FEDERAL UNIVERSITY OF SANTA CATARINA
TECHNOLOGY CENTER
AUTOMATION AND SYSTEMS DEPARTMENT
UNDERGRADUATE COURSE IN CONTROL AND AUTOMATION ENGINEERING

Lukas Alberto Belck

Development of an Application for Reusable Supply Chain Mapping

Berlin
2024

Lukas Alberto Belck

Development of an Application for Reusable Supply Chain Mapping

Final report of the subject DAS5511 (Course Final Project) as a Concluding Dissertation of the Undergraduate Course in Control and Automation Engineering of the Federal University of Santa Catarina.
Supervisor: Prof. Leandro Buss Becker, Dr.

Berlin
2024

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Belck, Lukas Alberto
Development of an application for reusable supply chain mapping / Lukas Alberto Belck ; orientador, Leandro Buss Becker, 2024.
64 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia de Controle e Automação, Florianópolis, 2024.

Inclui referências.

1. Engenharia de Controle e Automação. 2. Mapeamento da Cadeia de Suprimentos. 3. API GraphQL. 4. Design de Banco de Dados SQL. 5. Desenvolvimento Web. I. Becker, Leandro Buss. II. Universidade Federal de Santa Catarina. Graduação em Engenharia de Controle e Automação. III. Título.

Lukas Alberto Belck

Development of an Application for Reusable Supply Chain Mapping

This dissertation was evaluated in the context of the subject DAS5511 (Course Final Project) and approved in its final form by the Undergraduate Course in Control and Automation Engineering

Florianópolis, June 01, 2024.

Prof. Marcelo de Lellis Costa de Oliveira , Dr.
Course Coordinator
UFSC/CTC/DAS

Examining Board:

Documento assinado digitalmente
Leandro Buss Becker
Data: 23/07/2024 21:53:44-0300
CPF: ***.357.870-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Leandro Buss Becker, Dr.
Advisor
UFSC/CTC/DAS

Ana Selina Haberbosch
Supervisor
seedtrace GmbH

[illegible]

Prof. Eduardo Rauh Müller, Dr.
Evaluator
UFSC/CTC/DAS

Prof. Eduardo Camponogara, Dr.
Board President
UFSC/CTC/DAS

This work is dedicated to my grandmother Vera.

ACKNOWLEDGEMENTS

I dedicate this work to my grandmother Vera, who gave me all the love and support I needed to achieve this goal. I miss you every day. To my parents Alexander and Fabiola, who, despite the distance, have always been with me. For all the teaching, nudging and love I received, thank you very much. Dear NEO colleagues - thank you for believing in me, thank you for all the opportunities, but above all, thank you for the time we spent together. To the friends I made at UFSC along this journey - it was a great honor to walk this path with you. I'll never forget those moments. See you soon.

DISCLAIMER

Berlin, June 28-th, 2024.

As representative of seedtrace GmbH in which the present work was carried out, I declare this document to be exempt from any confidential or sensitive content regarding intellectual property, that may keep it from being published by the Federal University of Santa Catarina (UFSC) to the general public, including its online availability in the Institutional Repository of the University Library (BU). Furthermore, I attest knowledge of the obligation by the author, as a student of UFSC, to deposit this document in the said Institutional Repository, for being it a Final Program Dissertation (*"Trabalho de Conclusão de Curso"*), in accordance with the *Resolução Normativa n$^\circ$126/2019/CUn*.

[illegible]

Lukas Alberto Belck
seedtrace GmbH

[illegible]

Ana Selina Haberbosch
seedtrace GmbH

ABSTRACT

In the dynamic field of supply chain management, the demand for adaptable, transparent, and efficient mapping systems is becoming ever more crucial for companies across industries. The use or development of digital platforms to enhance the visibility and management of supply chains has become a priority for businesses seeking to optimize their operations and comply with both national and supra-national legislation. This thesis addresses these needs by developing a solution that introduces the concept of reusable chains in a digital supply chain management platform.

While existing solutions for mapping supply chains are available, feedback from users highlighted a significant limitation: the absence of a feature to create reusable chains. This limitation became apparent as businesses increasingly sought to enhance operational efficiency and sustainability by optimizing the mapping of components common to multiple products.

The core of this thesis is the creation of a new feature within a digital supply chain management platform that enables the use of reusable 'Component Chains'. This feature allows users to map the journey of individual ingredients of a product once and reuse them across multiple product supply chains, significantly enhancing operational efficiency and data consistency. The backend solution developed for this feature leverages GraphQL for API development and SQL for database design, ensuring robust support for the frontend application in creating and displaying these reusable chains.

Keywords: Supply Chain Management. GraphQL API. SQL Database Design.Sustainability. Transparency. Modular Mapping, Digital Platform. Web Development. Backend Development. Python.

RESUMO

No cenário em constante evolução da gestão da cadeia de suprimentos, a necessidade de sistemas de mapeamento mais adaptáveis, transparentes e eficientes é cada vez mais crítica. O desenvolvimento de soluções digitais para melhorar a visibilidade e a gestão das cadeias de suprimentos tornou-se uma prioridade para as empresas que buscam otimizar suas operações e cumprir a legislação nacional e supranacional. Esta tese aborda essas necessidades desenvolvendo uma solução que introduz o conceito de cadeias reutilizáveis em uma plataforma digital de gestão da cadeia de suprimentos.

Embora existam soluções existentes para mapear cadeias de suprimentos, o feedback dos usuários destacou uma limitação significativa: a ausência de um recurso para criar cadeias reutilizáveis. Esta limitação tornou-se evidente à medida que as empresas procuravam cada vez mais melhorar a eficiência operacional e a sustentabilidade, otimizando o mapeamento de componentes comuns a vários produtos.

O cerne desta tese é a criação de um novo recurso dentro de uma plataforma digital de gestão da cadeia de suprimentos que permite o uso de 'Cadeias de Componentes' reutilizáveis. Este recurso permite que os usuários mapeiem a jornada de ingredientes individuais de um produto uma vez e os reutilizem em várias cadeias de suprimentos de produtos, melhorando significativamente a eficiência operacional e a consistência dos dados. A solução de backend desenvolvida para este recurso utiliza GraphQL para o desenvolvimento de API e SQL para o design do banco de dados, garantindo suporte robusto para o aplicativo frontend na criação e exibição dessas cadeias reutilizáveis.

Palavras-chave: Gestão da Cadeia de Suprimentos. API GraphQL. Design de Banco de Dados SQL. Sustentabilidade. Transparência. Mapeamento Modular. Plataforma Digital. Desenvolvimento Web. Desenvolvimento Backend. Python.

LIST OF FIGURES

LISTINGS

LIST OF TABLES

CONTENTS

1 INTRODUCTION

In the evolving landscape of supply chain management, the need for more adaptable, transparent, and efficient mapping systems is increasingly critical. Businesses today are under pressure to optimize their operations and ensure compliance with both national and supra-national legislation. Additionally, there is a growing demand from consumers for transparency, as they seek detailed information about product ingredients, origins, and social/ecological impacts. This consumer-driven push towards sustainability and transparency has become a significant market trend, providing added value to businesses that can meet these demands.

According to various reports, consumers are highly concerned about production practices and packaging. In the food industry alone, 69% of consumers are very concerned about production, 59% research packaging, and yet 60% lack trust in certificates and quality labels. Furthermore, 75% of consumers desire more detailed information and are willing to switch suppliers to obtain it (RESEARCH, 2019; FMI; FOOD INTEGRITY, 2018; FMI; LABELINSIGHT, 2018).

To address these challenges, a digital platform was developed to focus on the management and communication of sustainability claims along global supply chains. This platform enables businesses to collect, analyze, and disclose relevant data while ensuring that reporting and compliance requirements are met. The platform facilitates a holistic approach to supply chain transparency, factoring sustainability criteria into every step along the chain. For confidential reasons, the company and platform names are not disclosed.

1.1 PROBLEM STATEMENT

Despite the advancements offered by the platform, feedback from clients highlighted a significant limitation: the absence of a feature to create reusable supply chain mappings. Businesses managing multiple products with common components faced redundancy and inefficiency, as the current system required each product's supply chain to be mapped individually. This limitation impeded operational efficiency and data consistency, posing challenges in the management of complex supply chains.

1.2 IMPORTANCE OF THE PROBLEM

The identified problem is crucial because it directly impacts the platform's ability to manage supply chains efficiently. The inability to reuse common components across different product supply chains results in increased workload, higher risk of errors, and inconsistent data management. Addressing this issue is essential for enhancing the platform's functionality and meeting the growing demand for more efficient supply chain solutions.

1.3 PROPOSED SOLUTION

To address this problem, a project was embarked upon to develop a new feature called ’Component Chains’. This feature allows the mapping of individual components once, enabling their reuse across multiple product supply chains. The introduction of ’Component Chains’ represents a pivotal evolution of the platform, directly addressing client needs and market trends.

1.4 RESULTS AND SIGNIFICANCE

The implementation of ’Component Chains’ has empowered clients to map their products more efficiently and effectively within the platform. Companies can now create a single, reusable map for common components, significantly reducing redundancy and streamlining the entire supply chain mapping process. This improvement not only enhances data consistency but also ensures that supply chain representations are more accurate and reliable.

Additionally, the enhanced functionality allows businesses to better communicate the intricacies of their supply chains to end consumers. By providing a clear, detailed view of the journey each product component takes, companies can build greater trust and transparency with their customers. This capability supports stronger compliance with sustainability and transparency regulations, reinforcing the credibility of their sustainability claims. Overall, the successful integration of ’Component Chains’ has markedly improved the user experience on the platform, making it a more powerful tool for supply chain management and consumer engagement.

1.5 CONTRIBUTIONS BY THE AUTHOR

The author was responsible for the backend development of the ’Component Chains’ feature. This involved designing and implementing the GraphQL API, creating the necessary SQL database schema, and automating the migration of existing supply chain data. The frontend development, including the user interface for visualizing and managing ’Component Chains’, was carried out by the frontend team, based on designs provided by the design team.

This thesis demonstrates the practical application of modern software development and data management methodologies to solve a critical problem in supply chain management.

1.6 OBJECTIVE

The primary objective of this thesis is to enhance the platform by developing a feature that allows for the creation and management of reusable ’Component Chains’. This feature aims to address the limitations of the current supply chain mapping system, which lacks the ability to efficiently handle common components shared across multiple products. By introducing ’Component Chains’, the platform will enable businesses to map the journey of individual components once and reuse this mapping across multiple product supply chains. This improvement is expected to significantly reduce redundancy, enhance data consistency, and improve operational efficiency.

To achieve this objective, the thesis focuses on the backend development necessary to support the new feature. This includes the design and implementation of a robust GraphQL Queries and Mutations for the frontend application to consume and display on the application and the restructuring of the database schema to accommodate reusable chains. The specific objectives of the project are:

1. **Develop a GraphQL API**: The API will facilitate the creation, management, and retrieval of ’Component Chains’, ensuring seamless integration with the existing supply chain mapping functionalities.

2. **Design SQL Tables**: New database tables will be created to store data on reusable ’Component Chains’ and their relationships with multiple supply chains and products.

3. **Automate Data Migration**: Existing supply chain data will be migrated to the new format, ensuring historical data integrity and supporting the new ’Component Chains’ concept.

4. **Enhance User Experience**: The platform’s user interface will be improved to offer detailed tools for visualizing and managing ’Component Chains’ within the context of supply chain mapping.

This project aligns with the growing demand for more adaptable, transparent, and efficient supply chain management systems, addressing both operational needs and compliance requirements. The successful implementation of this feature is expected to provide significant benefits to businesses by streamlining supply chain mapping processes and enhancing overall transparency and sustainability.

By achieving these objectives, the thesis aims to contribute to the field of supply chain management by providing a practical solution that leverages advanced technologies such as GraphQL and SQL to improve data management and operational efficiency.

1.7 MONOGRAPH'S STRUCTURE

This thesis is organized into six chapters, starting with the current chapter that introduces the project. Chapter 2 discusses the theoretical foundations essential for understanding the project's development. In Chapter 3, the project requirements, both functional and non-functional, are outlined along with a development plan; this chapter also presents the current state of the application. Chapter 4 describes the detailed implementation of the software solution, including the database and GraphQL schema, as well as code and system architecture. The project results and performance evaluation are presented in Chapter 5. The thesis concludes with Chapter 6, where final thoughts, project limitations, and future directions are discussed.

2 THEORETICAL BACKGROUND

In this chapter, we delve into the theoretical foundations underpinning the methodologies and technologies utilized in the thesis project. We explore essential concepts in supply chain mapping, Agile methodology, software development lifecycle practices, database technologies, API development, and various technologies critical to the project. Understanding these theoretical elements is crucial for comprehending the implementations discussed in the next chapters.

2.1 SUPPLY CHAIN MAPPING

Supply chain mapping in the context of this thesis refers to the process of documenting and visualizing the flow of materials, information, and financial transactions across different stages of the supply chain. This includes identifying all the entities involved, such as suppliers, manufacturers, and distributors, their activities, and understanding their relationships and dependencies. By mapping these elements, businesses can achieve greater transparency and efficiency, making it easier to manage complex supply chains and ensure compliance with sustainability standards. In this thesis, supply chain mapping is extensively used in chapter 3, where the current application state and its database structure are analyzed. The improvements proposed in section 3.2 leverage these concepts to enhance data reuse and component mapping.

2.2 AGILE METHODOLOGY

Agile development is based on iterative and incremental approaches, emphasizing flexibility, collaboration, and customer feedback. The core principles of Agile, as outlined in the Agile Manifesto, include valuing individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan (BECK et al., 2001). In the context of API development, Agile practices involve frequent iterations, continuous feedback, and adaptive planning. These practices help ensure that APIs are developed in a manner that meets the evolving needs of users and stakeholders. Techniques such as sprint planning, daily stand-ups, and retrospectives are commonly used to manage and improve the development process (SCHWABER; BEEDLE, 2002).

2.3 SOFTWARE DEVELOPMENT LIFECYCLE

Continuous Integration (CI) is a development practice where developers frequently integrate their code changes into a shared repository, usually several times a day. Each integration is automatically verified by automated tests to detect integration errors as quickly as possible. This practice helps maintain code quality and enables early detection of issues (DUVALL; MATYAS; GLOVER, 2007). Common tools used for CI include Jenkins, Travis CI, CircleCI, and GitLab CI, which provide automated build and testing capabilities to streamline the integration process. In this thesis, Terraform and GitLab CI are utilized to automate testing and validation tasks, as described in section 4.1.

Continuous Delivery (CD) extends the concept of CI by ensuring that code changes are automatically tested and prepared for release to production. Continuous Deployment goes a step further by automatically deploying every change that passes all stages of the production pipeline to the end-users. These practices aim to reduce the risk and time associated with releasing software updates (HUMBLE; FARLEY, 2010). Tools commonly used for CD include Jenkins, Spinnaker, AWS CodePipeline, and GitHub Actions, Terraform, GCP Cloud Build which facilitate the automated deployment and delivery of software updates, ensuring smooth and reliable releases. In this thesis, Gitlab CI, Terraform and GCP Cloud Build are used to manage the deployment pipeline, as detailed section 4.1.

2.4 DATABASE TECHNOLOGIES

SQL databases, or relational databases, use structured query language (SQL) for defining and manipulating data. They are based on a table-based schema and are ideal for applications requiring complex queries and transactional consistency. Popular SQL databases include PostgreSQL, MySQL, and Oracle (ELMASRI; NAVATHE, 2016). On the other hand, graph databases are designed to treat the relationships between data as equally important as the data itself. These databases use graph structures with nodes, edges, and properties to represent and store data. This makes them ideal for applications where understanding and navigating relationships is crucial, such as social networks, recommendation engines, and, importantly, supply chain mapping. Popular graph databases include Neo4j and Dgraph (PERRYMAN; BECHBERGER, 2020; JAIN, 2021). In this thesis, PostgreSQL is used as the primary database for managing the data. Section 5 mention graphs databases as Neo4j for future improvements.

2.5 API DEVELOPMENT

RESTful APIs are based on representational state transfer (REST) principles, which define a set of constraints for creating web services. These APIs use standard HTTP methods and are stateless, allowing for scalable and modular applications. RESTful APIs are widely used due to their simplicity and ease of integration (RICHARDSON; AMUNDSEN; RUBY, 2013). In contrast, GraphQL is a query language for APIs and a runtime for executing those queries. It allows clients to request only the data they need, making APIs more efficient and flexible. GraphQL was developed by Facebook and is increasingly being adopted for modern API development due to its capability to aggregate multiple sources into a single query (BYRON; BANKS, 2018). In this thesis, GraphQL is implemented for the backend application and is used to create the queries and mutation the frontend consumes, as discussed in chapter 6.

2.6 TECHNOLOGIES

Python is a high-level, interpreted programming language known for its readability and ease of use. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Python is widely used in web development, data analysis, artificial intelligence, and more (ZELLE, 2004). FastAPI is a modern, fast (high-performance), web framework for building APIs with Python 3.7+ based on standard Python type hints. It is designed to be easy to use and easy to deploy, with automatic interactive API documentation generated using Swagger and ReDoc (LUBANOVIC, 2023). PostgreSQL is an open-source relational database management system (RDBMS) known for its robustness, extensibility, and standards compliance. It supports both SQL and JSON for relational and non-relational queries, making it versatile for various applications (MOMJIAN, 2001). SQLAlchemy is a SQL toolkit and Object-Relational Mapping (ORM) library for Python. It provides a full suite of well-known enterprise-level persistence patterns, designed for efficient and high-performing database access (RAMM; BAYER; RHODES, 2011).

GraphQL, as previously mentioned, is a powerful query language for APIs. It allows clients to specify the structure of the response, improving efficiency and flexibility in data retrieval (BYRON; BANKS, 2018). Graphene is a Python library used to build GraphQL schemas/types fast and easily. It is fully featured with integrations for popular frameworks like Django and SQLAlchemy (AKBARY, 2015).

Git is a distributed version control system designed to handle everything from small to very large projects with speed and efficiency. It allows multiple developers to work on a project simultaneously without interfering with each other's work, making it a crucial tool for modern software development (CHACON; STRAUB, 2014).

Terraform is an infrastructure as code software tool that enables users to define and provision data center infrastructure using a high-level configuration language. It provides a consistent workflow for managing infrastructure across various service providers, allowing for the efficient and repeatable creation of cloud and on-premises resources (TERRAFORM. . . , n.d.). In this thesis, Terraform is used to automate the deployment and management of infrastructure, ensuring consistent environments across development, testing, and production stages, as discussed in section 4.1.

In this thesis, these technologies are used extensively across various stages: Python, FastAPI, and PostgreSQL form the core stack for developing and managing

APIs and database interactions; SQLAlchemy is used for ORM purposes, and Git is employed for version control and collaboration.

3 SYSTEM DESIGN AND REQUIREMENTS SPECIFICATION

In this chapter, it's presented the detailed system design and the requirements specification for the supply chain mapping application. We begin by examining the current state of the application, highlighting its capabilities and limitations. This analysis sets the stage for proposing enhancements aimed at addressing the identified challenges. Subsequently, the functional and non-functional requirements necessary for implementing the proposed solution are outlined, ensuring that the system meets both the user needs and performance expectations.

3.1 CURRENT APPLICATION STATE

The current state of the application being analyzed in this thesis is an existing supply chain mapping web application. This application enables companies to map their supply chains by detailing the steps and activities involved in the production and distribution of their products. The primary objective of this section is to describe the current capabilities and limitations of this application in the context of supply chain complexity and data reuse.

The existing application allows users to create and manage detailed supply chain maps for individual products. Each product in the application is associated with a series of steps and activities that represent its journey through the supply chain. Users can visualize the sequence of these steps and activities, providing a clear understanding of the supply chain's structure and flow. Due to confidential reasons, only mock designs of the application will be shown. Figure 1 shows a mock design of the current application state.

In this interface, a product can be mapped out with specific activities with location for each step in the supply chain. Each step is individually mapped and visualized, providing a comprehensive overview of the product's journey from raw material to finished product.

The application also includes functionality for creating new locations and activities within the product journey, as shown in figure 2. This interface allows users to specify the location, supply chain step, activity details, and incoming connections for a new location card. This process enhances the flexibility and customization of the supply chain mapping, enabling detailed and specific data entry for each step.

3.1.1 Database Structure

The current database structure supports the functionalities of the application by storing and managing the data necessary for supply chain mapping on each Product. This section outlines the main tables required to handle the information shown in the

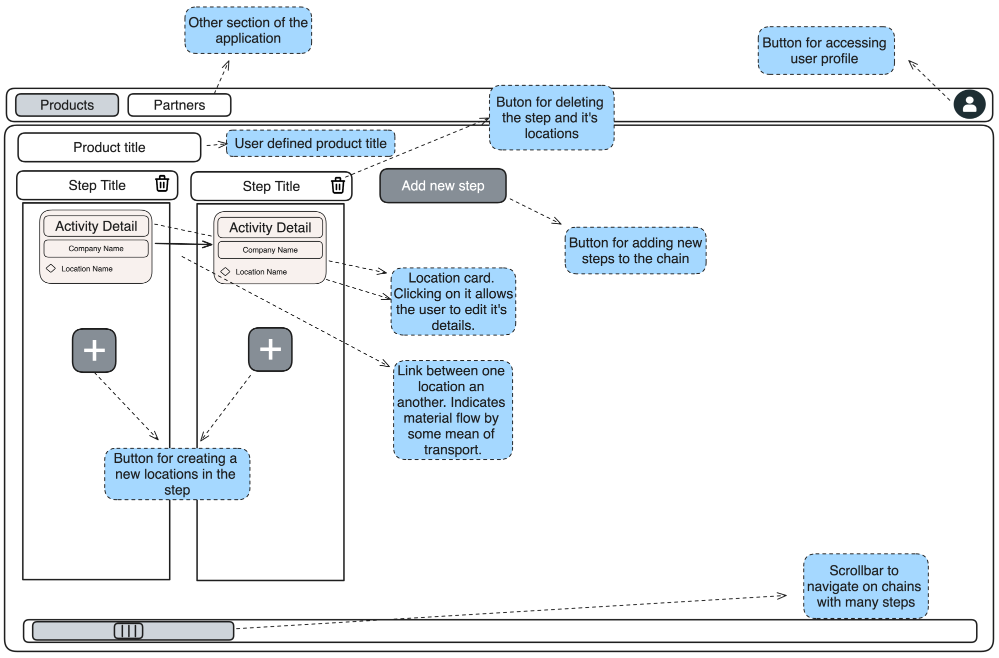
Other section of the application
Button for accessing user profile
Products
Partners
Buton for deleting the step and it's locations
Product title
User defined product title
Step Title
Step Title
Add new step
Activity Detail
Company Name
Location Name
Activity Detail
Company Name
Location Name
Button for adding new steps to the chain
Location card. Clicking on it allows the user to edit it's details.
Link between one location an another. Indicates material flow by some mean of transport.
Button for creating a new locations in the step
Scrollbar to navigate on chains with many steps

Figure 1 – Current Product journey

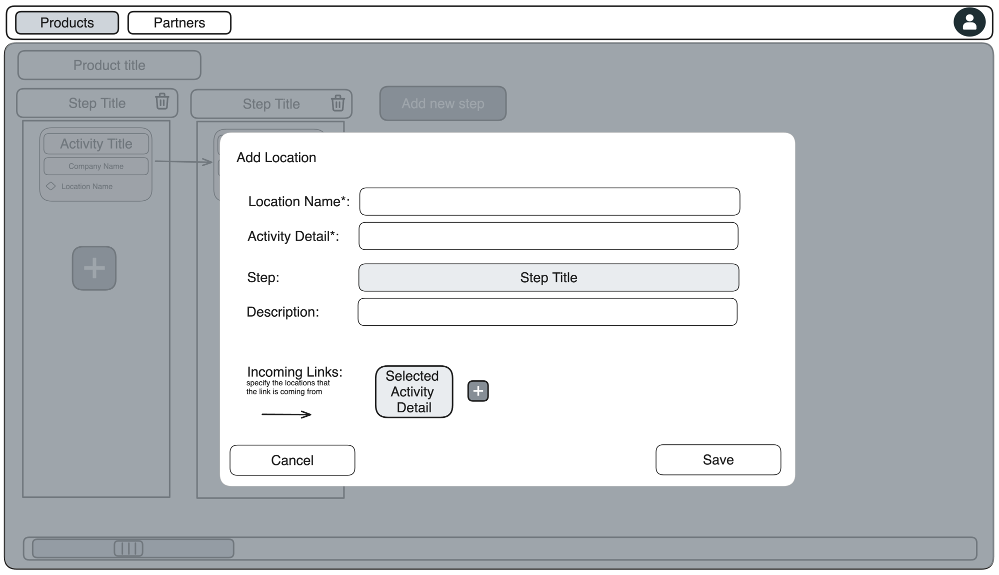

Figure 2 – Adding a new activity to the product journey

product journey and describes their roles within the application.

The key tables in the database include `supplier_locations`, which stores information about various locations involved in the supply chain, such as latitude, longitude, and supplier details. The `supply_chain_step_locations` table links specific locations to the supply chain steps, creating a detailed mapping of where each activity in the supply chain takes place. The `supply_chain_steps` table contains the steps involved in the supply chain for a product version, representing specific activities or stages in the product's journey. The `supply_chain_links` table defines the connections between different supply chain steps, which is crucial for visualizing the flow and sequence of the supply chain. The `products` table stores information about the products managed within the system, each of which can have multiple versions. The `product_versions` table stores different versions of a product, each with distinct supply chain steps and details.

The figure 3 shows a simplified version of the of those tables and how they are connected via the FKs.

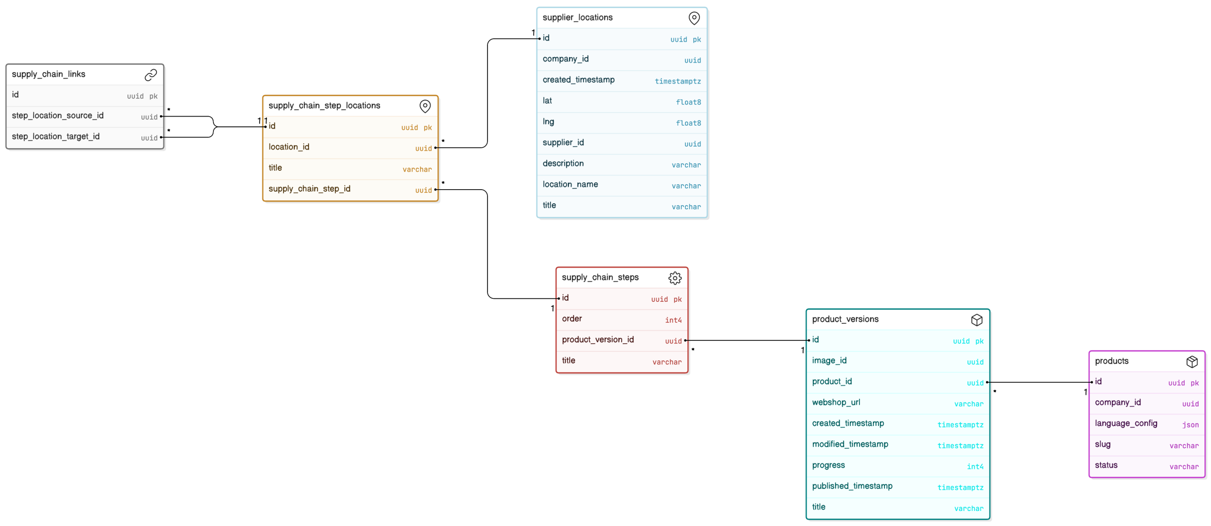

Figure 3 – Current Simplified Database Entity Relationship Diagram

The tables companies, products and supplier are also linked to the tables above but are out of the scope in this project.

The application utilizes these tables to enable users to map out their product supply chains effectively:

- **Mapping Activities**: The `supplier_locations` and `supply_chain_step_locations` tables allow users to specify and visualize the geographical locations and specific activities for each step in the supply chain. Users can add new activities, as shown in figure 2.

- **Defining Supply Chain Steps**: The `supply_chain_steps` table stores detailed information about each step in the supply chain, allowing users to define and organize the sequence of steps involved in the production and distribution of their products. All activities from `supplier_locations` are linked to a specific step.

- **Visualizing Supply Chain Flow**: The `supply_chain_links` table is used to connect different supply chain step locations, enabling users to connect the entire flow of the supply chain. This helps in understanding how various activities of different steps are interlinked.

- **Managing Products and Versions**: The `products` and `product_versions` tables allow users to manage multiple products and their different versions. Each product version can have a unique set of supply chain steps, which are tracked and visualized in the application. Other fields from these tables were removed for confidential reasons.

3.1.2 GraphQL queries and mutations

The application backend provides a set of GraphQL queries and mutations that facilitate the management and visualization of supply chain data. This section outlines the main queries and mutations used to interact with the supply chain information, describing their roles and how they support the functionalities of the application. The GraphQL schema for the product journey can be summarized by it's code representation on 3.1.

```
# Queries
supplyChain(id: ID!, lang: Language = null): SupplyChain
product(id: ID!): Product
supplierLocation(id: ID!, lang: Language = null): SupplierLocation
supplierLocations(lang: Language = null, ...):
    SupplierLocationConnection

# Mutations
createSupplierLocation(input: CreateSupplierLocationInput!):
    CreateSupplierLocationPayload
updateSupplierLocation(id: ID!, input: UpdateSupplierLocationInput!,
    lang: Language = null): UpdateSupplierLocationPayload
deleteSupplierLocations(ids: [ID]!): DeleteSupplierLocationsPayload
updateSupplyChainSteps(id: ID!, input: SupplyChainStepsInput!, lang:
    Language = null): UpdateSupplyChainStepsPayload
updateSupplyChainStepLocation(lang: Language = null, stepLocation:
    SupplyChainStepLocationInput!): UpdateSupplyChainStepLocationPayload
removeSupplyChainStepLocation(input: RemoveSupplyChainStepLocationInput
    !): RemoveSupplyChainStepLocationPayload
```

Listing 3.1 – Current GraphQL schema for Product Journey

The `supplyChain` query retrieves a specific supply chain by its ID, including its steps and step locations. The `product` query fetches details of a product by its ID, including its title, status, and related QR codes. This query is essential for accessing product-specific information within the application. The `supplierLocation` query retrieves detailed information about a specific supplier location by its ID. These locations are linked to supply chain steps locations to be referred to in specific chains and to allow some reusability across products. The `supplierLocations` query provides a list of supplier locations with pagination options. An optional language parameter can be used for localization, ensuring that the data can be presented in the desired language.

The `createSupplierLocation` mutation creates a new supplier location with the specified input data. The `updateSupplierLocation` mutation updates an existing supplier location by its ID with the provided input data. The `deleteSupplierLocations` mutation deletes multiple supplier locations identified by their IDs. This mutation is essential for managing and maintaining the integrity of supplier location data within the system. The `updateSupplyChainSteps` mutation updates the steps of a supply chain by its ID with the provided input data. The `updateSupplyChainStepLocation` mutation updates or creates a supply chain step location with the provided input data. The `removeSupplyChainStepLocation` mutation removes a supply chain step location with the provided input data. This mutation helps maintain the accuracy and relevance of supply chain mappings by allowing the removal of obsolete or incorrect step locations. The mutation provide an optional language parameter can be used for localization, facilitating updates to localized entries.

The application leverages these GraphQL queries and mutations to enable users to map and manage their supply chains effectively. The `supplyChain` query is used to fetch comprehensive details about a supply chain, including its steps and step locations. This data is essential for visualizing the entire supply chain journey within the application. The `product` query allows users to access detailed information about specific products, which is crucial for associating supply chain data with the correct product versions. Queries like `supplierLocation` and `supplierLocations` provide access to detailed information about supplier locations, which can be mapped to supply chain steps. Mutations such as `createSupplierLocation` and `updateSupplierLocation` enable users to add and modify these locations. The `updateSupplyChainSteps` mutation allows users to update the sequence and details of supply chain steps. The `updateSupplyChainStepLocation` and `removeSupplyChainStepLocation` mutations enable precise management of step locations within the supply chain, ensuring accurate and up-to-date mapping.

The key features of the current application include step and location mapping, interactive visualization, geographical context, supplier management, and customizable step names. Users can define and visualize each step and activity in a product's supply chain. This includes specifying locations and connecting different steps to illustrate the flow of the supply chain. The application offers an interactive interface where users can click on different elements to view more details, add new steps, or modify existing ones. Each step and activity can be associated with specific geographical locations, enhancing the clarity of the supply chain's geographical distribution. Users can add and manage suppliers directly within the supply chain map, facilitating the integration of supplier data into the overall supply chain visualization. Users have the flexibility to name the steps and activities as desired, allowing for personalized terminology that best fits their processes. While this flexibility can enhance user experience, it may also reduce the standardization necessary for effective sharing and comparison of supply chain data with other companies.

Despite its capabilities, the current application has several limitations, particularly in managing complex supply chains with reusable components. The current system does not support the concept of reusable 'Component Chains.' Each product must be mapped individually, even if multiple products share common components or steps in their supply chains. This leads to redundancy and inefficiency in managing supply chain data. Without reusable components, the same steps and activities must be recreated for each product, increasing the risk of data inconsistency and errors. As the number of products and their associated supply chains grow, the current system's approach can become cumbersome and difficult to manage. Transitioning existing data to a new format that supports reusable components poses challenges in maintaining historical data integrity.

In the following chapter, the proposed solution will be detailed, outlining the new functional and non-functional requirements necessary to address those limitations.

3.2 PROPOSED SOLUTION

The proposed solution introduces a new section for users to map "Component Chains" within the application. This chapter describes the new features and functionalities, accompanied by some illustration designs, and discusses how these translate into specific backend requirements.

The proposed solution for the application introduces a comprehensive approach to managing and visualizing complex supply chain data by introducing "Component Chains." Component chains allow users to map the journey of individual components separately, which can then be reused across multiple product supply chains. This modular approach enhances scalability and consistency, reducing redundancy by enabling users to define a component's journey once and apply it to various products. The key components of the proposed solution are:

- **Component Chains:** A modular approach to mapping that allows components to be mapped once and reused across multiple supply chains.
- **Supply Chains:** Similar to component chains, but linked to a specific product and typically describing the final product sold by the company.
- **Standardized Chain Steps:** Supply chain steps now follow standardized naming conventions to facilitate the importing functionality of chains.
- **Renamed Supplier Locations:** Supplier locations have been renamed to "Activities" but retain the same functionality with additional capabilities.

To begin with, Figure 4 illustrates the new section for users to map component chains. This example already includes a chain being imported, demonstrating how users can define the journey of individual components within a supply chain. Each component chain can be reused across multiple product supply chains, enhancing scalability and consistency in mapping similar components used in different products.

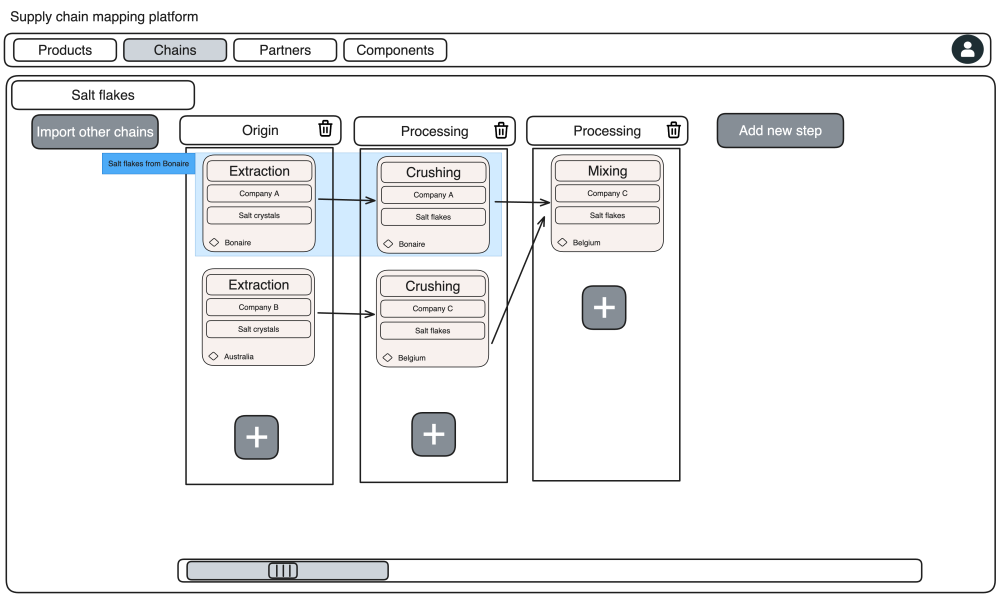

Figure 4 – New section for mapping component chains, including an example of an imported chain

Next, Figure 5 shows a supply chain of a product with multiple imported component chains. These imported chains can now be used in other products that reuse those parts of the supply chain but might include additional activities or materials. This integration allows for a comprehensive view of the product's entire supply process, linked to specific products and ensuring detailed, product-specific mapping.

Additionally, Figure 6 presents the updated dialog for adding activities, which were previously known as supplier locations. The new dialog now includes fields for output components and company details, enhancing the capability to manage various processes and locations within the supply chain. This renaming and additional functionality clarify the role of activities and support more precise and flexible mapping of supply chain steps.

To support the functionality of importing component chains, the steps within the chains are now standardized. Figure 7 illustrates the standardized chain steps, including Origin, Processing, Handling, Manufacturing, Distribution, and Return. This standardization ensures that all supply chains and component chains follow a uniform structure, making it easier to integrate and manage them.

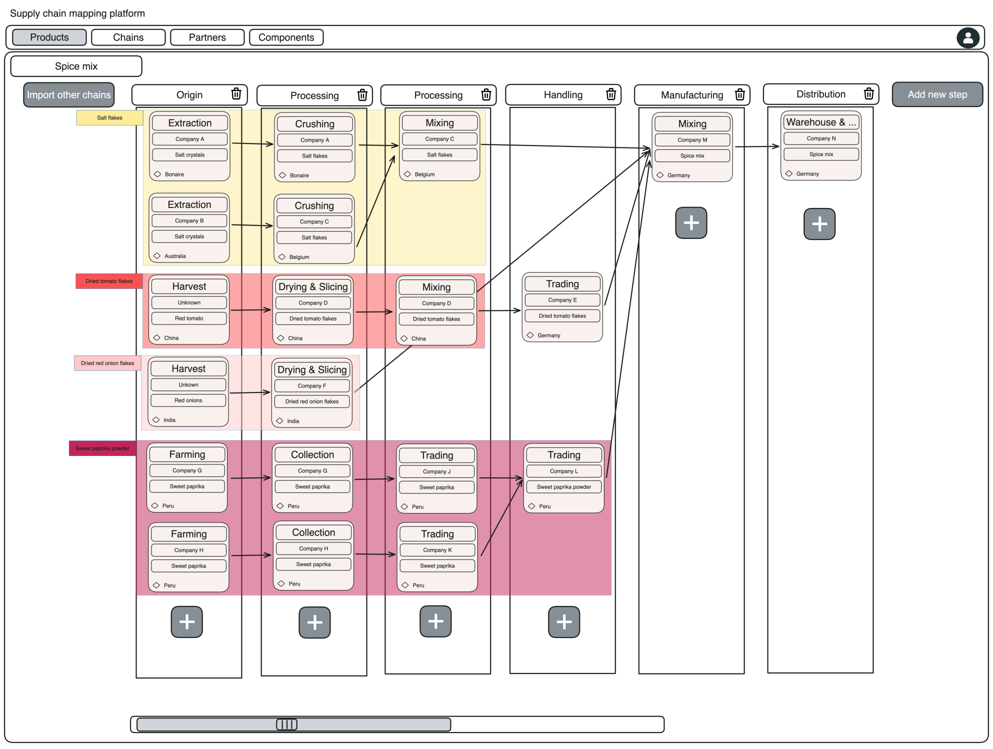

Figure 5 – Supply chain of a product with multiple imported component chains

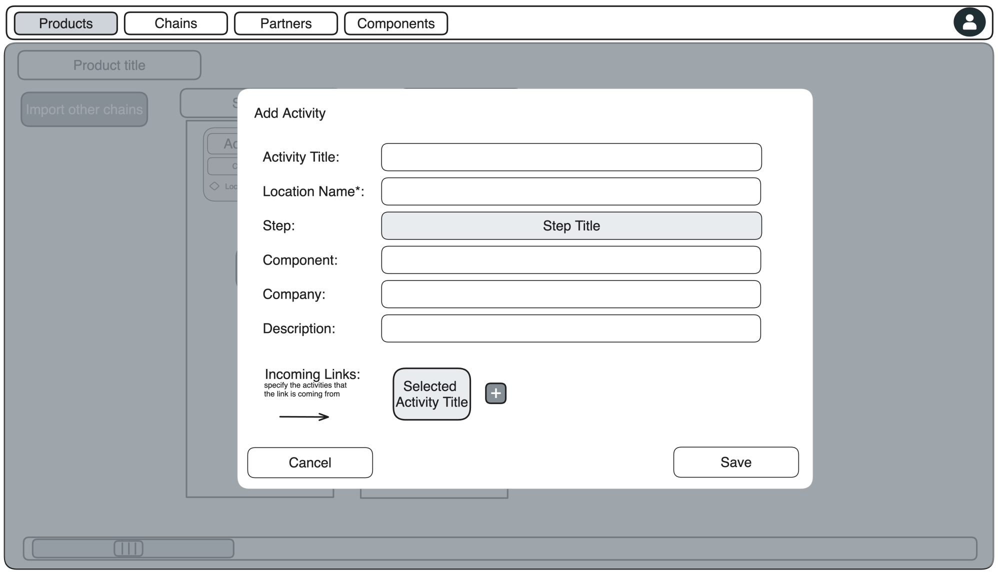

Figure 6 – New dialog for adding activities, including fields for output components and company details

3.2.1 Solution Requirements

The proposed solution should fulfill a set of functional requirements to ensure the successful implementation and operation of the new features within the application. These requirements are essential for enabling the creation, management, and integration of component chains and supply chains, along with standardized steps and

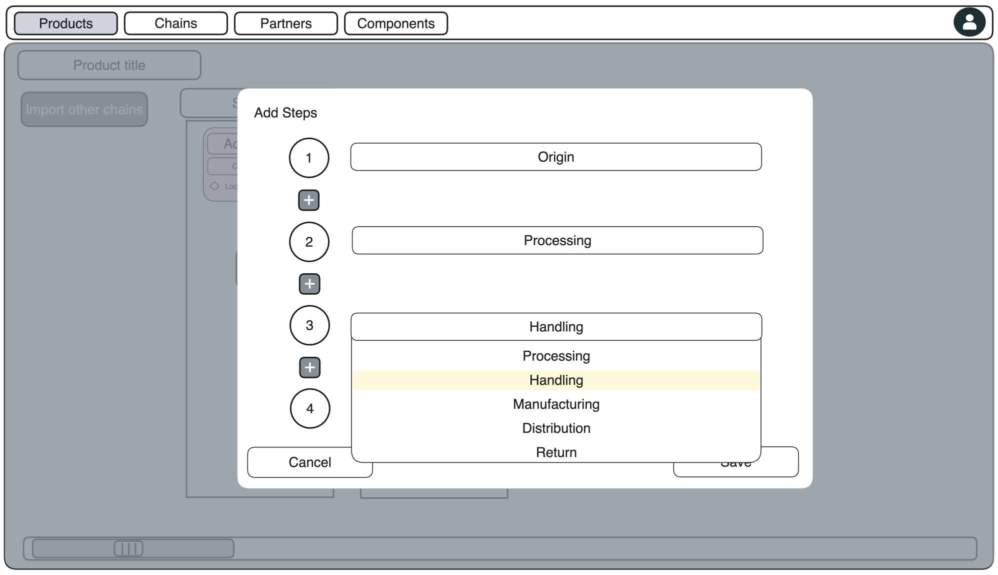

Figure 7 – Add step dialog showing stardarized steps

renamed activities. The summarized functional requirements are presented in Table 1.

In addition to the functional requirements, the proposed solution must meet several non-functional requirements to ensure it performs effectively and provides a robust user experience. These requirements address aspects such as performance, security, usability, and maintainability. The summarized non-functional requirements are presented in Table 2.

Requirement	**Description**
Creation of Component Chains	Allow users to create new component chains, optionally naming and attributing to a partner. Users can create steps and activities within each component chain.
Editing Component Chains	Provide functionality for users to edit details of existing component chains, including modifying steps and activities. Changes should be tracked and logged for audit purposes.
Importing Component Chains	Enable users to import existing component chains into other component chains or supply chains, ensuring imported chains maintain their integrity and relationships. Prevent the import of other supply chains into a supply chain.
Deleting Component Chains	Allow users to delete existing component chains that are not in use. Ensure chains being imported cannot be deleted to maintain data integrity.
Creation of Supply Chains/Products	Allow users to create new supply chains. An empty product is created and linked to the supply chain. Users can define steps and activities within each supply chain using standardized naming conventions.
Standardized Chain Steps	Support predefined, standardized step names and order: Origin, Processing, Handling, Manufacturing, Distribution, and Return. Validate that each step within a chain adheres to standardized naming conventions and order.
Creation and Management of Activities	Allow users to create and manage activities within a chain, including specifying a Partner. Store relevant information such as location, description, and owner details.
GraphQL API Enhancements	Include queries to retrieve component chains and their details. Provide mutations for creating, updating, and deleting component chains. Support functionality for importing component chains into other chains or supply chains.

Table 1 – Functional Requirements for the Application

Requirement	**Description**
System Responsiveness	Provide a fast and responsive user interface, with page load times of less than 10 seconds under normal load conditions. Backend operations should be executed within 5 seconds for 95% of transactions.
Scalability	Handle an increasing number of users and data without performance degradation. Support horizontal and vertical scaling to accommodate growing data volumes and user interactions.
Data Protection	Ensure all data, including supply chain and component chain details, are securely stored and transmitted. Use data encryption for sensitive information both at rest and in transit.
Data Integrity	Ensure accuracy and consistency of data through integrity checks and validation rules. Perform regular backups to prevent data loss in case of failures.
Code Quality	Follow best coding practices and standards to ensure the system is easy to understand, modify, and extend. Conduct code reviews and automated testing to maintain high code quality.
Documentation	Provide comprehensive documentation for both end-users and developers. Regularly update user manuals, API documentation, and system architecture guides.
Modularity	Design the system with modularity in mind, allowing components to be developed, tested, and deployed independently.
Browser Compatibility	Ensure compatibility with all major web browsers, including Chrome, Firefox, Safari, and Edge. Degrade gracefully on older browser versions, ensuring basic functionality is maintained.
Integration with Existing Systems	Integrate with existing modules and third-party systems via well-defined APIs. Ensure compatibility with common data formats and protocols for seamless data exchange.

Table 2 – Non-Functional Requirements for the Application

4 PROJECT IMPLEMENTATION AND TECHNICAL DETAILS

In this chapter, we delve into the practical aspects of implementing the proposed solution for the application. Building upon the foundational design principles and requirements outlined in the previous chapters, this section provides a detailed account of the technical steps taken to bring the solution to life. The focus lies on the comprehensive database design, robust API development, and thorough testing and integration processes necessary to ensure the system's functionality, and scalability.

The chapter begins with the development evironment and moves on to the design of the database schema, which forms the backbone of the application by supporting the management and visualization of complex supply chain data. This includes an overview of the SQL schema and the measures implemented to maintain data integrity and security. Following this, we explore the development of the GraphQL API, detailing the schema design, resolvers, query optimization, and mechanisms for building and propagating changes in supply chains.

Data migration is a critical aspect of transitioning from the existing system to the new implementation. This chapter outlines the strategies and tools used to migrate data seamlessly, ensuring continuity and accuracy. Lastly, we discuss the challenges encountered during the project and the solutions devised to overcome them, providing insights into the problem-solving approaches that were essential to the project's success.

By the end of this chapter, readers will have a comprehensive understanding of the technical implementation of the application's new features, from database design to API development, data migration, and the solutions to the challenges faced during the process.

4.1 DEVELOPMENT ENVIRONMENT

The development environment for this project was set up on a MacBook with an M1 chip, running macOS. The primary programming language used was Python 3.11, managed within the project repository using Poetry for package management. Below is a detailed list of the installed software and tools necessary for building and maintaining the application:

- **Visual Studio Code:** Adopted as the source code editor for its smart syntax highlighting, autocompletion features, and robust extensions for Python, Docker, and Git.
- **Python 3.11:** The main programming language for the project. This version has chosen by it's performance improvement and the language for it's popularity and ease of use.

- **Poetry:** Used as the package manager to handle dependencies and manage the Python environment. Poetry simplifies dependency management and ensures a reproducible environment.
- **Alembic:** Utilized for database migrations, allowing for version control of the database schema and ensuring consistency across different environments.
- **Terraform:** Infrastructure as Code (IaC) tool used to provision and manage infrastructure on Google Cloud Platform (GCP). It enables consistent and repeatable deployments.
- **Docker:** Employed for containerization, allowing the application to run in isolated environments both locally and in production. This ensures consistency between development and deployment environments.
- **PostgreSQL:** The database management system used for storing and managing application data. Locally, PostgreSQL runs within a Docker container, while in production, it is deployed as a managed SQL database on GCP.
- **Make:** Used for automating common setup and development tasks through a Makefile, simplifying environment setup and project management.

4.1.1 Python Project Dependencies

The project dependencies are specified in the pyproject.toml file in the application repository and managed using Poetry. The primary dependencies include:

- **SQLAlchemy:** For defining ORM models and database interaction.
- **FastAPI:** A modern, fast web framework for building API endpoints.
- **Graphene:** For creating a GraphQL API on the routes defined by FastAPI.
- **Pydantic:** For input and output type validation, ensuring data integrity and consistency.

4.1.2 Project Setup

With Python and poetry installed, running the command:

```
poetry install
```

Listing 4.1 – Installing dependencies with poetry

The repository includes a Makefile with commands to streamline setup and development tasks. For example, to set up the database, create a Docker container with PostgreSQL, and run migrations using Alembic:

```
make db
```

Listing 4.2 – Creating the DB with make

Under the hood runs alembic and bootstraps the DB with some initial data:

```
poetry run alembic upgrade head
poetry run python -m boostrap_db
```

Listing 4.3 – make db command

New migrations can be added using alembic. Running the following command compares the current SQLAlchemy ORM models state in the repository and whatever is on the database and create a appropriate migration file with changes to apply:

```
alembic revision --autogenerate "New revision based on changes"
```

Listing 4.4 – Adding new migrations to the DB

The SQLAlchemy models are defined in Python to represent the database schema. These models are used throughout the application to interact with the database in a structured way.

The application can be started using either of the following commands

```
poetry run python app/main.py
```

Listing 4.5 – Running the application with python

```
make api
```

Listing 4.6 – Running the application with make

Both commands will spin up the application using Uvicorn, with the port and database connection details specified by environment variables.

Locally, the PostgreSQL database runs in a Docker container. The connection details are specified in the application's configuration files to ensure seamless connectivity

4.1.3 CI/CD Pipeline

The application is deployed across three environments: development (dev), staging (stg), and production (prd). Each environment is represented by a corresponding Git branch: dev, master, and production. A CI/CD pipeline is triggered whenever changes are committed to these branches. The pipeline runs the appropriate Terraform scripts to provision and manage the infrastructure on GCP.

Figure 8 shows a diagram that illustrates the CI/CD pipeline for the repository:

The CI/CD pipeline for this repository is designed to automate the process of testing, validating, pushing, and deploying code changes. This pipeline ensures that every change is thoroughly tested and validated before being deployed to different environments.

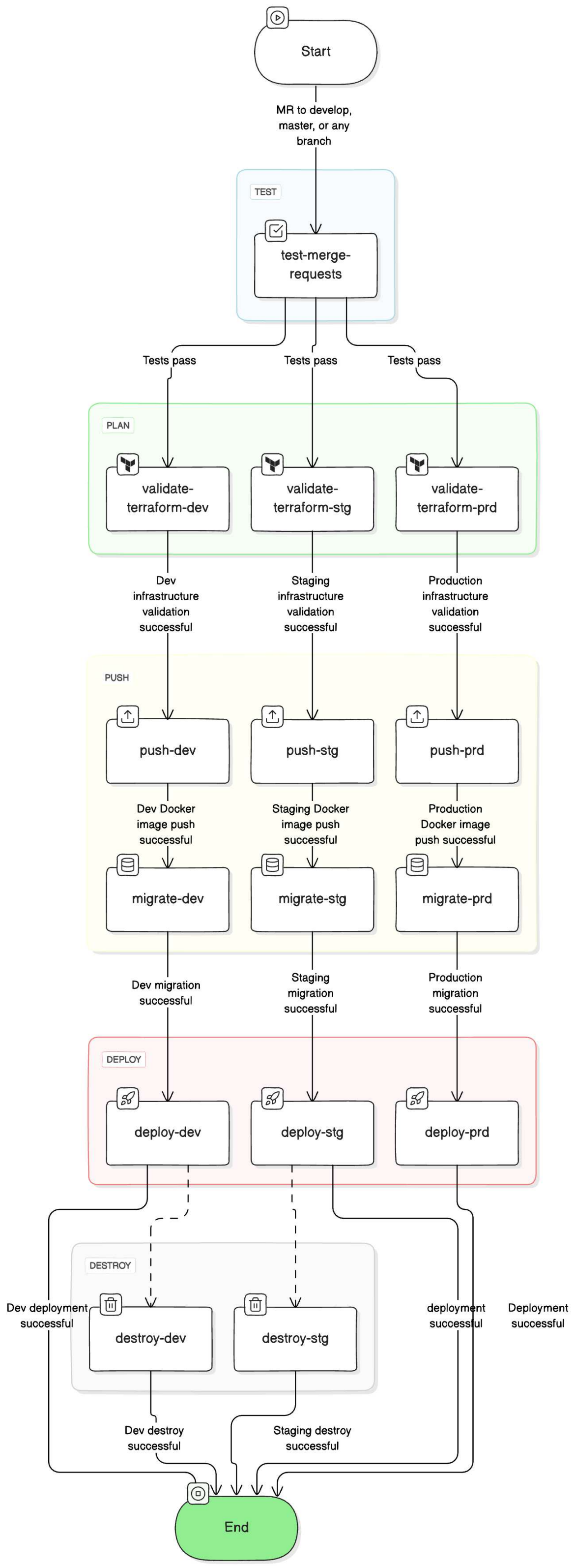
Start
MR to develop, master, or any branch
TEST
test-merge-requests
Tests pass
Tests pass
Tests pass
PLAN
validate-terraform-dev
validate-terraform-stg
validate-terraform-prd
Dev infrastructure validation successful
Staging infrastructure validation successful
Production infrastructure validation successful
PUSH
push-dev
push-stg
push-prd
Dev Docker image push successful
Staging Docker image push successful
Production Docker image push successful
migrate-dev
migrate-stg
migrate-prd
Dev migration successful
Staging migration successful
Production migration successful
DEPLOY
deploy-dev
deploy-stg
deploy-prd
DESTROY
destroy-dev
destroy-stg
Dev deployment successful
deployment successful
Deployment successful
Dev destroy successful
Staging destroy successful
End

Figure 8 – CI/CD Pipeline Diagram

The pipeline consists of several stages and jobs that facilitate this process. The stages include Test, Plan, Push, Migrate, and Deploy, each playing a crucial role in ensuring the application's integrity and functionality across environments.

In the **Test** stage, jobs such as `test-merge-requests` are triggered by any Merge

Request (MR) to the `develop`, `master`, or any branch. These jobs run tests written in Python in the application backend repository to ensure that the new changes do not break existing functionality and maintain code integrity.

The **Plan** stage involves validating the infrastructure as code (IaC) using Terraform. Jobs like `validate-terraform-dev/stg/prd` validate the Terraform configurations for the development, staging, and production environments, respectively, ensuring that the infrastructure changes are correctly defined.

The **Push** stage pushes the validated configurations and Docker images to the respective environments and creates any necessary infrastructure. Jobs such as `push-dev/stg/prd` send the Docker images to the Google Cloud Platform (GCP) repository for their respective environments and use Terraform to create or update necessary infrastructure like Cloud Run instances or SQL databases.

The **Migrate** stage applies necessary database schema changes using Alembic, a Python tool for database migrations. Jobs like `migrate-dev/stg/prd` apply these changes in their respective environments, ensuring the database schema is up-to-date.

The **Deploy** stage involves deploying the new Docker images to Cloud Run, creating new revisions. Jobs such as `deploy-dev/stg/prd` handle the deployment for their respective environments. These deployment jobs depend on the successful completion of both the Push and Migrate stages.

The pipeline also includes optional **Destroy** jobs for tearing down infrastructure in specific environments. These jobs, like `destroy-dev` and `destroy-stg`, are run manually if the developer wishes to perform cleanup.

This CI/CD pipeline ensures a robust process where changes are automatically tested, validated, and deployed, maintaining the stability and reliability of the application across all environments while allowing manual cleanup when needed.

4.2 DATABASE DESIGN

The database design for the application involves a comprehensive schema that supports the management and visualization of complex supply chain data through the introduction of "Component Chains." The schema is designed to enhance scalability, consistency, and data integrity by mapping component journeys and integrating them into product supply chains. The key entities and their relationships are outlined below.

4.2.1 SQL Schema Design

The proposed SQL schema for the solution is presented in Figure 9. This schema includes various tables and their relationships, facilitating the mapping of component chains and supply chains.

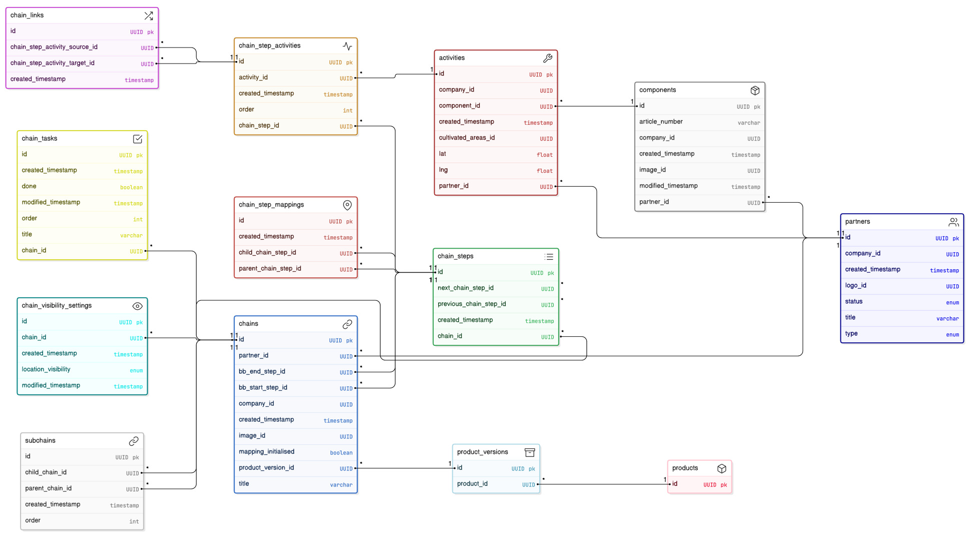

Figure 9 – Simplified Proposed Database Schema for Chains

The schema consists of the following primary tables and their respective relationships:

- **chains**: This new table represents chain entities and serves as the central connection point for the remaining tables to model a real-life chain. When linked to a `product_version` entity, the chain is considered a supply chain; otherwise, it is considered a component chain.
- **subchains**: This table allows the system to understand which chain is imported into which other chain. It includes the following columns:
  - **child_chain_id:** Foreign key linking to the `chains` table, indicating the imported chain.
  - **parent_chain_id:** Foreign key linking to the `chains` table, indicating the parent chain into which the child chain was imported to.
  - **order:** Indicates the position of the imported (child) chain, as multiple can be imported and rearranged.
- **chain_steps**: Contains the steps within a specific chain, similar to the `supply_chain_steps` from Figure 3.
  - **previous_chain_step_id:** A new column indicating the previous step in the chain. An empty value signifies the first step of the chain.
  - **next_chain_step_id:** A new column indicating the next step in the chain. An empty value signifies the last step of the chain.
- **chain_step_activities**: Maps activities to specific chain steps, detailing the order and associated activities. This is similar to `supply_chain_steps_locations` in Figure 3.

  - **order:** This column indicates the relative position of the activity card within the step. It is used by the frontend to determine the positioning of the activity card. This order is relative, as it needs to accommodate changes when dealing with subchains or when other activities are imported into the chain in question.

- **activities**: Stores information about activities, previously referred to as supplier locations, including details such as location and associated components.

- **components**: Contains details about components used in activities, including information on attributes and associations. They represent ingredients of a product.

- **chain_visibility_settings**: Manages visibility settings for different chains, including the visibility status. Although out of the scope of this thesis, it allows users to define how certain parts of the chains are seen by end consumers.

- **chain_tasks**: Contains tasks associated with each chain, including details about the task's status and order. These tasks guide the user in creating a complete and detailed chain representation within the application. Further details about this table won't be provided as it's not the focus of the thesis.

- **chain_step_mappings**: Maps parent and child chain steps to enable hierarchical chain structures, facilitating the organization of complex chains. This table is crucial for allowing chains to be imported into other chains and helps reconstruct the chain's structure with subchains (e.g., positioning steps and their activities). Further details are provided in section 4.3.3.

- **chain_links**: Defines the links between different chain step activities, detailing the source and target activities along with the creation timestamp.

Tables related to products and product versions have not been modified in the proposed schema. However, a product version can be linked to a chain. When a product version is associated with a chain, that chain is considered the supply chain of the product.

4.3 API DEVELOPMENT

The API layer is crucial for the functionality of the application, enabling communication between the frontend user interface and the backend database. The development of the GraphQL API focuses on providing efficient, flexible, and secure access to the underlying data. This section details the design and implementation of the GraphQL schema, resolvers, query optimization techniques, and the specific functionalities for building and propagating changes in chains.

The GraphQL schema serves as the blueprint for the API, defining the types of data that can be queried and mutated, as well as the relationships between different data entities. The proposed queries and mutations for this solution are presented and summarized below.

```
type Query {
    chain(id: UUID!, lang: Language = null): Chain
}
```

Listing 4.7 – Query to retrieve a specific chain by its ID

```
type Query {
    chains(chainFilter: ChainType = null, partnerId: UUID = null,
   mappingInitialised: Boolean = null, lang: Language = null, before:
   String = null, after: String = null, first: Int = null, last: Int =
   null): ChainConnection
}
```

Listing 4.8 – Query to fetch a collection of chains with filters and pagination

```
type Query {
    activity(id: UUID!, lang: Language = null): Activity
}
```

Listing 4.9 – Query to retrieve a specific activity by its ID

```
type Mutation {
    createNewProduct(input: CreateNewProductInput!):
   CreateNewProductPayload
}
```

Listing 4.10 – Mutation to create a new product and a supply chain

```
type Mutation {
    createActivity(input: CreateActivityInput!, lang: Language = null):
   CreateActivityPayload
}
```

Listing 4.11 – Mutation to create a new activity outside of a chain

```
type Mutation {
    createChain(input: CreateChainInput!): CreateChainPayload
}
```

Listing 4.12 – Mutation to create a new chain

```
type Mutation {
    deleteChain(id: UUID!): DeleteChainPayload
}
```

Listing 4.13 – Mutation to delete an existing chain by its ID

```
type Mutation {
    createChainSteps(input: CreateChainStepsInput!):
  CreateChainStepsPayload
}
```

Listing 4.14 – Mutation to add a sequence of steps to a chain

```
type Mutation {
    deleteChainStep(id: UUID!): DeleteChainStepPayload
}
```

Listing 4.15 – Mutation to delete a specific step from a chain

```
type Mutation {
    createChainActivity(input: CreateChainActivityInput!):
  CreateChainActivityPayload
}
```

Listing 4.16 – Mutation to create a new activity within a chain step

```
type Mutation {
    updateChainStep(input: UpdateChainStepInput!):
  UpdateChainStepPayload
}
```

Listing 4.17 – Mutation to update the details of an existing step in a chain

```
type Mutation {
    deleteChainActivity(id: UUID!): DeleteChainActivityPayload
}
```

Listing 4.18 – Mutation to remove an activity from a chain step

```
type Mutation {
    updateChainActivity(id: UUID!, input: UpdateChainActivityInput!):
  UpdateChainActivityPayload
}
```

Listing 4.19 – Mutation to modify the details of an existing activity in a chain

```
type Mutation {
    importChains(id: UUID!, ids: [UUID]!): ImportChainsPayload
}
```

Listing 4.20 – Mutation to import chains into another

```
type Mutation {
    reorderSubChains(id: UUID!, ids: [UUID]!): ReorderSubChainsPayload
}
```

Listing 4.21 – Mutation to reorder the sub-chains within a parent chain

```
type Mutation {
    updateChain(id: UUID!, input: UpdateChainInput!): UpdateChainPayload
}
```

Listing 4.22 – Mutation to update the details or structure of an existing chain

```
type Mutation {
    removeSubChain(id: UUID!): RemoveSubChainPayload
}
```

Listing 4.23 – Mutation to remove sub-chain from a chain

```
type Chain {
  id: UUID!
  companyId: UUID!
  imageId: UUID
  createdTimestamp: DateTime!
  title: String!
  image: ImageVariant
  mappingInitialised: Boolean!
  productVersionId: UUID
  partnerId: UUID
  partner: Partner
  visibilitySettings: ChainVisibilitySettings!
  inUse: Boolean!
  outputComponents: [Component]
  chainTasksDetails: ChainTasksDetails!
  chainSteps: [ChainStep]
  chainStepActivities: [ChainStepActivity]
  subChains: [SubChain]
  boundingBoxPosition: BoundingBoxPosition
  productId: UUID
  involvedPartners: [Partner]
}
```

Listing 4.24 – Type definition for Chain

```
type ChainVisibilitySettings {
  id: UUID!
  chainId: UUID!
  locationVisibility: ChainLocationVisibility!
  createdTimestamp: DateTime!
  modifiedTimestamp: DateTime!
}
```

Listing 4.25 – Type definition for ChainVisibilitySettings

```
enum ChainLocationVisibility {
  APPROXIMATE
  COUNTRY
  EXACT
}
```

Listing 4.26 – Enum for ChainLocationVisibility

```
type ChainTasksDetails {
  tasks: [ChainTask]!
  total: Int
  completed: Int
}
```

Listing 4.27 – Type definition for ChainTasksDetails

```
type ChainTask {
  id: UUID!
  title: String!
  done: Boolean!
  createdTimestamp: DateTime!
  order: Int!
}
```

Listing 4.28 – Type definition for ChainTask

```
type ChainStep {
  id: UUID!
  chainId: UUID!
  title: String!
  order: Int!
  createdTimestamp: DateTime!
  isDeletable: Boolean!
}
```

Listing 4.29 – Type definition for ChainStep

```
type ChainStepActivity {
  id: UUID!
  activityId: UUID!
  chainStepId: UUID!
  createdTimestamp: DateTime!
  activity: Activity
  incomingLinks: [ChainLink]
  pointPosition: PointPosition
}
```

Listing 4.30 – Type definition for ChainStepActivity

```
type Activity {
  id: UUID!
  partnerId: UUID
  componentId: UUID
  createdTimestamp: DateTime!
  description: String!
  title: String!
  name: String
  locationCoordinates: Coordinates
  langs: [String]!
  logs: [ActivityLog]!
  firstLog: ActivityLog
  latestLog: ActivityLog
  mediaList: [MediaType]
  impactClaims(before: String = null, after: String = null, first: Int =
    null, last: Int = null): ImpactClaimConnection
  component: Component
  partner: Partner
  cultivatedAreas: CultivatedAreas
}
```

Listing 4.31 – Type definition for Activity

```
type ChainLink {
  id: UUID!
  chainStepActivitySourceId: UUID!
  chainStepActivityTargetId: UUID!
}
```

Listing 4.32 – Type definition for ChainLink

```
type PointPosition {
  x: Int!
  y: Int!
}
```

Listing 4.33 – Type definition for PointPosition

```
type SubChain {
  id: UUID!
  parentChainId: UUID!
  childChainId: UUID!
  createdTimestamp: DateTime!
  outputComponents: [Component]!
  boundingBoxPosition: BoundingBoxPosition!
  title: String!
  outputStepActivitiesIds: [UUID]!
}
```

Listing 4.34 – Type definition for SubChain

```
type BoundingBoxPosition {
  xMax: Int!
  xMin: Int!
  yMax: Int!
  yMin: Int!
}
```

Listing 4.35 – Type definition for BoundingBoxPosition

```
enum ChainType {
  COMPONENT
  SUPPLY
}
```

Listing 4.36 – Enum for ChainType

GraphQL queries serve to fetch specific data of chains and their related models. The main proposed GraphQL query is `chain` (Listing 4.7), which returns an object with data that allows the frontend to display it to the user.

GraphQL mutations provide various ways to create, update, and delete data related to chains, components, and activities. The mutations that allow the recursive nature of chains to work include `importChains` (Listing 4.20), `createChainSteps` (Listing 4.14), and `createChainActivity` (Listing 4.16).

The schema types define the structure of the data and the relationships between different entities. These types are crucial for ensuring that the data returned by the API is structured correctly and that the relationships between different pieces of data are maintained.

In summary, the design and implementation of the GraphQL API for the application involve creating a schema that defines the data structures and relationships, implementing queries and mutations to manage the data, and ensuring that the API provides efficient, flexible, and secure access to the underlying data.

4.3.1 GraphQL Schema Usage

The `Chain` (Listing 4.24) object is pivotal in the response structure used by the frontend application to visually build the chain. By utilizing the `chainStepActivities` field that contains a list of `ChainStepActivity` (Listing 4.30) objects at their respective positions, defined on `pointPosition` attribute, the frontend is able to construct the chain, including activities from the sub-chains. This structure is constructed per request, ensuring that the latest data is always used. All mutations that change the structure of the chain (e.g., adding a step or importing a chain) return a full `Chain` type. This allows the frontend client to rebuild the chain visually with the updated data. Further details on how this is implemented follows.

4.3.2 Building chains on request

In this section, I discuss the process of constructing the chains response for the frontend, focusing on calculating the correct position of chain step activities. The approach involves recursively traversing imported chains and arranging their activities appropriately. The bounding boxes of subchains are also calculated to enable proper display on the frontend.

The main idea is to look at the subchain table to identify all imported chains. We then recursively traverse these chains to check for further imported chains. Once a root chain is reached, we start adding activities. Importantly, the activities of imported chains are always positioned before those of the source chain requested by the user.

Figure 10 provides an overview of how the chains are constructed per request and reflects the process implemented in the code which is not presented for confidential reasons. Here's a step-by-step explanation of the diagram:

1. **Request Chain:** The process starts when the frontend client (FE Client) sends a HTTP request for a chain.
2. **Initialize ChainBuilder:** The API server initializes the `ChainBuilder` class with the database session and chain ID.
3. **Call build():** The `build()` method of the `ChainBuilder` class is called.
4. **Fetch Root Chain:** If the root chain is `None`, the root chain is fetched using the `chain_id` from the database.
5. **Sort Steps:** The steps of the root chain are sorted using the `previous_step_id` and `next_step_id` attributes of each step.
6. **Initialize Output:** The output object is initialized with the root model's data.
7. **For Each Subchain:** For each subchain, the `build_sub_chain` function is called recursively.
   a) **Get Chain and Related Models:** The subchain and its related models are fetched from the database.
   b) **Extend Step List:** The step list is extended with the subchain steps.
   c) **Recursively Build Subchains:** The `build_sub_chain` function is called recursively for nested subchains. Runs step 7 for the chains.
   d) **Position Activities:** The activities of the subchain are positioned in the output model in the correct step.
8. **Position Root Chain Activities:** The activities of the root chain are positioned in the output model.

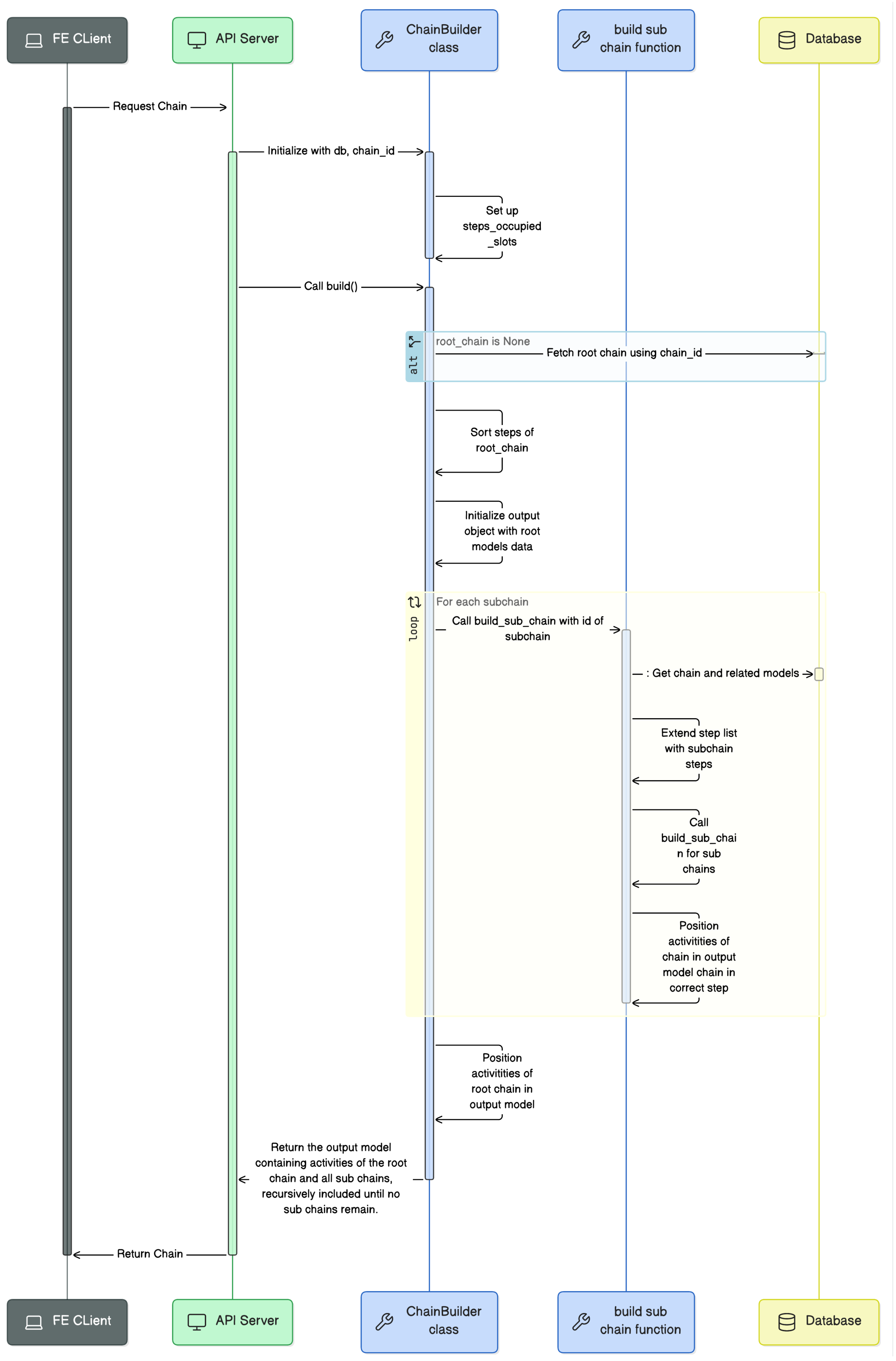
FE CLient
API Server
ChainBuilder class
build sub chain function
Database
Request Chain
Initialize with db, chain_id
Set up steps_occupied_slots
Call build()
alt
root_chain is None
Fetch root chain using chain_id
Sort steps of root_chain
Initialize output object with root models data
loop
For each subchain
Call build_sub_chain with id of subchain
: Get chain and related models
Extend step list with subchain steps
Call build_sub_chain for sub chains
Position activitities of chain in output model chain in correct step
Position activitities of root_chain in output model
Return the output model containing activities of the root chain and all sub chains, recursively included until no sub chains remain.
Return Chain

Figure 10 – Overview of the chain construction process per request.

9. **Return Output Model:** The output model, containing activities of the root chain and all subchains, recursively included until no subchains remain, is returned to the frontend client.

4.3.3 Propagating changes on Chains

Here it's described how changes in one chain are reflected in another chain when that chain is imported. As explained in the previous section, chains are built on request, but we need to ensure certain models are created for the chain to build successfully. Whenever an activity is added to a chain, it is sufficient to add a single corresponding `ChainStepActivity` model to the relevant `ChainStep` of the chain in question. This addition will automatically be reflected in all other chains that import this chain, as the `build_sub_chain` function will incorporate this activity.

To ensure the correct mapping of this activity to the appropriate step of the root chain (since the activity was added to a step of the imported chain and not directly to the root chain), we utilize the `ChainStepMapping` table. This table facilitates the mapping of steps from child chains to their respective parent chains, thereby enabling the correct positioning of activities from subchains within the root chain. Consequently, we can accurately determine the location of the activity within the parent chain.

4.3.3.1 Adding Steps to a Chain

When adding a new step to a chain, we need to ensure a new step is created and linked in all parent chains (chains that import that one), or if the parent chain already contains the necessary steps, just link them. This ensures the hierarchical structure and positions of the steps are maintained across the chains.

The code implementation is not presented for confidential reasons but can be described as follows.

The code manages and updates a sequence of steps within a hierarchical chain structure. This system ensures that steps are correctly added, ordered, and linked, maintaining the integrity and logical flow of the chain and its parent chains. The detailed steps below outline how this function operates:

1. **Fetching Current Steps:** The function begins by retrieving the current steps of the chain.

2. **Creating New Steps:** Based on user input, new steps are created and added to the database. These steps are then inserted into their correct positions within the sorted list of steps.

3. **Updating Step Links:** The linking between steps is updated to reflect the new order, ensuring the logical sequence of steps in the chain is maintained.

4. **Mapping Steps to Parent Chains:** For each parent chain that includes the current chain, the function recursively updates the parent chains to reflect the new steps added to the child chain. This is done using a mapping table that aligns each step

in the child chain with the corresponding step in the parent chain, ensuring the correct positioning of activities in subchains within the root chain.

4.3.3.2 Recursive Addition of Steps to Parent Chains

When adding a new step to a chain, it is crucial to ensure that the new step is also added to all parent chains that import this chain. This process ensures that changes propagate through the hierarchy of chains, maintaining the consistency and integrity of the overall chain structure.

Due to confidential reasons, the actual code implementation is not presented but can be descibred as follow.

The function manages and updates a sequence of steps within a hierarchical chain structure. This system ensures that steps are correctly added, ordered, and linked, maintaining the integrity and logical flow of the chain and its parent chains. The detailed steps below outline how this function operates:

1. **Fetching and Sorting Steps:** The function fetches and sorts the steps of the current chain to maintain a consistent order.

2. **Mapping Current Steps:** A dictionary is created to map the current steps, keeping track of their order and whether they are new.

3. **Adding New Steps:** For each step in the input, the function checks if it already exists in the current chain. If not, it creates a new step and inserts it into the correct position. It also updates the order of subsequent steps to accommodate the new steps.

4. **Updating Chain Step Mappings:** The function updates the chain step mappings using the `ChainStepMapping` table. This ensures that each step in the child chain is correctly mapped to a step in the parent chain, enabling the correct positioning of activities.

5. **Recursive Propagation:** The function recursively calls itself for each parent chain, ensuring that the new steps are propagated up the chain hierarchy.

6. **Updating Bounding Box:** If the last step is new, the bounding box end step ID is updated. The chain steps are linked again to maintain consistency.

7. **Final Commit:** The changes are committed to the database, and the chain is refreshed and updated.

By using the `ChainStepMapping` table to map steps between parent and child chains, the system ensures that any changes, including new activities or steps, are correctly reflected across all chains in the hierarchy. This mechanism maintains the integrity and consistency of the chain structures and their activities.

4.4 MIGRATION OF DATA

The migration of data from the old chain structure to the new one is a critical step in enhancing the platform's capabilities. This section outlines the process and steps involved in migrating the supply chain data from the existing tables to the newly designed ones. The goal is to ensure a seamless transition while maintaining data integrity and enabling the new feature of reusable 'Component Chains'.

Due to confidential reasons, the actual code implementation for the migration is not presented. Instead, the key steps taken during the migration process are outlined below:

1. **Setup and Initialization**: Setting up the environment and importing necessary modules to interact with the database. This includes configuring the path and importing required classes and functions from the application.

2. **Retrieve Products**: Fetching all products from the database to determine which supply chains need to be migrated.

3. **Fetch Latest Product Version**: For each product, retrieving the latest version to work with the most up-to-date supply chain data.

4. **Delete Existing Chains**: Removing any existing chains associated with the product version to avoid duplication and ensure a clean state for the new data.

5. **Create New Supply Chain Entries**: For each product, creating a new supply chain entry in the new structure, initializing it with necessary metadata such as the company ID and chain tasks.

6. **Migrate Steps and Activities**: Transferring each step and activity from the old structure to the new one, ensuring that all relevant details are correctly mapped. This includes mapping the old step activities to new ones and ensuring that all details, such as titles and order, are correctly transferred.

7. **Establish Links**: Creating the necessary links between activities in the new structure to maintain the relationships and dependencies that existed in the old structure. This step ensures that the relationships and dependencies between different parts of the supply chain are preserved.

8. **Finalize and Commit**: Committing the new data to the database and updating the product version to reference the new supply chain structure. The product version

is updated to reference the new supply chain structure, and any temporary or old data is cleaned up to ensure a smooth transition.

This comprehensive migration process ensures that the application's data is accurately and efficiently transitioned to the new structure, enabling enhanced functionality and the use of reusable 'Component Chains'.

4.5 CHALLENGES AND SOLUTIONS

During the implementation of the 'Component Chains' feature, several significant challenges were encountered. One of the primary challenges was dealing with recursive functions, which led to N+1 query problems in database operations. This issue resulted in the generation of numerous database queries, significantly slowing down the API response time.

The recursive nature of supply chain data meant that each level of the chain required additional database queries to retrieve its subcomponents. This recursive querying led to performance bottlenecks, as each query added incremental load to the database, resulting in slow response times for the API.

To mitigate this issue, we focused on optimizing the database read operations. Calculating the positions within the supply chain was relatively quick compared to the time required to retrieve all the necessary data from the database. Therefore, enhancing the efficiency of data retrieval was crucial.

One of the key solutions implemented was adding indexes to the foreign key columns of the subchain models. These indexes significantly improved the performance of read queries, reducing the overall time required to fetch the necessary data. Although this optimization provided notable improvements, further enhancements and considerations are discussed in chapter 6.

4.5.1 Implementation Conclusion

The development of the API layer is a pivotal component of the application, facilitating seamless communication between the frontend user interface and the backend database. This comprehensive schema design enables precise data retrieval and manipulation, crucial for managing the intricate chain structures within the application. The detailed queries and mutations not only support basic CRUD operations but also handle complex hierarchical relationships and ensure data consistency across the entire chain hierarchy.

The process of building and propagating changes within chains is meticulously designed to maintain the integrity and logical flow of activities. By employing recursive functions and updating step mappings, the system ensures that any new step or change in a chain is accurately reflected in all parent chains. This recursive propagation mechanism is essential for maintaining the hierarchical structure and the correct positioning of steps and activities.

In conclusion, the detailed steps and recursive processes described in this section underscore the importance of maintaining a coherent and dynamic chain structure within the application. The API development not only addresses current needs but also lays a strong foundation for future scalability and enhancements, ensuring that the application can adapt to evolving requirements while maintaining data integrity and performance. By following these structured approaches, the application is well-equipped to manage complex chain hierarchies and provide users with a reliable and efficient platform.

5 PROJECT RESULTS AND SYSTEM EVALUATION

To evaluate the performance of the newly developed "Component Chains" feature within the platform, various performance metrics were utilized. These metrics included response time, system throughput, and resource utilization. The key metrics and their evaluations are as follows:

Two types of component chains were used for the performance benchmarks:

- **Level 1 Chain**: A simple component chain that does not import any other chain. It has one Origin and one Processing step. The Origin step contains 3 Harvesting activities that link to the Peeling activity in the Processing step.
- **Level 2 Chain**: A more complex component chain that imports other chains. It has 2 imported chains. In total, 9 activities across 3 steps.

Figures 11 and 12 illustrate the structure of the Level 1 and Level 2 Chains, respectively.

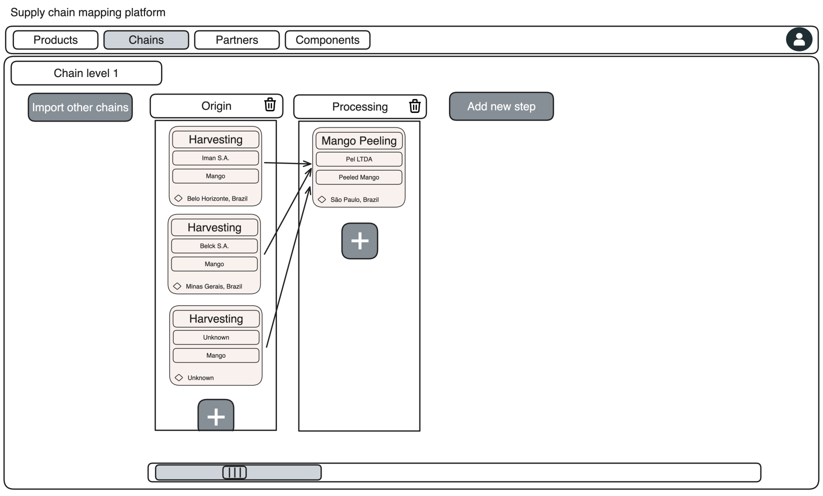

Figure 11 – Level 1 Chain without imported chains.

5.1 RESPONSE TIME PERFORMANCE

To assess the response times of the system a benchmark involving 100 requests to the endpoint to get the chain response, spaced by 10ms delays was performed. The results are as follow:

- **Level 1 Chain Benchmark**: The average response time was approximately 294.74 milliseconds with a standard deviation of 80.43 milliseconds.
- **Level 2 Chain Benchmark**: The average response time was significantly higher at 916.16 milliseconds, with a standard deviation of 95.68 milliseconds.

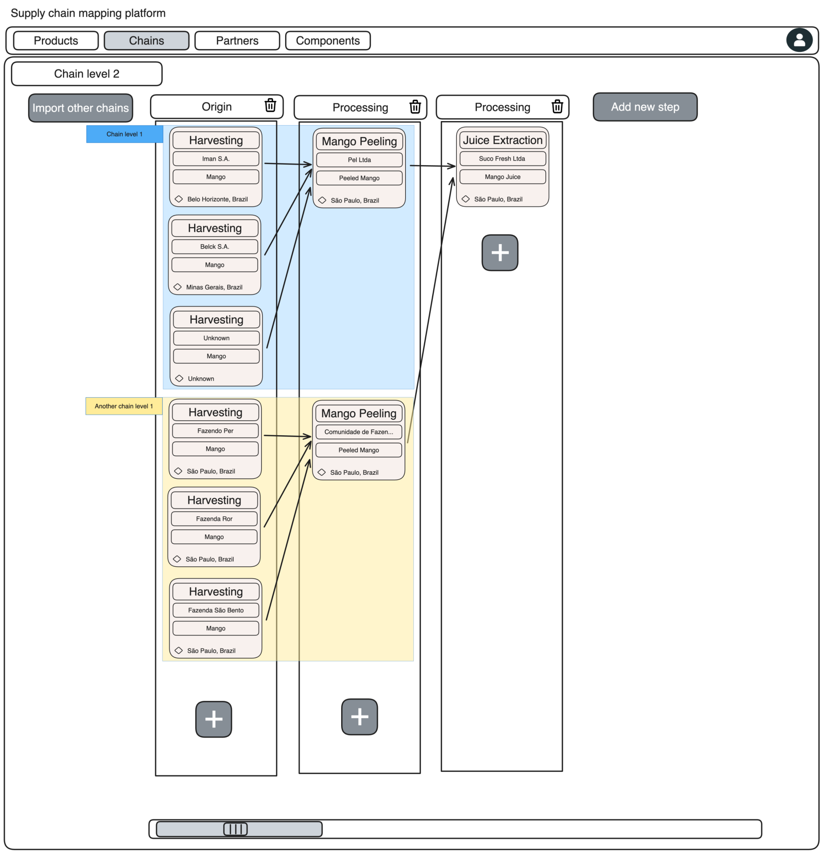

Figure 12 – Level 2 Chain with imported Level 1 Chain.

The comparison of the response times indicates that the Level 2 Chain introduces more complexity and thus requires more processing time. Figures 13 and 14 show the response times for both benchmarks. Additionally, histograms of response times for both benchmarks are shown in Figures 15 and 16.

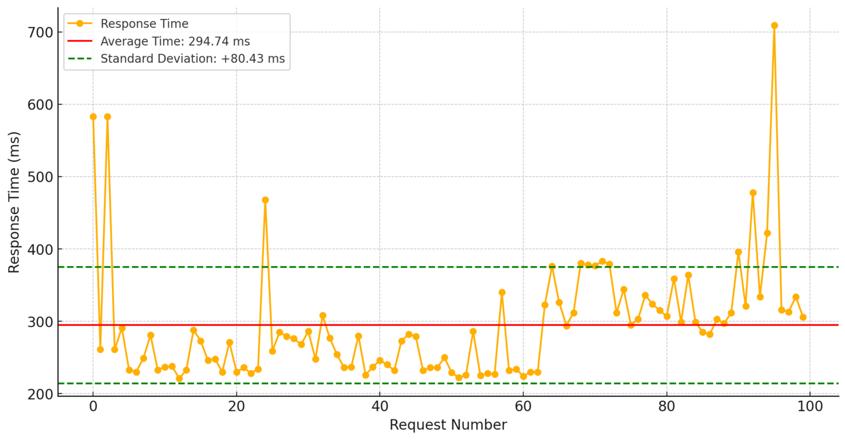

Figure 13 – Response Times for Level 1 Chain

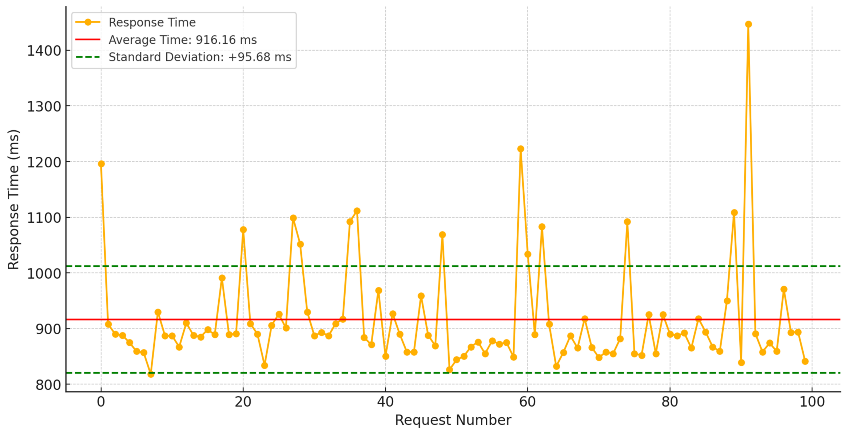

Figure 14 – Response Times for Level 2 Chain

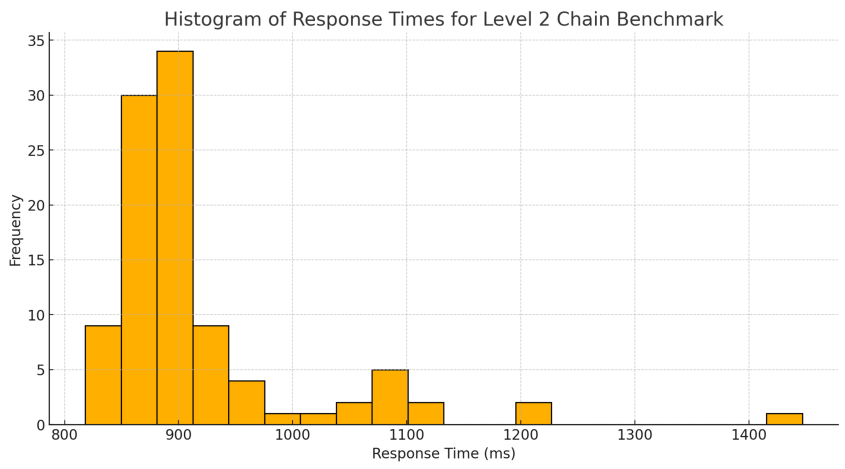

Figure 15 – Histogram of Response Times for Level 1 Chain

5.2 SYSTEM THROUGHPUT PERFORMANCE

System throughput was evaluated by measuring the number of requests processed per second. For both the Level 1 Chain and Level 2 Chain, the same performance benchmark was conducted using Postman. This involved simulating a fixed load of 3 users for 30 seconds. Then, the load was steadily increased to 10 users over the next 30 seconds and held for an additional 1 minute.

- **Level 1 Chain Benchmark**: The results indicate that the system managed to maintain a stable average response time with occasional spikes. The average response time remains relatively low, fluctuating around 400 milliseconds, with occasional spikes reaching up to 1100 milliseconds. The throughput remained relatively high, peaking at around 30 requests per second, demonstrating the system's capacity to handle increasing loads efficiently.
- **Level 2 Chain Benchmark**: The Level 2 Chain showed a higher average response time, reaching up to 3.5 seconds at peak times. The higher response times are expected due to the complexity of importing other chains. The through-

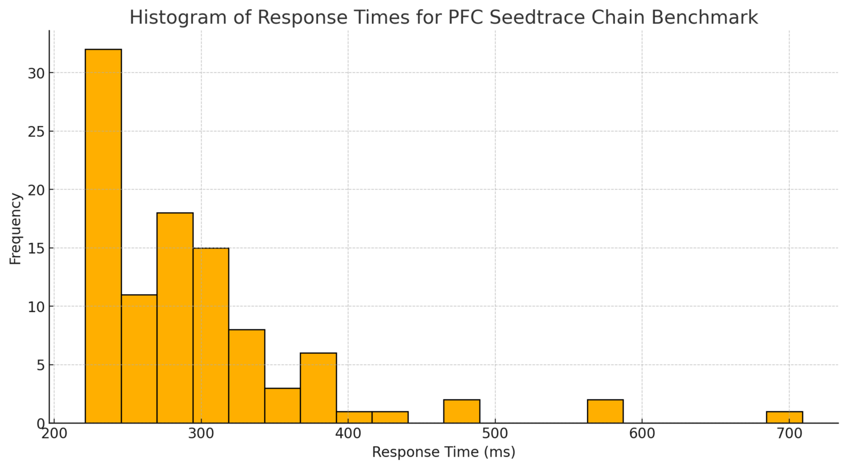

Figure 16 – Histogram of Response Times for Level 2 Chain

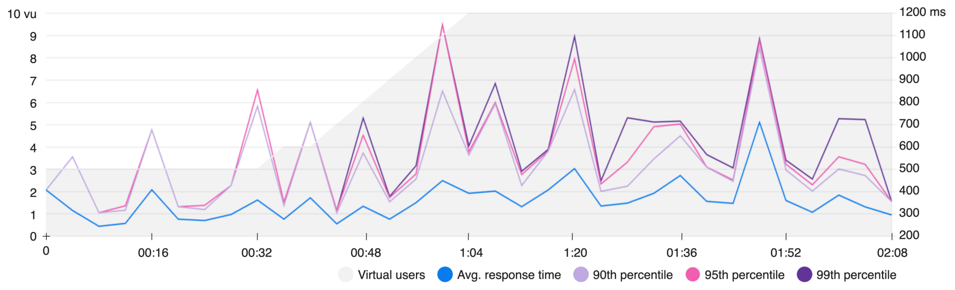

Figure 17 – Response Time for Level 1 Chain Performance Benchmark

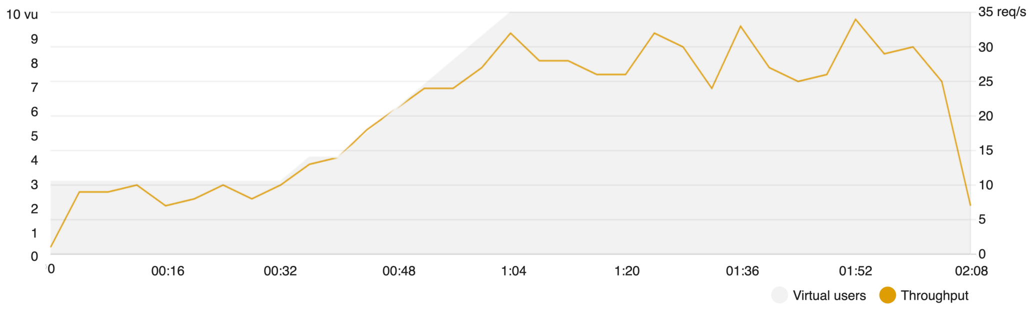

Figure 18 – Throughput for Level 1 Chain Performance Benchmark

put starts at a similar level to the Level 1 Chain but does not increase as consistently, peaking at around 8-9 requests per second, significantly lower than the Level 1 Chain. This lower throughput highlights the impact of the added complexity on the system's ability to process requests efficiently.

5.3 RESOURCE UTILIZATION

CPU and memory utilization were monitored during the benchmarking tests. The results showed that while the system remained within acceptable limits, the Level 2 Chain required more resources, indicating potential areas for optimization in handling more complex data structures. Figures 21 and 22 display the CPU and memory utiliza-

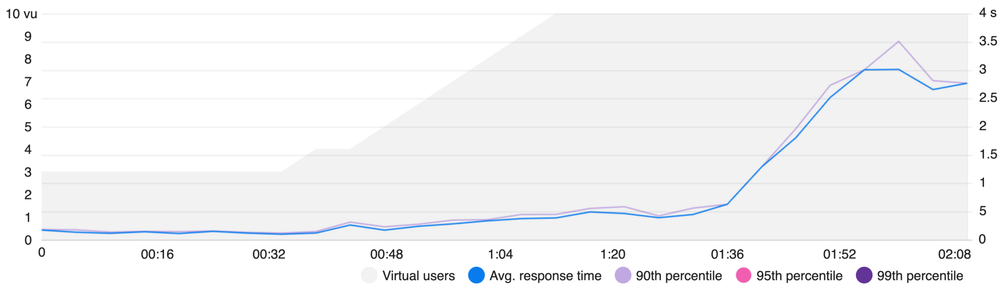

Figure 19 – Response Time for Level 2 Chain Performance Benchmark

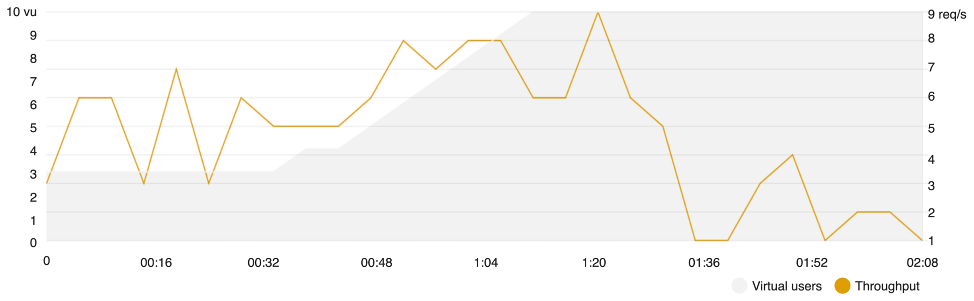

Figure 20 – Throughput for Level 2 Chain Performance Benchmark

tion of the Python Cloud Run container, respectively. Figures 23 and 24 show the CPU and memory utilization of the Cloud SQL database.

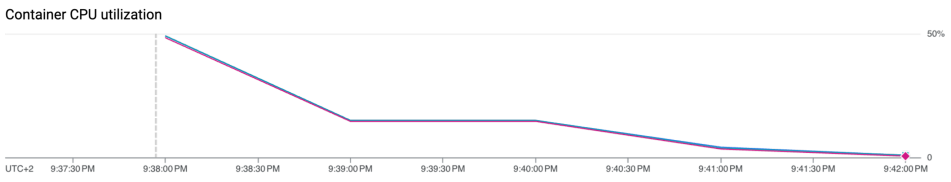

Figure 21 – CPU Utilization of Python Cloud Run Container

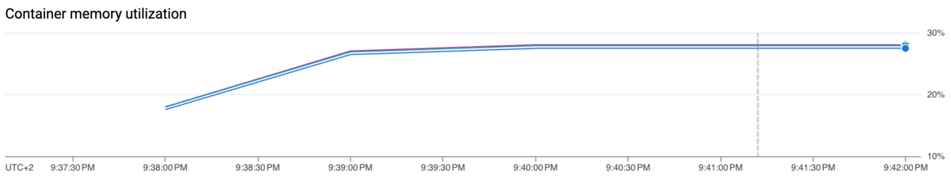

Figure 22 – Memory Utilization of Python Cloud Run Container

As observed, the CPU utilization of the Cloud SQL database peaks at 100%, indicating that the current query designs may need further optimization. Enhancing these queries could reduce the computational load, leading to better overall system efficiency and stability. Future work should focus on optimizing database queries and exploring more efficient data handling techniques to improve performance, especially when dealing with complex data structures in the Level 2 Chain.

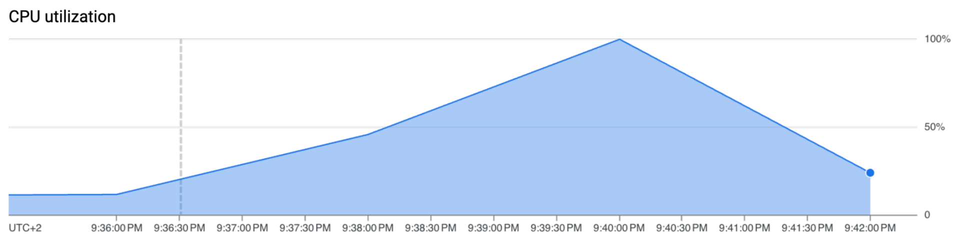

Figure 23 – CPU Utilization of Cloud SQL Database

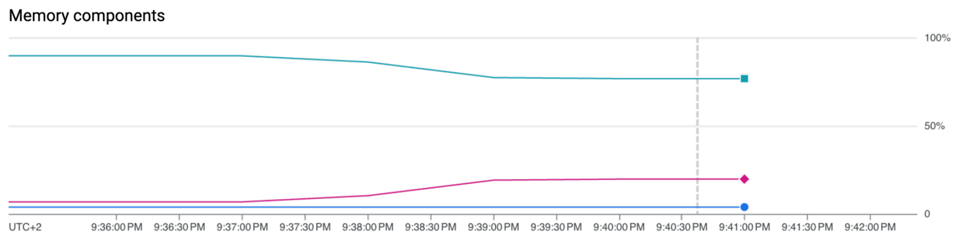

Figure 24 – Memory Utilization of Cloud SQL Database

5.4 COMPARISON WITH INITIAL REQUIREMENTS

The project aimed to introduce reusable "Component Chains" to improve operational efficiency and data consistency. A comparison with the initial requirements shows that:

- **Component Reusability**: Achieved with the development of a system that allows mapping individual components once and reusing them across multiple products.
- **Performance**: While the average response times are within acceptable limits for simpler chains, the more complex Level 2 Chain requires further optimization to meet the initial performance expectations.
- **User Interface**: The enhanced user interface has been well received, meeting the requirement for an intuitive and efficient user experience.

5.5 TECHNICAL ACHIEVEMENTS

The project introduced several innovative technical solutions:

- **GraphQL API Development**: A robust GraphQL API was developed to support the new "Component Chains" feature, allowing efficient querying and manipulation of data.
- **SQL Database Design**: The database schema was restructured to accommodate reusable chains, ensuring data integrity and efficient storage.
- **Automated Data Migration**: A seamless migration process was implemented to transition existing supply chain data to the new format without loss of historical data integrity.

5.6 LIMITATIONS OF THE CURRENT SYSTEM

Despite the successes, the current system has some limitations:

- **Performance Issues with Complex Chains**: The Level 2 Chain benchmarks revealed higher response times, indicating a need for optimization to handle more complex supply chains efficiently.
- **Scalability**: As the number of products and their associated chains grow, the system may require further enhancements to maintain performance and manageability.
- **User Documentation**: More comprehensive user documentation and tutorials are needed to help users fully leverage the new features and indicate they should start mapping their components chains first and then import those in their products supply chains.

5.7 CONCLUSIONS

In conclusion, while the project has significantly improved the platform's capability to manage reusable supply chains, further refinements are necessary to address performance and scalability issues fully.

- **Response Time:**
  - The Level 1 Chain maintains lower and more stable response times, suitable for environments where quick responses are critical.
  - The Level 2 Chain experiences higher and more variable response times, which is expected due to its complexity. Optimization may be needed to handle complex chains more efficiently.
- **Throughput:**
  - The system handles increased loads well for the Level 1 Chain, maintaining a higher throughput.
  - For the Level 2 Chain, the throughput is lower and less stable, indicating the need for performance tuning when dealing with complex chain interactions.
- **Overall Performance:**
  - The platform shows robustness in handling simple chains with minimal degradation in performance under load.

  – For more complex chains, while the system can still manage the load, there is room for improvement in both response time and specially throughput to ensure a consistent user experience.

These results provide valuable insights into the system's performance and highlight areas for further optimization, particularly for more complex chain interactions.

6 CONSIDERATIONS AND PERSPECTIVES

The implementation of the "Component Chains" feature within the platform marks a significant advancement in the realm of supply chain management. The methodology employed was robust, leveraging modern technologies such as GraphQL for API development and SQL for database management. The execution was methodical, ensuring that each stage of the development process was meticulously planned and executed. The outcomes, as evidenced by the performance metrics, indicate a substantial improvement in both efficiency and scalability.

However, the project was not without its challenges. The integration of reusable components posed significant technical difficulties, particularly in ensuring data consistency and managing the migration of existing supply chain data. Moreover, while the system throughput and response times improved, the scalability of the solution may still be limited by the underlying database architecture, which could be a point of concern as the platform scales. Additionally, it was observed that there are often multiple activities at the beginning of the chain, such as "harvest" activities from different farmers or workers. These activities should be clustered together and visualized differently to provide a clearer and more cohesive representation of the supply chain's initial stages.

6.1 FUTURE DIRECTIONS

Several technical improvements can be considered to further enhance the platform:

- **Graph Database Integration:** Incorporating a graph database such as Neo4j or Dgraph could significantly improve the efficiency and scalability of managing complex supply chain data. Graph databases are inherently designed to handle interconnected data, making them ideal for supply chain mapping where relationships between entities are critical.
- **Enhanced Data Analytics:** Developing advanced analytics tools to provide deeper insights into supply chain data could add substantial value. This could include predictive analytics, anomaly detection, and more sophisticated reporting capabilities.
- **Improved User Interface:** While the current interface is functional, there is always room for improvement in terms of user experience. Enhancing the UI to be more intuitive and providing better visualization tools, particularly for clustering initial activities like "harvest" from different sources, could further streamline the process for end-users.
- **Scalability Enhancements:** Addressing the scalability concerns of the current system by exploring distributed database solutions or optimizing the current

database schema can ensure that the platform can handle larger datasets and more complex queries as the user base grows.

6.2 CONCLUDING PERSPECTIVES

The introduction of the "Component Chains" feature has set a new benchmark for supply chain management platforms. It has addressed a critical need for reusability and efficiency, significantly reducing redundancy and enhancing data consistency.

The project’s impact is multifaceted, providing not only operational benefits but also enhancing transparency and trust among stakeholders. By enabling businesses to map their supply chains more effectively, the platform supports compliance with sustainability standards and fosters greater consumer trust.

In conclusion, the development of the "Component Chains" feature represents a significant step forward in the platform supply chain management feature. Future work should focus on further enhancing the platform’s scalability and functionality, ensuring that it continues to meet the evolving needs of businesses and their supply chains. Special attention should be given to the clustering and visualization of initial activities to improve clarity and usability.

REFERENCES

AKBARY, Syrus. **GraphQL framework for Python**. [S.l.]: GitHub, 2015. https://github.com/graphql-python/graphene.

BECK, Kent et al. **Manifesto for Agile Software Development**. [S.l.: s.n.], 2001. Available from: https://agilemanifesto.org/.

BYRON, Eve Porcello; BANKS, Alex. **Learning GraphQL**. [S.l.]: O'Reilly Media, Inc., 2018. ISBN 9781492030713.

CHACON, S.; STRAUB, B. **Pro Git**. [S.l.]: Apress, 2014. (The expert's voice). ISBN 9781484200766. Available from: https://books.google.de/books?id=jVYnCgAAQBAJ.

DUVALL, P.M.; MATYAS, S.; GLOVER, A. **Continuous Integration: Improving Software Quality and Reducing Risk**. [S.l.]: Pearson Education, 2007. (Addison-Wesley Signature Series). ISBN 9780321630148. Available from: https://books.google.de/books?id=PV9qfEdv9L0C.

ELMASRI, R.; NAVATHE, S.B. **Fundamentals of Database Systems, Global Edition**. [S.l.]: Pearson Education, 2016. ISBN 9781292097626. Available from: https://books.google.de/books?id=xdCuDAAAQBAJ.

FMI; FOOD INTEGRITY, The Center for. **Transparency Roadmap for Food Retailers: Strategies to Build Consumer Trust**. [S.l.], 2018. Retrieved from FMI and The Center for Food Integrity.

FMI; LABELINSIGHT. **The Transparency Imperative: Product Labeling from the Consumer Perspective**. [S.l.], 2018. Retrieved from FMI and LabelInsight.

HUMBLE, J.; FARLEY, D. **Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation**. [S.l.]: Pearson Education, 2010. (Addison-Wesley Signature Series (Fowler)). ISBN 9780321670229. Available from: https://books.google.de/books?id=6ADDuzere-YC.

JAIN, Manish. Dgraph: Synchronously Replicated, Transactional and Distributed Graph Database. **arXiv preprint arXiv:1708.09050**, 2021. Available from: https://github.com/dgraph-io/dgraph/blob/master/paper/dgraph.pdf.

LUBANOVIC, B. **FastAPI**. [S.l.]: O'Reilly Media, 2023. ISBN 9781098135461. Available from: https://books.google.de/books?id=XJHhEAAAQBAJ.

MOMJIAN, B. **PostgreSQL: Introduction and Concepts**. [S.l.]: Addison-Wesley, 2001. ISBN 9780201703313. Available from: https://books.google.de/books?id=GE9GAQAAIAAJ.

PERRYMAN, J.; BECHBERGER, D. **Graph Databases in Action: Examples in Gremlin**. [S.l.]: Manning, 2020. ISBN 9781638350101. Available from: https://books.google.de/books?id=vjszEAAAQBAJ.

RAMM, M.; BAYER, M.; RHODES, B. **SQLAlchemy: Database Access Using Python**. [S.l.]: Addison Wesley Professional, 2011. (Developer's Library). ISBN 9780132364676. Available from: https://books.google.de/books?id=GRvHOwAACAAJ.

RESEARCH, Allied. **Food Traceability Market Report**. [S.l.], Mar. 2019. Retrieved from Allied Research.

RICHARDSON, L.; AMUNDSEN, M.; RUBY, S. **RESTful Web APIs: Services for a Changing World**. [S.l.]: O'Reilly Media, 2013. ISBN 9781449359744. Available from: https://books.google.de/books?id=wWnGAAAAQBAJ.

SCHWABER, K.; BEEDLE, M. **Agile Software Development with Scrum**. [S.l.]: Prentice Hall, 2002. (Agile Software Development). ISBN 9780130676344. Available from: https://books.google.de/books?id=BpFYAAAAYAAJ.

TERRAFORM by HashiCorp. [S.l.: s.n.]. https://www.terraform.io. Accessed: 2024-06-04.

ZELLE, J.M. **Python Programming: An Introduction to Computer Science**. [S.l.]: Franklin, Beedle, 2004. (Franklin Beedle Series). ISBN 9781887902991. Available from: https://books.google.de/books?id=aJQILlLxRmAC.