications for users without requiring deep technical knowledge.

This library offers a robust customization option, enabling developers to create custom components and custom Kamelets tailored to their organization's specific needs. By doing so, they can encapsulate frequently used integration logic, streamline workflows, and develop reusable templates. This option reduces duplication and ensures consistency across integrations and significantly accelerates the development process, making it easier and more efficient to implement new integration solutions.

For monitoring and managing integration routes, Karavan also includes the Configuration, Monitoring, and Extensibility module. This module allows users to configure settings for individual routes, such as endpoint parameters, error handling, and other essential options. Additionally, it provides basic monitoring capabilities to track the performance and status of integrations. This implementation helps identify bottlenecks or failures in real time, ensuring more efficient troubleshooting and optimization of integration workflows.

---

2. Software as a Service (SaaS): Software delivery model where applications are hosted and managed by a provider and accessed by users over the internet. This enables developers to offload infrastructure management and deployment tasks, allowing users to configure and utilize the software independently.

3 Current State Analysis

Middleware platforms have become crucial for modern enterprise IT infrastructure, especially for large organizations like Onsemi [10] that rely heavily on seamless communication between internal and external systems. The purpose of these platforms is to act as a glue that connects disparate systems, databases, and applications. Without this layer of communication, it would be challenging to manage data integrations, which could lead to inefficiencies or the possibility of disruptions in business operations.

Many companies have explored the option of outsourcing their middleware platform to third-party providers as a way to reduce costs and leverage external expertise. This can lead to faster deployment and reduced operational overhead. However, there are also drawbacks, such as limited control over the platform and potential security risks. For Onsemi, data integrations are critical to daily operations. Maintaining and managing an in-house platform provides several advantages. It offers a better solution for data security, customization, and performance optimization. Additionally, it ensures that the platform can be tailored to meet specific business needs or deliver features for users without being constrained by the limitations of a generic third-party solution.

The main purpose of Onsemi's middleware platform is to enable communication and integration between systems, endpoints, databases, and other data sources and their final destinations. Figure 3.1 illustrates the communication scheme of the system, showing the interactions between its individual components. Without an integration platform, building and maintaining data routes would be inefficient and error-prone. For example, one of the key use cases is the Digital Thread project. It ensures that data from all relevant systems is synchronized, allowing the company to trace detailed information about each product—such as who was on shift during production or what materials were used. This level of traceability supports quality control and compliance across the entire production lifecycle. These systems might be internal solutions such as ERP systems for resource planning or CRM tools for customer relationship management, as well as external partner systems.

With the critical role of the platform and all its benefits, there are also challenges, including limitations in scalability, performance, and maintainability. Given the number of integrations, it is essential to provide users with as much control over managing their integrations as possible. One of the key challenges is ensuring an intuitive and user-friendly interface. In terms of performance and usability, the platform must be able to adapt to the company's growth and evolving technologies. Therefore, this chapter will analyze the current state of the platform, identifying its key strengths and weaknesses and assessing the need for an upgrade. The findings of this analysis will serve as the foundation for defining the requirements of an improved solution. This solution aims to support the company's long-term goals while enhancing the user experience, making it more efficient and seamless.

Figure 3.1: Communication scheme of the Onsemi middleware platform. Source: Onsemi Internal Middleware Architecture Guide (2024).

3.1 Analysis of Current Onsemi Platform

The current platform consists of five essential microservices: Runtime Engine, Dashboard, Delivery Engine, Logging Service, and Scheduler. Together, they form a platform that is crucial for the company's daily operations. Almost 2,000 gigabytes of data are transferred every day through 3,000 data integrations, which play a crucial role in the company's day-to-day operations and support data-driven decision-making. This is made possible by the ability to transfer vast amounts of data across the organization.

The system supports multiple routes, including connections to databases, file systems, cloud services, messaging services, APIs, and more. For integrations that involve simple logic, a Generic Interface is created. An interface, in this context, refers to data integration, as it defines how different systems connect, exchange, and process data. While data integration focuses on consolidating different data sources into a unified system, an interface serves as the specific mechanism that enables this connection and data exchange. The Generic interface is the default type of interface and can now be created using a wizard available in the platform's dashboard. These simple data transfers are the most commonly used in the platform. The second type of interface is a Custom interface, which is typically used for more complex logic that cannot be defined using the Generic Interface wizard. Custom interfaces must be created by integration developers, which often leads to bottlenecks in efficiency due to recurring patterns in their design. The biggest challenges in this process usually involve communication with users, preparing test data from source systems, and other operational hurdles. In some cases, it is even more efficient to create multiple Generic interfaces that work together to achieve the desired routing instead of designing and developing a dedicated Custom interface.

The platform runs on four environments, each consisting of multiple groups of servers (clusters). These environments are ranked based on their security and importance, progressing from DEV, INT, UAT, to PROD. DEV is used purely for developing new features and implementing new logic in custom interfaces. The second environment, INT is used for integration testing. It helps catch errors and bugs that could become more problematic in higher environments. UAT is a high-level environment where user acceptance testing is performed. This is the

final stage before production, making it crucial to have properly prepared test data from source systems to test the actual interface. It is also the last opportunity to discuss and refine the logic before deployment to production. PROD is the production environment, where the final version of the platform operates. Most bugs must be resolved before release to this environment to ensure system stability. Features move to production either through a four-week release train or manually. Deployments to production servers are usually scheduled to ensure everything is ready for activation.

3.2 Problems and Limitations of the Current Solution

The existing platform is capable of large-scale data integration. However, there are still issues that affect its scalability and agility. A major challenge is the communication and development of custom interfaces. These require manual coding by integration developers and direct interaction with customers, who do not always provide complete and clear information. This process is time-consuming and prone to inconsistencies. Recurring patterns such as API authentication or data transformation are often implemented redundantly. This duplication increases the risk of errors and can sometimes lead to bugs. The lack of a unified framework results in fragmented solutions, increasing technical debt and making maintenance more complex.

Another challenge is the platform's reliance on code-centric workflows. While the generic interface includes a wizard that simplifies basic integrations, custom interfaces with complex logic require developers to write low-level code without visual aids.

The absence of a visual representation of data routes makes it difficult for stakeholders[1] to understand the system while designing, debugging, or communicating flow logic. This lack of visualization increases the risk of miscommunication and misaligned requirements, which may only become apparent during the testing phase. These issues persist during testing, where users and developers must coordinate their efforts. Users are responsible for preparing test data,

1. Stakeholders include end users, developers, operations teams, security specialists, and business leaders—each group brings different priorities such as usability, maintainability, performance, security, or strategic alignment [11]

while developers focus on testing the interface. This separation of roles further complicates the process and may lead to deployment delays.

Due to these challenges and the fact that scalability is one of the most critical factors for large companies, it is essential to find a solution that ensures the platform remains flexible and does not limit its ability to adapt to evolving business needs. Such a solution should reduce complexity and support both technical and non-technical users. Addressing these needs is key to maintaining long-term sustainability and aligning the platform with strategic business goals.

3.3 Evaluation of Visual Integration Tools

Due to the need to provide the most suitable solution for a user-friendly way of designing integrations, it was necessary to find the best method to implement it. The organization has its own platform, and the team managing it consists of fewer than 10 administrators. Because of this, creating such a large project would require a lot of resources. Developing a full-featured visual integration application from scratch would necessitate significant time, resources, and long-term support efforts. For this reason, we decided to evaluate existing open-source[2] solutions that could complement our existing integration platform, which already supports Apache Camel.

For the project, we require the following criteria:

- Support for visual flow design with a modern and intuitive user interface
- Be open-source or have a sustainable cost model.
- Possibility to create custom components
- Require minimal custom development and changes to the current project
- Be lightweight and easy to deploy

2. Open source refers to software whose source code is made freely available for use, modification, and distribution by anyone, promoting collaborative development and transparency [12]

3.4 Evaluated Tools

Based on the defined criteria, the following tools were found:

- **Apache Camel Karavan** – A lightweight, open-source visual editor for Apache Camel routes. It supports YAML and Java DSL and is designed to work natively with Camel K and Kubernetes.
- **Red Hat Fuse Online (Syndesis)** – A commercial product based on Apache Camel offering a polished UI and visual flow design capabilities. However, it is complex to install and maintain.
- **MuleSoft Anypoint Platform** – A commercial integration platform with extensive enterprise features and connector support. Not directly compatible with Apache Camel and requires a costly license.
- **WSO2 Enterprise Integrator** – An open-source integration suite with visual tooling, extensive protocol support, and cloud features. However, its setup is complex and not directly Camel-based.
- **Node-RED** – A visual low-code tool. While easy to use, it is not intended for complex enterprise integration scenarios and lacks Camel compatibility.

After evaluating the above tools, Apache Camel Karavan was selected as the most suitable solution for our use case. The key reasons are that it is fully compatible with Apache Camel, is lightweight, it has intuitive graphic interface and there is not any licensing cost, unlike other tools that would require either financial investment or additional setup effort. The decision to adopt Karavan allows our existing platform utilizes visual designing without reinventing the wheel, while also enables future scaling and developer empowerment.

3.5 Proposed solution

The challenges described in Section 3.2 highlight the need for a modern solution. Apache Camel Karavan addresses these issues by providing

visual tools for designing data routes. It includes drag-and-drop functionality, a library of components, and Kamelets—reusable logic building blocks—that reduce the need for manual programming. These prebuilt components and Kamelets save time and ensure consistency across the development team. Additionally, Karavan offers built-in testing tools and DevOps pipelines for streamlined deployment.

The visual design enhances communication between developers and non-technical stakeholders, reducing misunderstandings and accelerating the delivery of new custom interfaces. It also allows more technically inclined users to design and test their own routes, increasing efficiency.

By adopting Karavan, the platform becomes more flexible, easier to maintain, and enables developers to focus on scaling the system and achieving the company's goals. This shift not only improves development efficiency but also aligns the integration process more closely with strategic business objectives, ensuring long-term sustainability and adaptability.

3.6 Requirements

For a successful integration of Apache Camel Karavan into the Onsemi middleware platform, several challenges and requirements must be addressed. Defining clear and precise requirements is critical to ensuring that the solution aligns with business goals. Without well-defined requirements, the implementation could lead to inefficiencies, misaligned expectations, and various other issues that may slow down development [13].

The following sections list the key functional and non-functional requirements for this integration.

3.6.1 Functional Requirements

Visual Interface Design

- Provide a drag-and-drop interface for designing data flows, enabling developers and non-technical users to create, modify, and visualize integration workflows.

- Improve the Apache Camel Karavan user interface so that users only see buttons and functionalities they are permitted to use, creating a streamlined and user-focused experience.

Configuration Provision

- Allow users to configure their own connections via predefined placeholders, making the configuration of Kamelets easier for business users.

Custom Kamelet Creation

Kamelet objects in Apache Camel serve as reusable templates for common patterns and connectors, enabling faster and easier development. As part of the proposed solution, the creation of custom Kamelets is considered to address specific integration scenarios that may not be covered by existing components. The goal is to encapsulate logic frequently used in the organization's business integrations, such as connectors to databases, cloud services, or various communication protocols.

- Evaluate existing components and predefined Kamelets to determine their coverage and applicability.
- Identify integration scenarios not supported by available components.
- Design and implement custom Kamelets to handle missing or organization-specific use cases.

Deployment Automation

- After a route is created in the graphical designer, it should be automatically deployed to the Git repository used for managing routes.

Minimal Changes to the Karavan Instance

- To ensure maintainability and future usability, the integration must require minimal modifications to the Karavan instance.

3.6.2 Non-Functional Requirements

Usability

- Provide an intuitive interface for developers and business users, minimizing the learning curve.
- Deliver comprehensive documentation and training materials for onboarding new users.

Maintainability

- Ensure a modular and well-structured codebase to simplify updates, debugging, and future development.
- Reuse predefined components, such as Kamelets, to improve consistency and reduce duplication.

Scalability

- Deploy the initial beta version as a standalone Docker container for simplicity and ease of testing.
- Using a Docker container enables future deployment to a Kubernetes environment, supporting future scaling needs.

4 Solution Design

The integration of Apache Camel Karavan into Onsemi's middleware platform requires a well-structured architecture that ensures flexibility, scalability, and maintainability. The system will be implemented within a Spring Boot environment and designed as an independent container, aligning with the overall system architecture, which separates key concerns such as interaction, scheduling, execution of integrations, and logging. This modular approach will improve scalability, simplify debugging, and enable independent updates for components without affecting the entire system.

The proposed solution consists of a single microservice that will provide a graphical interface for managing integrations, handle deployment, and manage the synchronization of YAML-based integration definitions. The microservice will expose RESTful APIs that allow users to interact with Karavan, deploy integration routes, and monitor their execution.

The solution will include a mechanism that ensures that YAML integration files stored in a GitHub repository are synchronized with a containerized storage system, where the Apache Camel runtime will execute them. This synchronization process will be automated, ensuring that any modifications, updates, or newly created integration routes are reflected in the runtime environment without requiring manual intervention.

By designing the platform as a single container, it will be possible to scale individual parts of the system independently, ensuring that performance is optimized for handling large numbers of integration routes while maintaining stability and reliability. The microservice will orchestrate the execution of Apache Camel routes, ensure real-time synchronization of integration configurations, and provide logging and monitoring capabilities for users to track integration performance and detect issues proactively.

This chapter will describe the technical design of the integration, covering its Spring Boot implementation, overall architecture, handling of integration files and execution, configuration management, and security and scalability considerations.

4.1 Design of Apache Camel Karavan into Spring Boot Environment

Before integrating Apache Camel Karavan into the Spring Boot environment, it is first necessary to update Apache Camel to version 4.8. This update was really important as older versions lacked the necessary backend processing capabilities required for handling user-created integration routes efficiently. By upgrading to Camel 4.8, the system now supports improved route execution; there are also enhanced API interactions and better compatibility with Karavan's visual editor, which leads to integration definitions being dynamically processed and deployed.

Another key design requirement for the system is the modification of the Karavan GUI to better align with user needs and specific use cases. The default interface gives a wide range of functionalities, many of which are not relevant to the intended use case. To enhance usability and streamline the user experience, the GUI will be modified to remove unnecessary options, ensuring that users see only the features that are relevant to their integration tasks.

Additionally, Karavan requires a Git repository to manage integration definitions, as it has built-in tools for storing, versioning, and deploying YAML-based integration routes [14]. To facilitate this, it is necessary to prepare two separate Git repositories—one for the microservice itself and another for managing and deploying integration YAML routes.

A detailed interaction map between the user and system modules is illustrated in Figure 4.1, highlighting how different components collaborate to support integration creation and deployment.

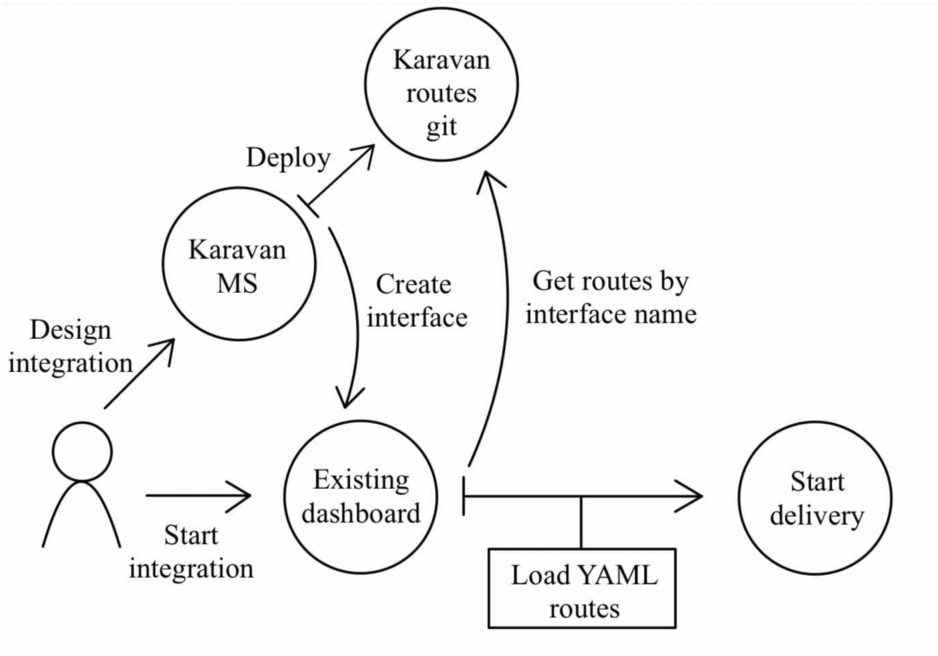

Figure 4.1: Design of implementation Apache Camel Karavan graphic interface into existing system

4.2 Architectual design

The architecture of the system follows a microservice-based approach, with Apache Camel Karavan integrated into the middleware platform to provide a graphical interface for designing, managing, and executing integrations. The workflow consists of several key components:

- **User Interaction** – The user interacts with the system through an existing dashboard, where they can either start an integration or design a new integration using Karavan.
- **Karavan Integration Design** – If the user chooses to design a new integration, they will do so using Apache Camel Karavan, which provides a visual route editor for defining integration workflows.

- **Version Control and Deployment** – Once the integration is designed, Karavan deploys the integration to a Git repository, which serves as the centralized storage for all integration YAML routes.

- **Retrieving Routes for Execution** – When an integration is started from the dashboard, the system retrieves routes from the Git repository based on the interface name. This ensures that the latest version of the integration is always used.

- **Execution of Integrations** – After retrieving the YAML-based route definitions, the system loads them into the runtime environment and executes them using Apache Camel.

- **Monitoring and Completion** – Once the integration is running, it continues executing until it reaches completion, at which point the system marks the process as finished. Logs is accesible via existing dashboard interface.

4.3 Concrete Custom Kamelets

Kamelets (term formed by combining Camel K and templates) are modular and configurable building blocks used for constructing more complex parts of integration flows. They simplify the development of integration routes and enhance usability, while also promoting standardization in the development process. Kamelets abstract away the complexity of traditional Camel route definitions. They are especially useful in the Karavan graphical interface, where users can create integrations by dragging and connecting visual components that represent the individual steps of the integration process. These logical blocks are then translated into YAML files, which can be stored in an online repository. In the future, this may become a key component of the user interface, as the abstraction makes the platform more accessible even to business users.

During the analysis phase of the thesis, it was found that the specific integration needs of the Onsemi integration platform do not currently require the development of custom Kamelets. This is primarily due to the comprehensive library of predefined Kamelets and

components that come with Apache Camel. The shift from the original plan was also influenced by the limited official documentation available for developing custom Kamelets in Apache Camel Karavan.

This predefined library covers a wide spectrum of integration scenarios and supports a broad range of technologies. For example:

- **Protocol-based Kamelets** – These enable communication over various protocols such as HTTP, FTP, MQTT, Kafka, and more. Kamelets allow seamless integration between systems using standardized transport mechanisms.
- **Cloud Service Kamelets** – These Kamelets simplify connectivity to cloud platforms like AWS, Azure, and Google Cloud, making data exchange easier and faster.
- **Transformation Kamelets** – Used for converting data formats (e.g., JSON to XML), which facilitates further processing and integration between systems with different data representations.
- **Utility Kamelets** – Provide supporting functions such as timers, logging, conditional routing, and other tools to control and monitor integration flows.
- **Database and Messaging Kamelets** – Allow integration with databases and messaging systems (e.g., JDBC, MongoDB, JMS, OracleDB), enabling reliable data exchange and event-driven communication.

Thanks to the wide range of predefined Kamelets and their support for all key integration scenarios relevant to Onsemi's business needs, it is no longer necessary to create and implement custom Kamelets as originally planned. The existing components already cover the required functionality, which helps save development time, reduces the complexity of maintaining custom solutions, and improves maintainability and usability for developers.

4.4 Configuration Management

To ensure seamless deployment across different environments, it is essential to avoid hardcoded values and enable dynamic configura-

tion management. During development, testing, and production, a centralized system will be used to load and update configurations dynamically, eliminating the need for manual redeployment. This approach will result in a smoother development process, more efficient testing, and reliable operation in production.

To achieve this, the platform will utilize Consul as the centralized configuration store, providing dynamic service discovery, health checking, and a key-value store for configuration settings [15]. This approach allows configuration values to be updated in real time, ensuring that changes propagate across all running instances in each environment without manual intervention.

Each environment (as described in Section 3.1) has its own Consul instance, allowing for environment-specific key-value configurations. This is particularly useful for settings such as timeouts, where higher thresholds can be configured for production to accommodate increased workload demands.

To ensure security and access control, Consul is secured using Access Control List (ACL) keys. These ACL policies restrict access to configuration settings, ensuring that only authorized services and users can read or modify specific keys. Additionally, sensitive data such as API keys and credentials are encrypted and managed separately using a secure secrets management solution.

4.5 Security and Scalability

In the context of integration platforms, ensuring security and scalability is essential for maintaining system reliability, data protection, and long-term sustainability [16].The primary focus of this implementation was to improve the design and management of integrations. Therefore, it was more important to consider how the solution fits into the broader infrastructure from both operational and architectural perspectives.

4.5.1 Security

The security of the Apache Camel Karavan must be designed with respect to the internal standards of the organization. Since the solution

is deployed within the internal infrastructure, it is possible to access only via corporate VPN[1], there was no need to implement a separate authentication or authorization layer. Access to application is already secured via the company's network policies.

For monitoring purposes, existing system utilizes Grafana [18] and its own traffic visualization application to visualize metrics collected from integrations. Logging is limited to tracking the updates and initiation of integration runs.

4.5.2 Scalability

From a scalability perspective, the solution is designed for future extensibility. The initial beta version of the implementation will run as single Docker container, what brings simplified testing and rapid feedback without the overhead of full orchestration layer. This setup is used for early-stage validation and internal evaluation of the Karavan interface and its integration capabilities. In the future, it can be seamlessly deployed into a Kubernetes cluster, enabling better orchestration, fault tolerance, and autoscaling capabilities [19].

1. Virtual Private Network (VPN): A technology that establishes a secure, encrypted connection over the internet, allowing users to protect their data and maintain privacy while accessing networks remotely. [17]

5 Implementation

The integration of Apache Camel Karavan into the Onsemi middleware platform represents a pivotal advancement in enhancing the efficiency and usability of the company's integration processes.It marks a transition toward a more modular, flexible, and user-oriented development environment, paving the way for a more scalable model. This chapter provides a comprehensive exploration of how the theoretical concepts and plans outlined in the previous chapter will be translated into a functional system that aligns with both the established requirements and the company's overarching business goals and technological landscape. Emphasis is placed on environment preparation, system integration, Kamelets development, and ensuring that the middleware platform can operate reliably with the new system components. These enhancements aim to improve accessibility for users and streamline development efforts, ultimately saving time for developers.

A key objective of this implementation is to streamline the creation of custom interfaces and reduce communication overhead that often hinders development. By introducing a user-friendly graphic designer for route creation, the system aims to simplify integration processes and minimize the need for manual coding. This approach empowers users to design their own interfaces using the graphical tool, deploy them directly into an existing dashboard, and test, run, or promote them to higher environments with ease. Such capabilities are expected to significantly save time for both developers and system administrators.

This chapter delves into the implementation details of integrating this new system component into the existing platform. It focuses on setting up the environment, developing the graphic designer system, creating custom Kamelets, and establishing provisioning strategies to ensure seamless configuration and deployment. The integration also required developing a secure property handling mechanism, allowing sensitive configuration data to be managed without exposing them in source files or requiring manual intervention. The following sections provide an in-depth exploration of the decisions made during the implementation process and their broader impact on the system.

5.1 Preparation of Development and Testing Environment

One of the first steps in the integration platform development involved configuring a local environment to safely implement, test, and extend new integration logic on top of the existing system. A properly configured local setup enables rapid iteration, easier debugging, and efficient validation of changes before deploying them to shared or production environments.

To achieve this, we chose an existing project that serves as a typical archetype for Apache Camel integrations. Its established structure provided a solid starting point, allowing us to focus on building new functionality rather than spending time on basic configuration. We were able to run the integration locally and, using Maven, connect all the required modules and dependencies necessary for our planned upgrades and further development. The scheme of this environment is shown in Figure 5.1.

This setup proved to be both flexible and maintainable. It allowed us to test new features in isolation, ensure compatibility across components, and prepare stable releases for deployment. The local environment became an essential part of our workflow, streamlining development and supporting continuous improvement throughout the project lifecycle.

The structure of the integration platform consists of several Maven modules, each serving a specific role:

- **INTEGRATION PROJECT** – Main module that serves as archetype project for integration. Contains Junit test that can be started localy.
- **UE CAMELBRIDGE – FTP** – Handles Camel routes related to FTP-based file transfers.
- **UE COMMON** – Provides shared utilities, constants, and reusable data structures.
- **UE CAMELBRIDGE** – Defines the base routing logic and integration configuration using Apache Camel.

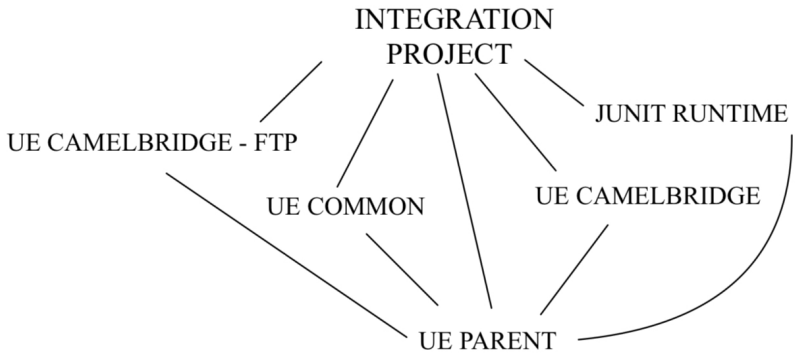

Figure 5.1: The local development environment served as a testing ground for upgrades and a workspace for implementing new functionalities in a controlled and isolated setting.

- **UE PARENT** – Maven parent project managing dependency versions and common build settings.
- **JUNIT RUNTIME** – Contains dependencies and tools required for running unit and integration tests.

All of these modules are now maintained in local versions, collectively forming a comprehensive environment for the development and verification of new functionality.

5.2 Environment Upgrade

Upgrading Apache Camel was a complex process primarily due to the significant version jump from 1.8 to 4.8, which is the minimum required version for leveraging Apache Camel Karavan. This upgrade was not just a matter of a compatibility adjustment—it introduced fundamental changes to the framework's core, especially in the definition, loading, and management of routes. Those changes are listed in Table 5.1

The most significant change, for our use case, was in the use of the RouteBuilder class and overall lifecycle management of routes. In version 1.8, routes were mainly written in Java DSL using a relatively flexible and permissive interface. In contrast, version 4.8 enforces

Table 5.1: Comparison of chosen features of Camel 1.8 and Camel 4.8

Feature	**Camel 1.8**	**Camel 4.8**
DSL	Java only	Java, YAML, XML, REST, Groovy
Route loading	Static	Dynamic via loaders
Lifecycle	Manual	Managed, automated
Configuration	In code	Properties, CLI, env, hot reload
Observability	Manual logging	Health, metrics, tracing out-of-the-box
Extensibility	Cumbersome, via code	Extension points, services, SPI
Application startup	Custom main method	camel-main, Quarkus, Spring Boot, CLI

stricter typing, promotes modular design, and adopts a more contemporary development model. It provides a structured approach to route configuration. Many legacy methods and patterns were deprecated or removed, requiring a complete refactoring of how we approach integration logic.

As described earlier in this section, significant new functionality was added and major architectural changes were introduced. This led to several issues with the current solution on the Onsemi platform. As shown in Figure 5.2, one of the most significant changes was the architectural redesign itself, for example, the modification of the CamelContext class. The complexity of the upgrade was further increased by the fact that the release notes primarily focused on listing new features or adjustments, without providing sufficient documentation for the underlying source code. Upgrading from version 1.8 to 4.8 was particularly challenging, as it involved tracking and adapting to nearly three years of development changes. The lack of detailed migration guides or backward compatibility support made it necessary to manually analyze and refactor large parts of the integration logic. In many cases, we had to reverse-engineer behaviors from the source code to ensure consistency with the previous implementation.

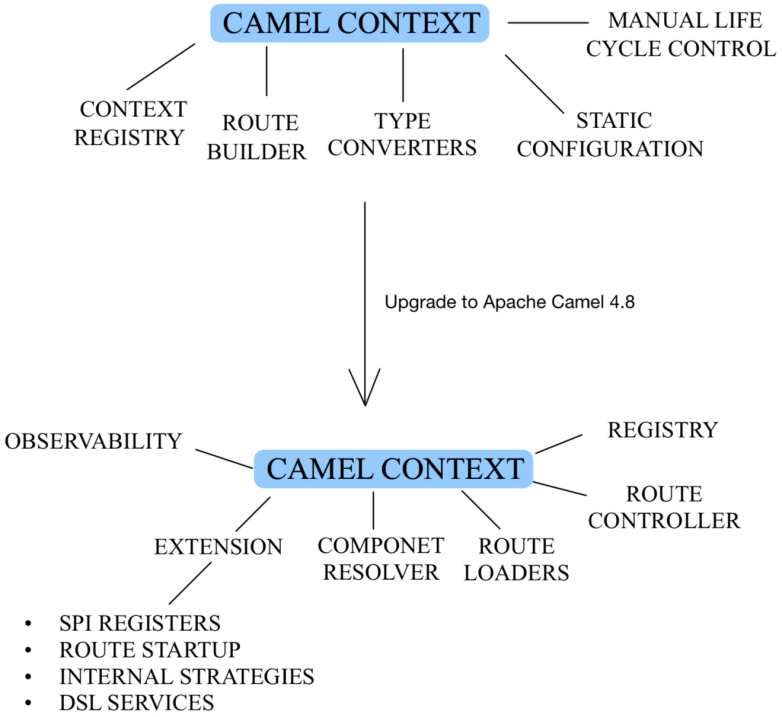

Figure 5.2: Comparison of the Camel context structure before and after the upgrade.

5.3 Custom Karavan Instance

In addition to the local development setup, we now maintain our own forked instance of Apache Camel Karavan, which is deployed on one of our internal servers dedicated to the development and testing of new features. This instance allows us to independently test and validate integration logic created through the Karavan UI without relying on external or shared environments.

To enable this functionality, a dedicated GitLab repository was created to serve as the storage backend for all generated YAML integration files [20]. A correctly configured Docker Compose setup was prepared for the Karavan instance, ensuring that all required services are launched with the appropriate settings. This includes mounting necessary volumes, defining environment variables, and assigning a

deploy key that grants secure access to the GitLab repository. This integration enables Karavan to automatically push all changes made via the UI directly into version control, supporting traceability, versioning, and collaboration across the team.

The connection with Git is essential, as Karavan relies on Git repositories as its primary storage mechanism for route definitions and project configurations. Without this integration, it would not be possible to persistently save changes, roll back to earlier versions, or conduct peer reviews of integration logic. In addition to backend integration, we also applied several frontend customizations to tailor the Karavan interface to our internal workflows. These adjustments included modifying or removing unnecessary UI elements, hiding default routes and template examples, and updating specific buttons and navigation options to focus exclusively on our custom integration components. As a result, the user experience is now more intuitive and streamlined.

The configuration file used to launch the instance is shown below. For security reasons, all sensitive credentials and repository URLs have been anonymized:

```
version: "3.7"

services:
  karavan:
    container_name: karavan
    image: registry.gitlab.com/<your-group>/<
       your-project>/camel-karavan-mdw:latest
    ports:
      - "8194:8080"
    volumes:
      - /var/run/docker.sock:/var/run/docker.
         sock
    environment:
      - DOCKER_HOST=unix:///var/run/docker.sock
      - KARAVAN_GIT_REPOSITORY=https://gitlab.
         com/<your-group>/<your-repo>.git
      - KARAVAN_GIT_USERNAME=<your-username>
      - KARAVAN_GIT_PASSWORD=<your-access-token>
      - KARAVAN_GIT_BRANCH=main
```

This configuration performs the following:

- Assigns a static container name (`karavan`).
- Uses a custom Karavan image from the GitLab Container Registry.
- Maps container port `8080` to host port `8194` for UI access.
- Mounts the host's Docker socket for internal Karavan operations.
- Defines Git repository credentials and target branch for storing YAML routes.

Figures 5.3, 5.4, and 5.5 show a slightly modified version of the Karavan graphical interface, which has been integrated into the system. These adjustments were made to improve usability and align the tool more closely with our existing workflows. In future phases, this graphical interface will serve as the basis for further user testing and feedback collection.

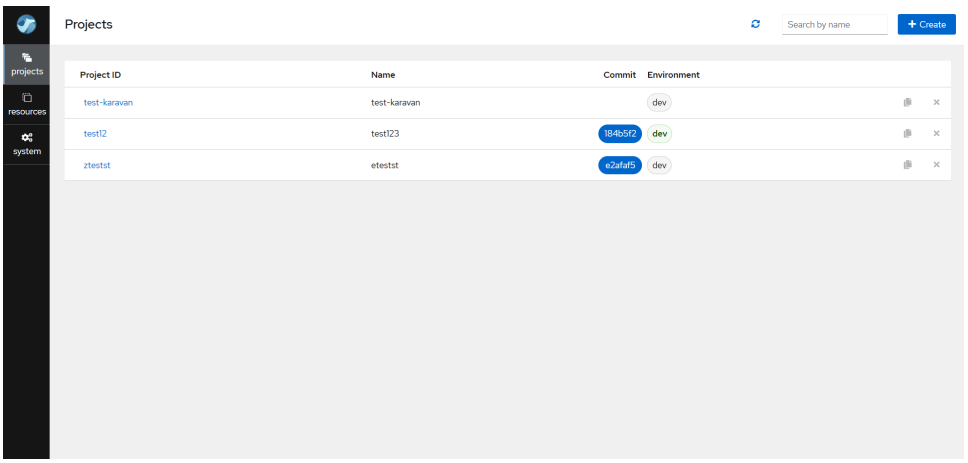

Figure 5.3: Main screen of the Karavan platform displaying available integration projects.

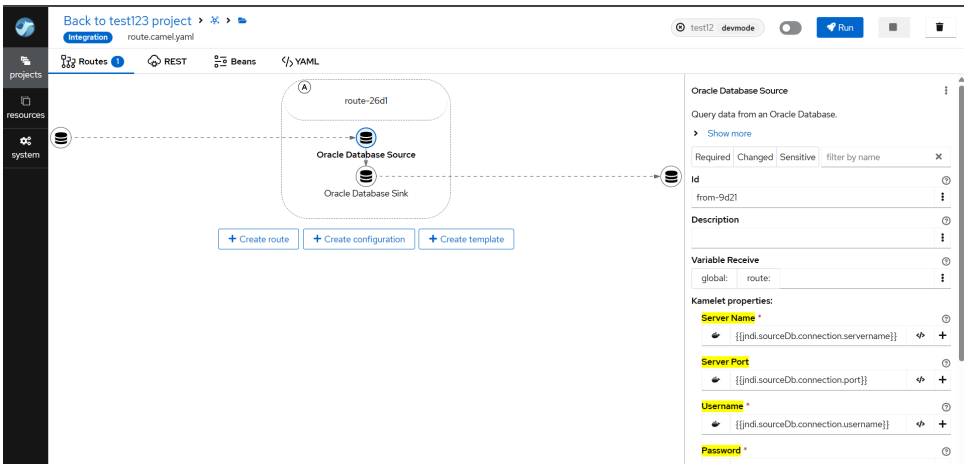

Figure 5.4: Opened integration route in Karavan, with the configuration panel displayed on the right side.

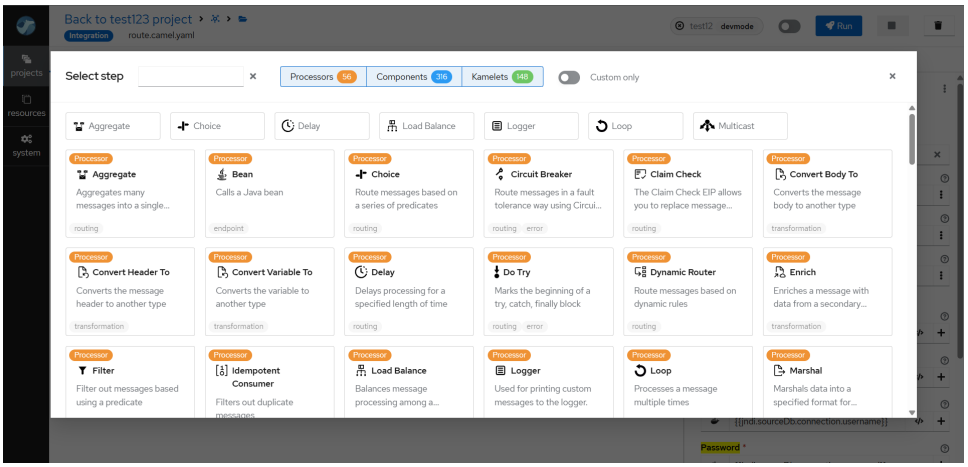

Figure 5.5: Modular panel showing available objects that can be added to the route.

5.4 Route loading

With the introduction of the customized Karavan instance and Git-based integration storage, the process of loading and executing routes has been streamlined and made more modular. Instead of manually

implementing Camel routes in Java or XML, the integration flow is now defined as a YAML file using the Karavan UI and stored in the designated Git repository. Comparison between YAML representation and visualized route is displayed in Figure 5.6 and 5.7.

These YAML files are version-controlled and synchronized with the local development environment, where they can be further processed. As part of our platform, we are developing a custom route loader component, which dynamically loads YAML route definitions and constructs the corresponding Camel routes.

The core of this mechanism is a custom implementation of the `RouteBuilder` class. This builder is responsible for locating YAML files in a predefined directory (e.g., `/resources/routes`), reading their contents, parsing them, and transforming them into executable route definitions at runtime. The parsing is typically handled using the `CamelYamlDsl` component or other compatible YAML parser integrated with the Camel runtime.

During the application startup, the builder scans the directory for all available YAML files. For each file, it loads the content, validates its structure, and constructs a corresponding in-memory route representation. Once validated, the route is programmatically registered into the Apache Camel context using standard Camel APIs. This dynamic registration allows full flexibility and decouples route implementation from code deployment.

The following simplified steps illustrate the lifecycle:

1. Route is created in Karavan and exported as a YAML file.
2. The YAML file is pushed to a Git repository.
3. Local environment pulls the file and stores it in a designated folder.
4. A custom `YamlRouteLoader` reads the file from disk.
5. The YAML content is parsed and converted into a Camel route.
6. The route is registered into the Camel context.
7. Once registered, the route is active and starts processing exchanges.

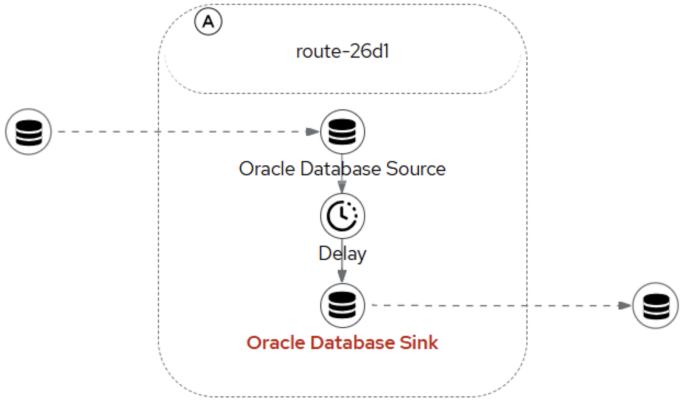

Figure 5.6: Example of simple Oracle database integration route

```yaml
- route:
    id: route-26d1
    nodePrefixId: route-93e
    from:
      id: from-9d21
      uri: kamelet:oracle-database-source
      parameters:
        serverName: server_name
        serverPort: 9999
        password: "{{group.property}}"
        query: QUERY
        databaseName: db_name
        username: "{{jndi.sourcedb.username}}"
      steps:
        - delay:
            id: delay-7170
            expression:
              groovy:
                id: groovy-d15f
                expression: "100"
        - to:
            id: to-106d
            uri: kamelet:oracle-database-sink
            parameters:
              serverName: sv_name
              username: ADS
              password: "{{props.pacleholder}}"
              query: QUERY
              databaseName: db_name
```

Figure 5.7: YAML representation of the route shown in Figure 5.6

5.5 Handling of Configuration Data via Placeholder Injection

One of the biggest challenges when integrating Apache Camel Karavan into the existing system was securely handling sensitive configuration data such as server credentials or access parameters. Key requirement from team was to avoid major modifications to Karavan instance itself, which limited the range of possible solutions. At the same time, it was essential to ensure that this sensitive information would not be exposed directly in the route definitions or committed to the shared Git repository, especially given the strict security constraints.

To address this problem we adopted a dynamic placeholder mechanism that is shown in Figure 5.8. This mechanism is based on external property files. Every integration project in the Karavan repository has one or more property files, which initially contain placeholder keys populated with mock values. During synchronization, before loading the YAML route, properties are loaded into a map and then dynamically replaced. These resolved properties are then registered into the CamelContext as a PropertiesComponent. At runtime, when Karavan loads and executes the route definitions it automatically replaces the placeholders with the actual injected values. This allows the routes to operate with correct configuration data without requiring any changes in the Karavan interface or exposing credentials in source files. These changes can be made through existing dashboard microservice, where you can edit the connection details with all necessary information and then design the logic in Karavan's graphical interface. This approach ensure seamless integration with the broader system.

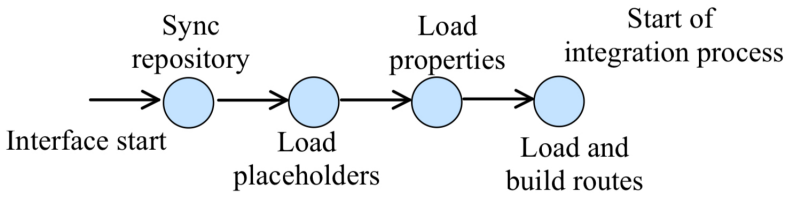

Figure 5.8: Backend processes before starting of the integration.

6 Testing and Optimization

In the development of integration platforms, it is essential to ensure reliability, efficiency, and scalability of integrations and processes. Given the complexity of connecting various systems and services, continuous testing and optimization are critical to maintaining system integrity and performance. For Onsemi and their middleware platform, having reliable data is crucial for day-to-day decision-making.

With the addition of a visual interface for creating integrations (Karavan), the design and deployment of integration routes became much simpler. This led to significant improvements in efficiency. However, due to the dynamic nature of integration scenarios, a robust testing strategy is necessary to validate the correctness of routes and their interactions with external systems. Additionally, as the volume and complexity of integrations increase, optimization becomes key to ensuring the platform remains resource-efficient.

This chapter explores the methods and best practices used to test and optimize our implementation. It covers the strategies employed to validate the correctness of integrations created through the GUI, the tools and frameworks utilized, and the optimization techniques implemented to improve performance, scalability, and usability.

6.1 Static Analysis and Runtime Testing During Upgrade

Upgrading to a newer version of Apache Camel is a critical process that can involve numerous challenges due to API changes, deprecated features, or shifts in architectural design. To ensure a smooth transition and preserve the integrity of integration flows, it was essential to adopt a well-defined strategy. This involved leveraging existing unit tests and updating them to be compatible with the new version, conducting static code analysis to detect potential issues, and performing local runs of integrations alongside the updated dependent projects. Together, these steps formed our approach that ensured the upgrade of Apache Camel was stable, reliable, and thoroughly tested.

- **Static Code Analysis** – During this phase, static code analysis played a crucial role, especially since we upgraded Apache Camel from version 1.8 to 4.8. With such a major version jump, many of our existing usages were outdated, deprecated, or removed entirely. Static analysis was the first layer of support, highlighting changes we had not yet addressed. By resolving these issues early, we mitigated the risk of runtime failures and ensured compatibility with the updated framework.

- **Leveraging Existing Unit Tests** – Before initiating the upgrade, we executed our existing unit tests, which provided solid coverage of the core functionalities used in our platform's runtime, particularly in the integration processes. These tests offered immediate feedback on the impact of the upgrade, enabling us to catch and address issues that were not evident during the initial implementation changes.

- **Runtime Testing with Custom Test Interface** – While unit tests and static analysis are effective for detecting structural and syntax-related problems, they cannot fully capture runtime behavior or system interactions. To bridge this gap, we developed a custom test interface capable of delivering data across systems, simulating real execution scenarios. This allowed us to detect and resolve unexpected behaviors before they could affect the production environment.

6.2 User-Centered Testing of the User Interface

To tailor the visual integration tool to the specific needs of our platform, the open-source Karavan project was forked from its public repository. This allowed for direct modification of the user interface to better align with our security requirements and platform capabilities.

The primary objective of these customizations was to limit access to advanced features and configuration options that are irrelevant or potentially harmful in the context of our deployment, especially for users with limited experience in integration design. To ensure these modifications were not only technically correct but also usable and intuitive, a structured testing process was conducted. During

the first use of the customized version, it was important to test how easy the interface is for people without a technical background. One of our non-technical employees helped us understand what regular users expect from the app. Thanks to this, we found out that users who would usually contact us to provide connection details or ask for help were able to create basic integration logic on their own. In many cases, they managed to do it successfully on the first try using the available blocks. The testing confirmed that even non-technical users were able to independently design integration routes, emphasizing the importance of user-centered design in building intuitive and accessible software interfaces [21].

6.3 Test-Driven Development

Test-Driven Development (TDD) is a software development strategy where tests are written before the actual implementation of features [22]. This process follows a short and iterative cycle. First, a failing test is created to define a desired improvement or new function. Then the amount of code is written to make the test pass, next is refactoring to better structure the code and ensuring the rest of the tests pass too.

This approach encourages better design, improves code quality, and ensures that the system behaves as expected from the beginning of development to the final deployment

6.4 Unit Testing in Backend of System

Another key phase in the development and testing process was the validation of the backend logic responsible for processing defined integrations written in YAML format. A typical use case begins with the user creating a flow in Karavan, which generates the corresponding YAML file. This file is then loaded into the runtime engine at the start of the integration, where it is parsed, validated, and executed.

To ensure the reliability and correctness of this implementation, we adopted a TDD approach. For each new feature or change in functionality, the corresponding test was written first. The implementation was then developed incrementally until all tests passed and the expected behavior was achieved.

6.4.1 Parsing and Deploying Yaml Routes

The initial step was to verify that the system was capable of loading a basic YAML file, converting it into a Camel route, and injecting it into the Camel Context where it would run automatically. A custom YAML file was created with mocked endpoints to simulate message routing without requiring actual connections or complex logic that might be incorrectly designed. The test confirmed that the route was successfully loaded, the structure matched expectations, and the mocked endpoints received the intended messages.

6.4.2 Feature Development Testing

As the platform evolved, the processing logic also changed. It moved towards a more secure and user-friendly model. This model supports configuration using placeholders and allows values to be provided dynamically from the backend.

This became the next key feature to develop. The implementation followed a TDD approach.

A set of comprehensive tests was created to verify the following behavior:

A comprehensive set of tests was implemented to ensure the following:

1. Property files are correctly loaded and parsed.
2. All placeholders in the YAML files are accurately replaced with the corresponding values.
3. The resulting routes preserve the correct structure and expected behavior.
4. If some routes fail, the rest continue to function without interruption.
5. Proper error handling is triggered in cases of malformed properties.

6.5 Optimization

The primary focus of this thesis was the optimization of user experience and integration efficiency across the platform. By introducing Apache Camel Karavan as a visual interface and simplifying the process of creating and deploying integration routes and solution significantly reduces complexity traditionally associated with backend integration development.

As a part of implementation, several optimization were introduced to ensure efficient performance and readiness for future scaling. The separated container of the Karavan app allows for independent deployment and resource allocation, reducing the risk of resource consumption and simplifying testing and development.

Although the solution currently runs in a lightweight Docker environment, it was designed with future scalability in mind. In the next phase, it can be seamlessly deployed into Kubernetes cluster which enables better orchestration, fault tolerance, and autoscaling capabilities as platform usage grows.

7 Benefits and Evaluation

The purpose of this chapter is to analyze the benefits, limitations, and potential future improvements of the Apache Camel Karavan implementation on our platform. It highlights the advantages this solution brings to developers, operations staff, the organization as a whole, and the end users. The implementations goal is to improves productivity, maintainability, and route clarity. It also examines how the tool enhances collaboration with the middleware team. The chapter then presents the results of a structured evaluation, conducted through surveys.

The implementation of Apache Camel Karavan on the middleware platform—responsible for providing data for day-to-day decision-making—is a strategically driven step toward modernizing and streamlining the organization's integration design process. As Onsemi expands its digital ecosystem, the complexity of system communication, data orchestration, and automation also grows. Therefore, improving efficiency across processes becomes essential. Since the integration layer serves as the backbone that connects internal systems with external partners—ensuring real-time data flow, operational continuity, and scalability—it is crucial to make the creation and management of integrations as simple and accessible as possible. With Onsemi's continued growth, inefficiencies and bottlenecks must be eliminated. That is why introducing a graphical interface for integration design was included among the organization's strategic goals for 2025. Reducing the need for excessive communication, requirement meetings, and manual coordination—especially for more complex integrations—contributes significantly to improving overall productivity.

Historically, the development and maintenance of custom interfaces at Onsemi relied heavily on manual coding or the use of multiple generic interfaces, which were less intuitive than creating tailored ones. With the introduction of Apache Camel DSL, the platform gained flexibility, but it remained primarily developer-oriented. This brought new challenges, such as a steep learning curve, limited visibility for business users, and a higher risk of inconsistencies or code duplication. Additionally, the absence of a visual tool made collaboration between

technical and non-technical stakeholders difficult, often leading to miscommunication and misaligned expectations.

Apache Camel Karavan implementation provides a visual editor and designer for Apache Camel routes, offering a user-friendly interface with YAML-based definitions stored in a single repository. The main goal of Karavan is to allow users to design their own integrations without needing assistance from administrators for design or deployment. This aligns with the middleware team's strategy to deliver a SaaS plwiatform supported by a small operations team that can assist less technically skilled users. And separate team will handle more complex tasks. This approach aims to increase overall team efficiency and reduce the need for highly skilled personnel to handle both development and support roles.

At the time of writing, the implementation is in the beta testing phase. This means the primary users are developers themselves, along with a few other technically capable workers. A small group of selected users from this group were asked about their opinions, problems, and overall feelings about the application. Feedback from users who were not part of the development team or involved in implementation meetings is especially valuable, as it allows us to evaluate the real benefits and identify issues. Although the optimization process will continue even after the end of this thesis, this feedback provides a strong foundation for further improvements.

7.1 Benefits for the Onsemi Middleware Team

The integration of Apache Camel Karavan into the platform and integration design workflow brings to several benefits even in its current beta phase. These include faster implementation of integration flows, improved code readability, and more efficient team collaboration. However, with increased efficiency, there are also potential risks that are important to acknowledge.

One of the most significant benefits is the help with acceleration of the integration design process by enabling users to create their integrations through a graphical interface. In the future, this could eliminate the time users currently spend communicating details, preparing test data, and coordinating testing with administrators.

Another benefit is improved readability and maintainability of integration routes. Unlike traditional code-based approaches, the visual format makes even complex routes much easier to understand. Additionally, there is a lower barrier to entry for developers who are new to integrations, which can lead to shorter onboarding times and a smoother learning curve.

Despite these benefits, it is important to also consider potential risks. With integration design becoming more accessible and the process more streamlined, there is a possibility that the company may consider restructuring the team or even outsourcing, especially if the design process becomes more efficient. In such a scenario, maintaining a medium-sized development team could become too costly for support and related tasks.

7.2 Benefits for Business Users and Integration Stakeholders

Although Karavan was designed and implemented primarily to support developers, it also offers significant advantages for business users. These users previously had limited insight into the technical implementation and relied on developers to translate their requirements into route designs. This changes with the introduction of a visual interface, which not only improves understanding but also enhances communication and reduces delays. The added transparency speeds up feedback loops and empowers more technically inclined stakeholders to take a proactive role, potentially even prototyping flows themselves. This is especially important in a global organization where the business is based in the United States and the middleware team operates from the Czech Republic and the difference in time zones can slow down communication. Therefore, the ability for users to work independently and prototype their own solutions before involving developers increases autonomy in integration design.

7.3 Summary of Surveys Results

To assess the usability and practical value of the Karavan platform in real-world conditions, a user evaluation was conducted during the

beta testing phase. The focus was on gathering qualitative feedback from internal users who had varying levels of technical expertise. The primary goal of evaluation was to assess how easily both technical and non-technical users can work with the Karavan platform to create integrations independently. A secondary focus was to identify any limitations or missing features that could hinder broader adoption within the organization.

To evaluate the impact of Karavan during this phase, it was not feasible to use a survey with closed questions due to the limited number of users—currently 7 active users, all of those are developers. Instead, we used open-ended questions and included feedback from middleware team members and one business user. Their responses provided valuable insights into the perceived benefits, limitations, and expected future features.

The feedback was collected from four internal users of the Karavan platform. Three of them were middleware developers and one was a member of the operations team responsible for external projects (non-technical user). All participants received a brief introduction to the Karavan processes and graphical user interface. Afterwards, they were given time (15 minutes) to explore the platform independently. Finally, they were asked to provide written responses to a set of open-ended questions.

The following open-ended questions were used to collect feedback:

1. What do you think about the addition of a visual designer to the platform environment?
   (Do you find it useful or beneficial in your workflow?)

2. How would you rate the user interface of Karavan?
   Did you find it intuitive, or did you experience any difficulties while using it?

3. How was your experience working with the graphical designer for integrations?
   Was it clear and easy to understand?

4. Are there any features you feel are missing in Karavan?
   What improvements or additions would you like to see in the future?

5. Are there any Kamelets you feel are missing that would help you create data integrations more efficiently?
6. If you have previously worked with Camel DSL in another format (e.g., XML or Java), how was the transition to Karavan's visual designer for you?
7. Have you experienced any performance or stability issues while using Apache Camel Karavan?
8. Would you recommend using Karavan to your customers? Yes / No – Please explain why.

General feedback was largely positive. The visual designer was appreciated by the non-technical user, who liked the opportunity to become less dependent on developers. The graphical user interface was seen as clean, intuitive, and modern. However, two out of three developers pointed out the need for training materials and manuals to be created, because although it is simpler than coding, it is not easy enough for users to create complex routes. For beginners, creating integrations was easy at first, but over time it became less intuitive and required more guidance. Suggestions included features such as the ability to clone integrations (which is already possible by simply copying the YAML file and pasting it into a new project) and better visibility of connections (which is currently not possible, as the goal is to keep the Karavan instance as close to the original as possible). None of the users mentioned missing Kamelets or expressed a desire to add them, though some noted that if Karavan gains wider adoption, specific Kamelets might be useful for certain projects. Due to beta testing phase and the low number of users at a time, performance has not been an issue. However, half of the respondents stated that the feature is not yet ready for full rollout. Instead, they are using it for integration creation while waiting for manuals and training to be completed to raise adoption.

7.4 Current Limitations and Future Challenges

The recent survey results indicate that the initial implementation of Apache Camel Karavan has been received mostly positively by users.

A particularly appreciated feature is the ability to design and modify integrations without direct developer involvement. Many users described the visual designer as modern and intuitive, making it accessible even to those with low technical backgrounds.

However, early user testing has also uncover several limitations and challenges that were not identified during initial development phase. This section will explore those issues in detail and outline how they may be addressed in the future, as this thesis serves as the foundational step in the broader restructuring of the middleware team's workflows.

One of the first suggestions received was to implement functionality for cloning existing integrations and to improve the visibility of connections within Karavan's graphical user interface. Currently, these features are not available due to the original assignment, which required minimal changes to the core Karavan instance. If the scope is later adjusted and management supports extending Karavan's capabilities, these features will be among the top priorities for future development.

The most significant limitation identified is the learning curve associated with using Karavan effectively. Users need time and guidance to become familiar with the graphical interface and to understand how to work with components, Kamelets, and configurations. Ensuring that all potential users receive adequate training will be one of the most critical follow-up tasks for the success of this project.

Fortunately, Onsemi has a big library of internal training videos, manuals, and a comprehensive Confluence knowledge base. However, a substantial effort will be required to create Karavan-specific educational content, including tutorials, examples, and best practices tailored to our internal processes. This material will be essential in enabling broader and faster adoption of Karavan and ensuring effective use across the organization.

8 Conclusion

This thesis aims to design and implement a modern solution for user-friendly graphical interface development by integrating Apache Camel Karavan into Onsemi's middleware platform. The motivation behind this initiative is the growing need to make the process of designing complex integrations more accessible, efficient, and standardized, while also empowering users to work more independently. As the result, this project has become a strategic organizational goal for 2025. By simplifying the creation and configuration of integrations through graphical user interface, the solution is expected to reduce workload, contribute to overall organizational efficiency and optimize both processes and individual productivity.

A detailed analysis of the current state of the platform revealed some key limitations. The most significant issue was a strong dependency on developers for the creation of integrations. Communication delays, often caused by time zone differences, leaded bottlenecks in the development process. Additional challenges included the lack of visualization tools for designing and configurating routes, as well as various inefficiencies, such as unclear or changing requirements. These problems were further exacerbated by absence of standardization in integration design, redundant implementations, and steep learning curve for newcomers due to the high level of task abstraction. Collectively, these factors led to limited accessibility for working with the platform and made it virtually impossible to develop integrations containing logic outside the middleware team.

To address the challenges identified during the analysis, this work proposed and implemented a tailored solution. The integration of Apache Camel Karavan offers robust and scalable solution for platform's limitations. The first and most challenging step was upgrading Apache Camel. This was an essential move, as the platform was previously running version 1.8, while Karavan requires at least version 4.8. Over three years of Camel development introduced significant architectural changes that had to be solved during the upgrade. The next step involved deploying a dedicated instance of Karavan and customizing it for our specific use case. This included removing unnecessary UI elements and configuring own repository for route definitions.

Deployment was prepared using Docker to simplify testing and to enable easier future scalability with Kubernetes. Yaml based routes were used to support dynamic loading and execution.

Throughout the beta phase, early testing showed promising results. Even users with only basic integration knowledge were able to create default integrations without manual assistance or extensive training. Developers reported faster prototyping and clearer logic thanks to the graphical representation of routes, while business users gained valuable visibility into the integration design. User feedback, gathered from questionnaires, confirmed that Karavan not only reduces overhead for the middleware team but also improves user autonomy and development speed. The conclusion of the surveys was that the implementation itself is just the first step in a corporate environment, and user adoption is a much longer journey.

In conclusion, this work demonstrates that the thoughtful implementation of a visual integration tool like Apache Camel Karavan, when broadly adopted, can significantly enhance the efficiency, accessibility, and scalability of a middleware integration platform. This solution could become a key component in the restructuring of middleware team workflows and enable more efficient use of resources. It meets all the objectives defined within the project scope, from technical feasibility and architectural integration to business usability, long-term scalability, and maintainability. It represents not only a technical enhancement but also a strategic step forward—marking a shift that supports organizational growth in the new digital era.

Bibliography

1. GAZIS, Alexandros; KATSIRI, Eleftheria. Middleware 101: What to know now and for the future. *Queue*. 2022, vol. 20, no. 1, pp. 10–23. ISSN 1542-7730. Available from DOI: 10.1145/3526211.
2. RED HAT. *Application connectivity in hybrid cloud environments*. 2023. Available also from: https://www.redhat.com/en/blog/application-connectivity-hybrid-cloud. Accessed: 2025-05-07.
3. IBM. *What is Middleware?* 2023. Available also from: https://www.ibm.com/think/topics/middleware. Accessed: 2025-05-07.
4. LARRUCEA, Xabier; SANTAMARIA, Izaskun; COLOMO-PALACIOS, Ricardo; EBERT, Christof. Microservices. *IEEE Software*. 2018, vol. 35. ISSN 07407459. Available from DOI: 10.1109/MS.2018.2141030.
5. SOFTWARE, Fiorano. *The Role of Middleware in the Cloud-Native Ecosystem*. 2022. Available also from: https://www.fiorano.com/blogs/The_Role_of_Middleware_in_the_Cloud_Native_Ecosystem. Accessed: 2025-05-07.
6. DAS, Suman Saurabh. *Kubernetes: What's, Why's and How's of Kubernetes with Demonstration*. 2022. Available also from: https://dev.to/ssd/kubernetes-whats-whys-and-hows-of-kubernetes-with-demonstration-28dn. Accessed: 2025-05-14.
7. ALLAM, Karthik; ANKAM, Madhu; NALMALA, Manohar. Cloud Data Warehousing: How Snowflake Is Transforming Big Data Management. *International Journal of Computer Engineering and Technology (IJCET)*. 2023, vol. 14. ISSN 0976-6375.
8. CAMPOSO, Guilherme. *Cloud native integration with Apache Camel: Building agile and scalable integrations for Kubernetes platforms*. 2021. Available from DOI: 10.1007/978-1-4842-7211-4.
9. CAMEL, Apache. *Apache Camel Karavan Documentation*. 2024. Available also from: https://camel.apache.org/camel-karavan/latest/index.html. Accessed: 2025-05-06.

10. *Intelligent Power and Sensing Technologies | onsemi*. [N.d.]. Available also from: https://www.onsemi.com/.

11. CONCEPTA TECHNOLOGIES. *How to Define Stakeholders for Your Software Development Project*. 2023. Available also from: https://www.conceptatech.com/blog/how-to-define-stakeholders-for-your-software-development-project. Accessed: 2025-05-07.

12. OPENSOURCE.COM. *What is Open Source?* 2025. Available also from: https://opensource.com/resources/what-open-source. Accessed: 2025-05-04.

13. SOLTECH. *The Importance of Clear Software Requirements for Project Success*. 2024. Available also from: https://soltech.net/business-objectives-not-software-requirements/. Accessed: 2025-05-06.

14. FOUNDATION, Apache Software. *Apache Camel Karavan - Git Integration Guide*. 2024. Available also from: https://camel.apache.org/camel-karavan/latest/git-integration.html. Accessed: 2025-05-06.

15. HASHICORP. *Consul by HashiCorp - Service Discovery and Configuration*. 2025. Available also from: https://www.consul.io/docs. Accessed: 2025-05-06.

16. THE CTO CLUB. *Data Integration Architecture: A Comprehensive Guide*. 2025. Available also from: https://thectoclub.com/topics/data-integration-architecture/. Accessed: 2025-05-07.

17. CYBERINSIDER. *What is a VPN? Comprehensive Guide to VPNs and Privacy*. 2025. Available also from: https://cyberinsider.com/vpn/what-is-a-vpn/. Accessed: 2025-05-04.

18. GRAFANA LABS. *Grafana: The open and composable observability platform*. 2025. Available also from: https://grafana.com/. Accessed: 2025-05-07.

19. BURNS, Brendan; BEDA, Joe; HIGHTOWER, Kelsey. *Kubernetes: Up and Running: Dive into the Future of Infrastructure*. 3rd. O'Reilly Media, 2021. ISBN 9781492046530.

20. GITLAB. *Using GitLab as Configuration Storage for Visual Editors*. 2024. Available also from: `https://docs.gitlab.com/ee/topics/gitops/`. Accessed: 2025-05-06.

21. PEERBITS. *User-Centered Design: Why It's Essential for Software Development*. 2025. Available also from: `https://www.peerbits.com/blog/importance-of-user-centered-design-in-software-development.html`.

22. JANZEN, David; SAIEDIAN, Hossein. Test-driven development: Concepts, taxonomy, and future direction. *Computer*. 2005, vol. 38, no. 9, pp. 43–50.