Software Architecture for a Self-Organizing Logistics Planning System.

A continuation study on the SOLiD project.

D. Valdivia

TU Delft

Software Architecture for a Self-Organizing Logistics System

By

D. Valdivia

in partial fulfilment of the requirements for the degree of

Master of Science

in Management of Technology

at the Delft University of Technology,

Faculty of Technology, Policy, and Management

This research was conducted in cooperation with Prime Vision.

Supervisors:	Dr. J.H.R. van Duin,	TU Delft
	Mw.drs. J. Ubacht,	TU Delft
	B. van Dijk, MSc,	Prime Vision
Thesis committee:	Dr.ir. L.A. Tavasszy,	TU Delft
	Dr. J.H.R. van Duin,	TU Delft
	Mw.drs. J. Ubacht,	TU Delft
	B. van Dijk, MSc,	Prime Vision

This thesis is confidential and cannot be made public until December 31, 2022.

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TUDelft

Acknowledgments

To Ron van Duin and Jolien Ubacht, without your support, quick judgment, and consistent feedback, this project would not have been successful.

To Bernd van Dijk and the Prime Vision team, who made these months feel like playtesting among friends. Thank you for your time, company, and expertise, you have made my internship experience wholesome.

To my parents, Ada and Simón, who have been there for me through continents, political crises, global pandemics, and family strife. Your love and support have been the cornerstone of my career. The unfathomable odds of me being in this country and university is a feat only you two can boast of. I will forever be indebted to your effort and sacrifice. You are everything. This achievement is only mine after being yours.

To Mara and Alejandro, because I do not have one set of amazing parents, but two. You do not know how blessed you make me. Thank you for letting me have the warmth of home and even the taste of *cachapas* in a distant land. It is only fair that after sharing the lowest points of a global crisis, you get to share the high ones as well. This work is dedicated to you too.

To my siblings, Johana and Darío, from surfboard battles and Halloween costumes to the first integrals and heat transfer problems, you have always been there for me. I hope you know how blessed I am to have such fine people as my blood. I am a lucky man for being able to meet you consistently in these last years, even through borders and horrid winters. I consider this another statistical miracle. Let this be one shared achievement of many more to come.

To Eduardo, the most unconditional friend you could ever hope for. I thank you for your nomadic ways, they sure have brought me amazing memories, today and always. I hope you find what you seek.

To Mick, for your ever-present support, expertise, good humor, and for reminding me that I should not have messed with this project.

To Sasha and Oriana, my editors in chief. I am lucky to have you in my life as family and friends. Thank you for your work, it takes a lot of love to do what you did for me.

To Jessica and Julia, for your privileged company, confidence, and affection.

To Verónica, for calling me out in a moment where I was less than I could be. Your love and motivation made this project possible.

To Yasmin, Artur, and Logan, for making my two years at TU Delft so special. I cherish the moments we have shared and hope that time can reunite us once again, there are many more things I want to cook for you.

To Freddy, Francesco, and Andrea, as we say in Venezuela, *primero vino sábado que domingo*, you are still the gems of my university days, and neither distance nor time will take away from that.

To Alfredo, Bruno, Jorge, and Miguelangel, as distance will never tarnish our bond as *elmanos*.

D. Valdivia

Delft, October 2020

Abstract

The rise of e-commerce has led to a congested last-mile delivery paradigm. Increasing customer expectations have pushed carriers into a delivery market with diminishing profitability. Furthermore, the current state of last-mile delivery has high societal costs in congestion and environmental impact. To address these challenges, scientists in the logistics field have proposed multimodal transport and collaborative delivery. However, current centralized logistics planning systems are unable to cope with the complexity that these solutions pose. For this reason, Thymo Vlot developed a Self-Organizing Logistics algorithm that leverages decentralization to enable multimodal transport and collaborative delivery. However, the logistics planning system that would utilize this algorithm was left undeveloped, which motivates this thesis project. The first step to develop a software project is to define its architecture. In this thesis project, I analyze Thymo Vlot's algorithm and develop the software architecture of a logistics planning system that would use it. The main design tool consisted of the selection and application of architectural patterns, which are documented solutions to commonplace problems in software development, drawn from the literature and modern distributed systems. The result of this project is an event-driven microservices architecture, which is highly granular, modifiable, and scalable. These characteristics give the project significant commercial value, as the architecture can be applied to different use cases with different algorithms. The project also leaves behind an architecture with small components, which eases the development cycle of the logistics planning system, thus addressing important managerial challenges. Finally, this project makes significant scientific contributions by developing a software solution that addresses managerial, societal, and environmental challenges of last-mile delivery, thus providing a stepping stone for further research that bridges the gap between the logistics and computer science fields.

Contents

1. Introduction ... 1
1.1. Self-Organizing Logistics Systems ... 2
1.2. Self-Organizing Logistics in Distribution Project (SOLiD) ... 2
1.3. Software Architecture and Self Organizing Logistics ... 3
1.4. Problem Statement ... 3
1.5. Research Questions and Research Design ... 4
1.5.1. Phase 1: Problem Definition ... 5
1.5.2. Phase 2: Conceptual Design ... 6
1.5.3. Phase 3: Preliminary design ... 7
1.5.4. Phase 4: Detailed design ... 8
1.5.5. Phase 5: Communication of the design ... 8
1.6. Scope ... 10
1.7. Thesis Outline ... 10
2. Software Architecture and Architecture Patterns ... 12
2.1. Defining Software Architecture ... 12
2.2. The Importance of Software Architecture ... 13
2.3. Quality Attributes and Software Design ... 14
2.4. Architectural and Non-Architectural Aspects of Quality Attributes ... 14
2.5. Functional requirements, Quality Attributes, and Software Architecture ... 15
2.6. Architectural Styles and Patterns ... 15
2.7. The main architectural *styles* ... 16
2.7.1. The Monolithic Architecture ... 16
2.7.2. The Service-Oriented Architecture (SOA) ... 17
2.7.3. The Microservices Architecture ... 17
2.8. Summary ... 19
3. Last-Mile Delivery ... 20
3.1. Overview of the Last Mile of Delivery ... 20
3.2. Key stakeholders in Last-Mile Delivery ... 22
3.2.1. Customers ... 23
3.2.2. Shippers ... 23
3.2.3. Carriers ... 24
3.2.4. Delivery Drivers ... 26
3.2.5. Software Providers for Carriers (Prime Vision) ... 27
3.3. Summary ... 29

4. The SOLiD Algorithm ... 31
4.1. The main functionality of the Algorithm ... 31
4.2. Describing the activities of the SOLiD algorithm in a real delivery scenario ... 32
Step 1: Parcel Creates Transport Request ... 33
Step 2: Central Platform Finds Relevant Vehicles ... 34
Step 3: Vehicle announces participation in the system ... 35
An alternative to Steps 2 and 3: Using a Depot Platform, Depot Digital Twins, and a Vehicle Filter ... 36
Step 4: Vehicle creates bid ... 38
Step 5: Parcel selects the best vehicle ... 40
Step 6: Parcel requests transport when already assigned ... 41
Step 7: Parcel loading process ... 42
Step 8: Parcel pick-up process ... 43
Step 9: Transfer Evaluation at Depots ... 44
Step 10: Parcel Drop-off ... 47
4.3. Summary ... 51
5. Conceptual Design ... 52
5.1. Choosing an architectural *style* ... 53
5.1.1. Considering the monolithic Architecture ... 53
5.1.2. Comparing the Service-Oriented Architecture (SOA) and the Microservices Architecture ... 53
5.2. Defining the Microservices Architecture ... 55
5.2.1. Determining the high-level domain model of the system ... 56
5.2.2. Decomposing the Logistics Planning System in Microservices ... 61
5.2.3. Assigning APIs to the selected services ... 64
5.3. Summary ... 66
6. Application of Architectural Patterns ... 67
6.1. Addressing customer requirements with architectural patterns ... 67
6.1.1. Interprocess Communication, Transactions, and Business Logic ... 67
6.1.2. Data Management and Queries in a distributed system ... 73
6.1.3. External APIs and Security ... 76
6.1.4. Designing for Testability and Modifiability ... 80
6.1.5. Designing for Deployability ... 81
6.1.6. Designing for Scalability ... 83
6.2. Synthesizing the Application of Architectural Patterns ... 84
6.3. Summary ... 86
7. Design Verification ... 88
7.1. The Expert Panels ... 88

7.1.1. Transactions in a Distributed System 89
7.1.2. Specific changes to Services 89
7.1.3. Addition of other functions and services 90
7.1.4. Information Access and Filtering 90
7.2. Non-Architectural Requirements 90
7.3. Defining the final Software Architecture 91
7.4. Summary 94
8. Technology Options 95
8.1. Communication options 95
8.1.1. Event Streaming Platforms 95
8.1.2. Remote Procedure Invocation (Request/Response mechanisms) 96
8.2. Database options 97
8.2.1. Databases that use Structured Query Languages (SQL) 97
8.2.2. Databases that use Non-Structured Query Languages (NoSQL) 97
8.3. Machine Learning Options 98
8.4. Deployment Options 99
8.5. API Gateway and Security Options 100
8.6. Visualization and Monitoring Options 100
8.7. Summary 101
9. Conclusions 102
9.1. Revisiting Research Questions 102
9.2. Relationship with Management of Technology & Supply Chain Management 106
9.3. Scientific Value of the Project 107
9.4. Societal Value of the Project 107
9.5. Practical & Managerial Implications of the Project 107
9.6. Project Limitations & Recommendations for Future Research 108
Bibliography 111
Appendix A. Questionnaire for Developers 116
Appendix B. Interviewee A. Manager Interview at Prime Vision 1 117
Appendix C. Interviewee B. Manager Interview at Prime Vision 2 120
Appendix D. Interviewee C. Commercial Director at Prime Vision 122
Appendix E. Interviewee D. Project Manager Interview from Dutch Carrier 1 124
Appendix F. Interviewee E. Project Manager Interview from Dutch Carrier 2 127
Appendix G. Preliminary List of Operations 130
Appendix H. List of Operations 133

1. Introduction

Parcel delivery is one of the main components of the modern economy: goods need to be transported across vast distances and in increasingly larger volumes. While the total volume of parcels is increasing, a large portion of parcels are directed to single recipients in small volumes in urban areas (Allen et al., 2018; Ducret, 2014). For example, during the coronavirus crisis, e-commerce rose 44% in the Netherlands and 188% in Belgium (Emarsys, 2020). Therefore, the number of deliveries done to customer's doorsteps has grown explosively. This last leg of a parcel's journey is what is considered *last-mile delivery* in the field of logistics (Allen et al., 2018; Gevaers et al., 2011). However, the shift towards denser last-mile networks has come at a cost, since these networks need to cover entire cities, thus increasing congestion, pollution, and noise levels. These consequences are typical of all dense city networks, and the problem is aggravated by their duplication and triplication, as all Carriers have rushed to offer delivery to any point at any time. The result is a very inefficient paradigm of last-mile delivery that impacts both the economy and the environment (Gevaers et al., 2011).

There are several proposals to combat the challenges posed by the oversaturated last-mile paradigm. For example, many carriers like DHL and Hermes, and e-commerce players such as Amazon have begun promoting delivery to smart lockers and alternative Collection and Delivery Points (CDPs), which are spread out throughout the city (Ducret, 2014; Faugere & Montreuil, 2016). Delivering to these alternative locations minimizes the number of parcels sent to customers' doorsteps, easing the planning process, city congestion at peak hours, and opens the door to support crowdsourced delivery (Ducret, 2014; Faugere & Montreuil, 2016). Nevertheless, customers still widely prefer home delivery (Okholm et al., 2013).

Next to delivery to alternative locations, Carriers have also migrated towards more environmentally friendly options for delivery by changing the composition of their fleets, by adding bicycles and electric vehicles for the last mile (Ducret, 2014; Gevaers et al., 2011). These adaptations towards greener options have not always been voluntary, but a result of governmental pressure. This is visible in cases like the recently imposed restrictions in the city of Amsterdam that banned diesel vehicles older than 15 years by 2020 and incoming bans for all buses running on petrol or diesel by 2022 (Amsterdam, n.d.).

Together with changing fleet composition and delivery to alternative locations, another solution to the challenges of last-mile delivery is collaborative delivery. Collaborative delivery hypothesizes the collaboration between competing Carriers to lower the environmental impact of parcel delivery while reducing the congestion and route overlap that occurs when two providers cover the same area with their vehicles (Cruijssen et al., 2007). However, successful implementation of competing Carriers has seldom occurred in practice (Martin et al., 2018). While collaborative delivery has potential, the barriers to its implementation are twofold. In one dimension, Carriers simply do not wish to collaborate, which results in route overlaps between them. The duplication of distribution networks is a natural consequence of competition between carriers, as all of them desire to further their market shares, especially in the dense city areas, which are economically and strategically crucial. Collaborative delivery would imply sharing sensitive commercial data, such as customer segmentation and delivery volumes (Cruijssen et al., 2007; Martin et al., 2018). Research in the field has furthermore assumed total transparency of the data, which is a highly unlikely scenario (See Allen et al. (2017) and Munoz-Villamizar (2015)). Trust is, therefore, a necessity for horizontal collaboration, but it is inherently hard to foster between arduous competitors (Daudi et al., 2016).

Collaborative delivery also has substantial technical difficulties. It requires the orchestration of operations between carriers that utilize centralized planning systems, and thus including the routes from other carriers, estimation of arrival times, and joint handling make collaborative delivery particularly complex (Allen et al., 2018; Cleophas et al., 2019; Martin et al., 2018). This complexity begins in the algorithms that calculate the delivery routes and carries over to the software systems that support these algorithms. For example, if two companies decided to share data, they would have to first make each other's data coherent to the other, in other words, have interoperable data. Then, they would have to find proper digital channels to receive and process said data.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Traditional Centralized systems are severely challenged by the rising complexity of collaborative delivery. This challenge has led researchers to develop a different type of system that is better suited to this cope with this complexity. These are Self-Organized Logistics Systems.

1.1. ***Self-Organizing Logistics Systems***

It could be argued that centralized systems are inherently incompatible with collaborative delivery, as they work under the assumption that there is full information of the fleet. To be more specific, a planner would need to know the potential collaborator's schedule and position of its vehicles ahead of time to be able to fit their parcel into their itinerary. This would be a very unlikely scenario. To adapt to this problem of information incompleteness, and even able to even cope with changing information, a different approach has been theorized: Self-Organizing Logistics Systems (SOLSs). These can be described as logistics systems that do not require significant intervention by humans and are based on local interactions rather than a centralized system (Bartholdi et al., 2010). A Self-Organizing Logistics System decentralizes the decision making process and is thus able to respond to route disturbances and service unavailability because every agent in the system solves the present situation they are in (Pan et al., 2016). These characteristics make Self-Organizing Logistics Systems a fantastic choice for complex, integrated scenarios like the ones hypothesized in collaborative delivery.

These self-organizing systems can be developed with varying degrees of granularity, depending on the focus of the researcher or organization. Most of the available research is focused on the long-haul global transportation, and thus literature on self-organization in last-mile delivery is very scarce. Pan et al. (2016) compiled the main characteristics of these systems, namely openness, intelligence, and decentralized control, which have also been found in work by other authors in the field such as Hülsmann & Windt (2007), Mcfarlane et al. (2013, 2016), and Costa et al. (2016). Motivated by the potential environmental benefits of adopting this technology, and the lack of current applications of these self-organizing systems in logistics, the Self-Organizing Logistics in Distribution (SOLiD) project was launched. The SOLiD project encompasses many universities in the Benelux area, including TU Delft, TU Eindhoven, University Groningen, and the Erasmus University Rotterdam, research partners like TNO, and industry partners like DPD and Prime Vision. One of the latest outputs of the SOLiD project was Thymo Vlot's thesis (2019), which forms the basis of this thesis project, and will be further explored in the following section.

1.2. ***Self-Organizing Logistics in Distribution Project (SOLiD)***

Vlot (2019) developed an algorithm that applies the concepts of self-organization in last-mile delivery. This project was carried out with the help of Prime Vision, a Dutch software company who is specialized in the postal and parcel logistics industry in the Benelux area and is part of the SOLiD project. The algorithm proposed Vlot's work (2019) takes the perspective of a single parcel (as if it was one person) that requests transportation from vehicles based on its preferences, which can be either the fastest transportation possible, the most economical transportation possible, or the greenest transportation possible. Vlot's algorithm will hereafter be referred to as the "**SOLiD algorithm**". The algorithm allows decentralized decision-making by each parcel using a bidding system and it also allows transshipment of the parcel between vans, passenger cars, scooters, and bicycles. The parcel requests transportation, and then the vehicles send offers which are then accepted in terms of the parcel's preferences. The parcel can send multiple transport requests along its route, evaluating whether a transfer is desirable, depending on the current disturbances in its route. If these requests are accepted, the parcel can switch between vehicles, making the system more hypothetically robust than current centralized systems (Vlot, 2019, p. 86). By setting up a simulation of this rule-based algorithm with 1280 parcels and 11 vehicles in the city of Delft and its surroundings, the method allowed for a reduction of 18,1% in distance traveled, operational costs, and CO_2 emissions (Vlot, 2019, p. 77). This 18,1% reduction was equal in distance traveled, operational costs, and CO_2 emissions because Vlot's model used linear relationships between the three variables. Given these results, this method has great potential. When the simulation was repeated assuming collaborative delivery between Carriers, the savings increased to a 24.77% reduction in distance traveled (Vlot, 2019, p. 84).

Given the potential environmental and commercial impact of this technology, Prime Vision, the company that supported Vlot's project, wants to develop a logistics planning system that utilizes the SOLiD algorithm. However,

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

to begin building such a logistics planning system, a blueprint is necessary. In a software development project, said *blueprint* would be the *software architecture* of the system, which motivates the development of this thesis project[*].

1.3. Software Architecture and Self Organizing Logistics

All logistics solutions need a supporting software structure to be implemented. Modern carrier operations are based on a *logistics planning system*, which identifies parcels, calculates vehicle routes, and performs many other activities that are crucial for day to day logistics operations (Gevaers et al., 2011). The modern logistics paradigm is thus linked to digital platforms. The logistics' research field has not, however, developed in unison with computer science. On the one hand, as explored in Section 1.1, the concept of self-organization is novel in the logistics field (Bartholdi et al., 2010; Pan et al., 2016; Vlot, 2019). On the other hand, the backbone of any digital platform, called its *Software Architecture*, is a research field that began in 1992 and has continued to grow ever since (Zhu, 2005). Software architecture expanded in the advent of new technologies like the Internet of Things (IoT), Cloud Computing, Artificial Intelligence, and Machine Learning. Recent applications of these technologies in the industrial world have led to the emergence of completely new concepts, such as the Industry 4.0 paradigm, which merges operational activities in manufacturing to state-of-the-art digital systems to support it (Lasi et al., 2014). In essence, *Software architecture* refers to the abstract representation of a digital system, showing its main characteristics and connection between components (Bass et al., 2003; Zhu, 2005).

Self-Organizing Logistics systems are a new phenomenon, and they have not been tested in real-case scenarios. Given the novelty of the field, it is no surprise that there are no current links between Self-Organizing Logistics and Software architecture; and that products that utilize Self-Organizing Logistics Systems have not been launched in the market. The latest advances in the area have been Vlot's thesis project (2019) and TNO's latest investigation on smart vehicles (2020). This latter whitepaper explores the possibility of making vehicles the main decision-makers in the delivery. Since there are no links between software architecture and Self-Organizing Logistics, the benefits from algorithms like SOLiD cannot be tangibly applied. After the development of the SOLiD algorithm by Vlot (2019), the important question of how to integrate such an algorithm into a digital structure has been left unanswered, which I address in this thesis project. To do this, I will apply the principles of software architecture to create a blueprint for a planning system that utilizes the SOLiD algorithm.

Furthermore, from a managerial point of view, the development of a system's *software architecture* allows for a clear idea of what comprises the system. The software architecture of a system also enhances communication between stakeholders and minimizes the risk of any software project, since there can be a clear expectation of how the system will work (Bass et al., 2003). This gives practical, commercial, and theoretical value to the investigation of the software architecture of a system that harnesses the principles of Self-organization in Logistics.

1.4. Problem Statement

The high societal impact of last-mile delivery makes solutions like the SOLiD algorithm especially important. However, this solution is far from a practical and applicable invention. Such an algorithm would need a substantial digital and physical infrastructure to be implemented. I will partly address the necessary digital infrastructure in this thesis project, which I will develop with the help of Prime Vision. Prime Vision has a unique position in the logistics industry due to its ability to collaborate with several Carriers, many of whom are already customers of the company. Prime Vision also participates in the Self-organizing Logistics in Distribution (SOLiD) project, and therefore Prime Vision wants to develop the software that would support this algorithm and offer it to its customers as an alternative to existing logistics planning systems used in last-mile parcel delivery. This hypothetical product would harness the environmental and societal benefits that the SOLiD algorithm offers and produce operational savings for Prime Vision's customers. This led me to the following problem statement:

Design a software architecture for a Self-Organizing Logistics Planning System.

[*] In this project, the main deliverable is the Software architecture for a Logistics Planning System that would use the Self-Organizing Logistics algorithm. Therefore, I call it the Self-Organizing Logistics Planning System. It is important to clarify that it is the Logistics Planning System that is Self-Organizing, as it uses Thymo Vlot's SOLiD algorithm, not the system's Software Architecture. The software architecture does not present the traits of *self-organization* as defined in the literature by Pan et al. (2016) and Bartholdi et al. (2010).

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Herein, a software architecture refers to an abstract representation of a software system that shows the elements that comprise it and the relationships between them. These elements and their relationships should clearly show how the customer and software requirements are met, and what the design properties of the system will be (Zhu, 2005). The software architecture works at an abstract level that does not delve into the specific details of each element or relationship. Therefore, the resulting blueprint should be comprehensible for both management and development teams.

The relevance of this thesis is twofold: On the one hand, by resolving the previous problem statement, I will design software architecture for a self-organized logistics system, a point that the literature does not tackle directly. To develop and implement these innovations, a software architecture is needed. Therefore, this project addresses important implementation obstacles for Self-Organizing Logistics Systems using tools from computer science. On the other hand, the output of this project should aid Prime Vision in developing a planning system that harnesses the societal and operational benefits of the SOLiD algorithm. Therefore, this project has both scientific, societal, and commercial value.

1.5. ***Research Questions and Research Design***

The previous problem statement leads to the main research question:

What software architecture could support a Logistics Planning System that uses Self-Organization?

To answer the main research question, I will base this thesis project on engineering design. More specifically, I will utilize the Systems Engineering design framework by Dym, Little, and Orwin (2014). I evaluated other frameworks like Johannesson and Perjons's and found many similarities (2014). However, Johannesson and Perjons' framework specifically emphasizes the *demonstration and evaluation of the artifact*. As defined by Johannesson and Perjons, these are portions of the design process that focus on piloting and experimentation. However, given that I will develop the software architecture and will not have software to test, it would be impossible for me to carry out Johannesson and Perjons's *demonstration and evaluation of the artifact*. In contrast, Dym, Little, and Orwin's framework allows for a looser selection of methods to carry out the equivalent activities to prove the design (Dym et al., 2014). Given that the development stage of the Self-Organized Logistics Planning System has not begun, there are important restraints on the available methods for testing the design. For this reason, I chose Dym, Little, and Orwin's framework for this thesis project.

In the Dym et al.'s framework, a design problem is separated into 5 distinct phases, the *problem definition, conceptual design, preliminary design, detailed design,* and *design communication,* as shown in Figure 1-1 (Dym et al., 2014). The *problem definition* phase begins with the problem statement and is the point where the context of the project is established and the objectives for the design problem are formed. This stage also defines constraints and principal functions. After these have been defined, the *conceptual design* phase begins. This phase generates design alternatives and chooses a design concept that suits the requirements mentioned in the previous phase. Furthermore, the *conceptual design* phase delineates several metrics to measure the extent to which the requirements are being met. Then comes the *preliminary design* phase, where the chosen design concept is evaluated on the established metrics. Subsequently, the *detailed design* phase begins, where design specifications are determined. Finally, the final design is documented and conveyed in the *design communication* phase, which includes the report of the design process. This thesis report constitutes this phase of the design.

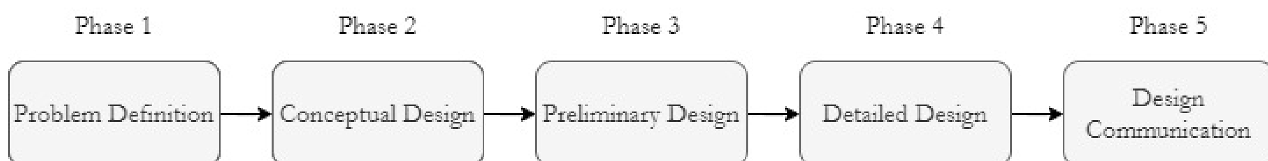

Figure 1-1. Systems Engineering Design methodology by Dym, Little, and Orwin (2014)

This project, however, consists of the development of a software architecture of a product that is yet to be developed. Therefore, the detailed measurement of certain metrics, especially those that involve performance, runtime, and other specific tests, will not be possible, which puts a significant limitation on this research project. To replace these tests, performed in the *preliminary design* phase, I will use expert panels to verify the compliance

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

with customer and functional requirements. Thankfully, one of the main characteristics of Software Architecture is that it allows a developer to infer the characteristics and the performance of the system (Bass et al., 2003). This characteristic strengthens the verification of the project through expert panels. Furthermore, I will utilize known architectural patterns and solutions, which further ease the recognition of the system's qualities.

The level of abstraction in this project's software architecture is so that individual components and their internal workings are out of the scope of the project. Thus, I will not go into each component in depth in this project's *detailed design* phase. In turn, I will discuss potential technologies applicable to the Software Architecture of the Self-Organizing Logistics Planning System. This lack of specificity is also beneficial for the experts that will use this architecture as a blueprint later, as they will have the freedom to assign different solutions that they are familiar with to each of the components in the implementation stage of the project.

In the following section, I will briefly discuss each phase of the research design and the research methods I utilized in each of them. Each phase will lead to a set of sub-questions that help answer the main research question.

1.5.1. Phase 1: Problem Definition

In this section, we will explore the original problem statement by identifying the desired qualities and functions of the planning system. The first step is to establish the context. This project consists of software architecture concepts applied to last-mile delivery. Therefore, we must first understand what software architecture entails, leading to research sub-question 1:

1. *What is software architecture and how can it be designed?*

To answer sub-question 1, I will create an overview of software architecture with the existing scientific literature. This will allow me to explain the terminology utilized in the rest of the project. I cover three important concepts in this section: Software Architecture, Architectural *patterns*, and Architectural *styles*. In short, software architecture allows for a conceptual view of a software system, architectural *patterns* are solutions to commonplace problems, and architectural *styles* demark the most important early design decisions for software systems (Bass et al., 2003; Richardson, 2019; Zhu, 2005). These three concepts will be important to understand the design process of the planning system.

After being acquainted with software architecture, I delve into the last-mile delivery paradigm, which leads to the following sub-questions:

2. *What are the current operations and capabilities of Carriers in last-mile delivery?*
3. *What are the customer requirements for a logistics planning system?*

Research sub-question 2 refers exclusively to the Carrier's capabilities since they would be the users of the Logistics Planning System, and their practical capabilities are important to develop the Software Architecture of a feasible new system. I answer this question by introducing the operations in last-mile delivery and extracting the capabilities of the carriers from their activities. Then, I seek to answer research sub-question 3 by combining a literature study with expert interviews and a stakeholder analysis of the last-mile delivery paradigm. To specify the customer requirements and answer research sub-question 3, I carried out five interviews with industry experts. The industry experts included Prime Vision's product managers and commercial director. Furthermore, I interviewed two senior project managers at major Dutch Carriers. Furthermore, I sent a questionnaire to experts in the Prime Vision development team. This questionnaire helped rank the importance of certain characteristics for the system called *quality attributes*. These characteristics are this project's customer requirements, answering research sub-question 3.

Once the capabilities of parcel delivery operators and their desired qualities in a panning system are known, I delve into the SOLiD algorithm and the functional requirements that it imposes, leading to research sub-question 4:

4. *What are the functional requirements for a self-organizing logistics planning system?*

To answer this sub-question, I developed several Business Process Model Notation (BPMN) diagrams that show the stepwise logic of the algorithm in practice. I confirmed the validity of these diagrams by interviewing the original developer of the algorithm Thymo Vlot. This marks the end of Phase 1 in Dym et al.'s design methodology (2014). At the end of this section, there should be a good understanding of the functional requirements for applying

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Vlot's algorithm (2019), key Carrier capabilities, and the stakeholder's requirements for a self-organizing logistics planning system.

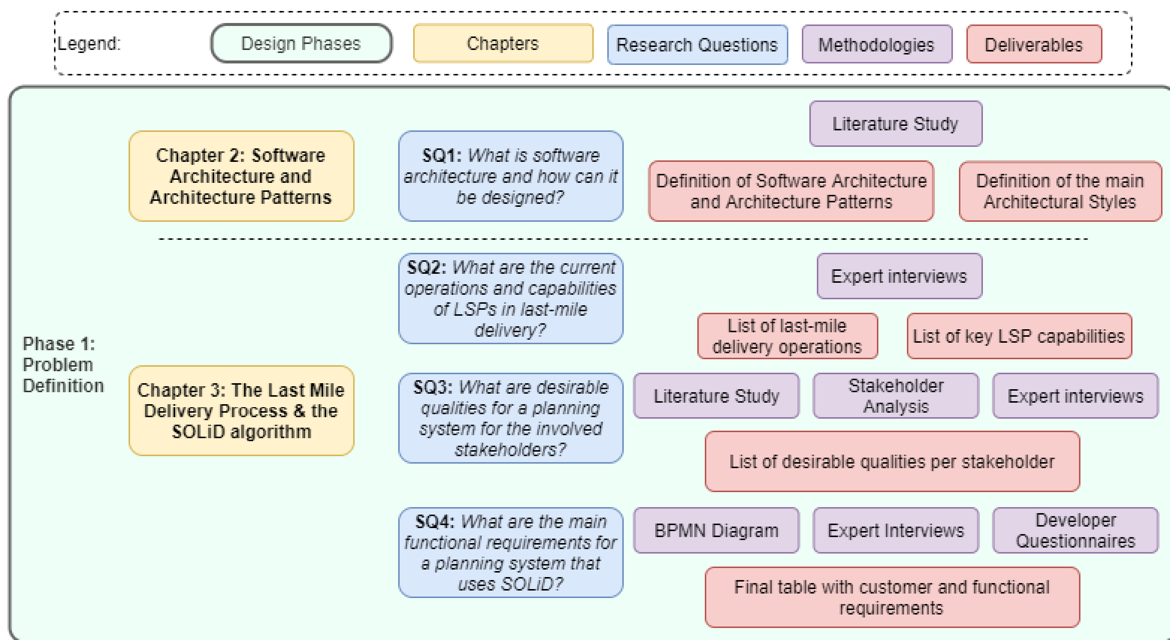

Figure 1-2. Phase 1, Problem definition, adapted from Systems Engineering Design methodology by Dym et al. (2014)

1.5.2. Phase 2: Conceptual Design

The next phase in the engineering design process concerns the *conceptual design*, which ultimately seeks solutions to the customer requirements posed in the previous phase. The output of this section is a draft of the final software architecture. Therefore, in this section I develop a supporting architecture for the SOLiD algorithm, being careful to address the most relevant customer requirements. The first step to define a system's software architecture leads to the following question:

5. What is the most appropriate architectural style for a self-organizing planning system?

To answer this question, I make a brief discussion on the available architectural *styles*: the *monolithic style*, the *Service-Oriented Architecture* (SOA) *style*, and the *Microservices* Architectural *style*. Supported by the customer requirements found in Phase 1, I make an informed decision on the appropriate architectural *style* for the system. After deciding on a style, many implications follow, such as the system's decomposition. Thus, I must answer the following question:

6. How to decompose a self-organizing logistics planning system?

The first step towards decomposing a system is knowing what a system does. Therefore, the BPMN diagrams developed in the previous section will be a good start to imagine the algorithm in practice. This will lead to a rough sketch of the components of the system. Starting from this sketch, I make a more granular separation of system responsibilities, which results in a visual representation of the system's components, called a *Class Diagram*. After developing this graphic representation, I define an explicit list of all the operations in the system. The systems operations are crucial for the final decomposition, which can be done using one of two *decompositional strategies*. I briefly discuss each of them and select one decompositional strategy as the most appropriate for the system. By applying a *decompositional strategy,* I obtain the final components of the planning system.

After defining the components, I must address the problems inherent to a distributed system. To do this, I apply the architectural *patterns* that I described earlier. This led me to the following sub research question:

7. Which architectural patterns would best suit a self-organizing logistics planning system?

Once the domains or pieces of the software are set, several challenges must be addressed. To solve them, I select specific architectural patterns found both in the literature and practice. In Bass et al.'s words, the best way to design

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

software architecture is to employ a process of selecting, tailoring, and combining patterns (Bass et al., 2003). Patterns are, by definition, not "invented", but "discovered" in practice. Working with architectural patterns means taking each problem objectively and exploring alternatives when making a design decision. Thus, in this Section, I develop a series of decision trees for each of the main obstacles to distributed systems. These decision trees show the most relevant *pattern* to solve each of the problems. The customer requirements become the main driver behind the selection of applicable *patterns*.

After selecting the architectural *patterns* and decomposing the system, I develop the conceptual design of the system. This would be the Draft Software Architecture for the Self-Organizing Logistics Planning System. Developing this draft finishes Phase 2 of the design process. Figure 1-3 summarizes the questions, activities, and deliverables for this phase.

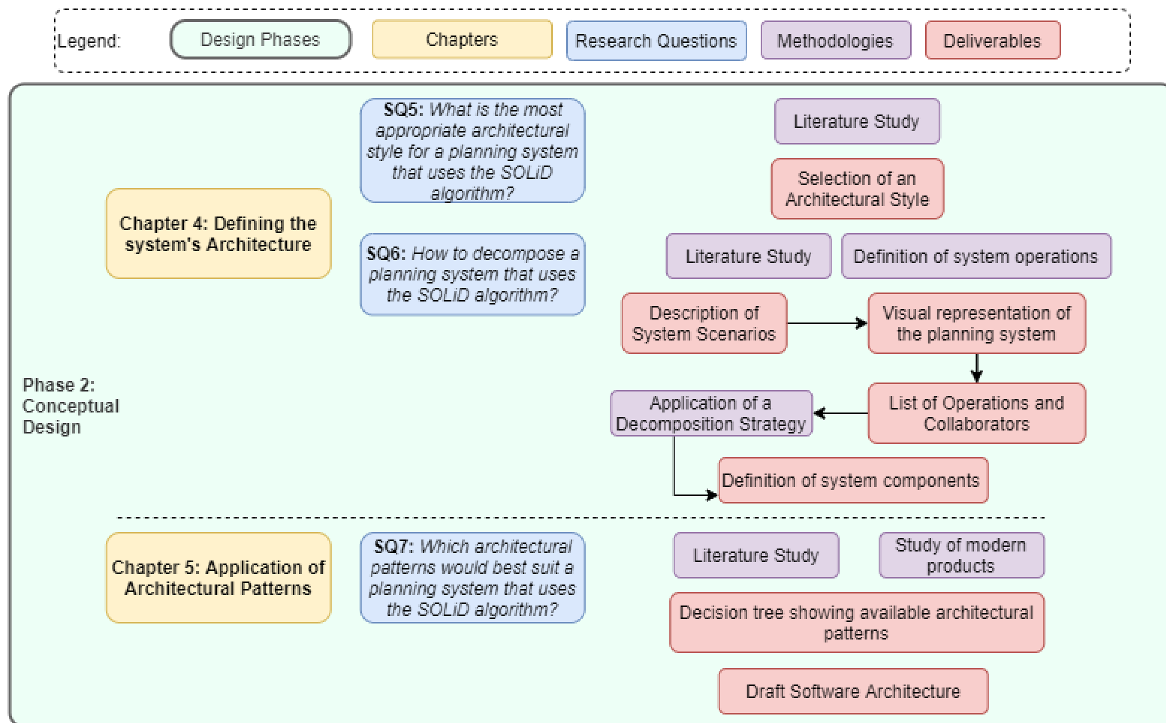

Figure 1-3. Phase 2, Conceptual Design, adapted from Systems Engineering Design methodology by Dym et al. (2014)

1.5.3. Phase 3: Preliminary design

Phase 3 of the Dym et al.'s framework consists of the preliminary design. In this stage, I test the design's compliance with customer requirements. This led me to the following research sub-question:

8. *Would the conceptual design comply with the customer requirements?*

To verify the viability of the conceptual design, I held discussions with software development experts at Prime Vision. The experts comprise software developers from the Innovation department, software developers from the sorting system currently used for first-mile delivery, and software developers from the Internet of Things teams. Therefore, the expert panel comprises experts in computer science, robotics, parcel logistics, and distributed systems. These experts' knowledge strengthens the verification method of this project. The discussions with these experts identified several shortcomings of the draft architecture. I correct these shortcomings in this section as well. This led me to the final software architecture of the Self-Organized Logistics Planning System, answering the main research question of this thesis project. As mentioned before, the extensive testing that usually fills this section is impossible due to the developmental stage of the project. Figure 1-4 summarizes the research design for this Phase.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

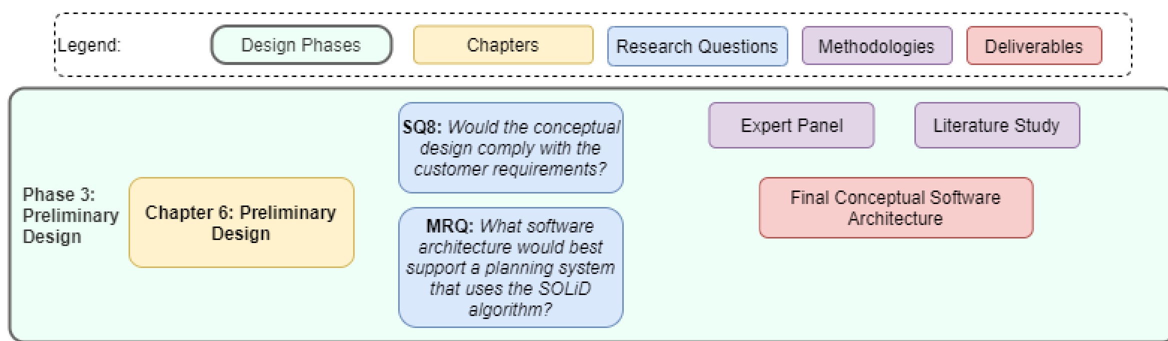

Figure 1-4. Phase 3, Preliminary Design, adapted from Systems Engineering Design methodology by Dym et al. (2014)

1.5.4. Phase 4: Detailed design

Phase 4 of Dym et al.'s methodology (2014) consists of the *detailed design*. This section pertains to the specific design details for the chosen design. When defining research objectives, the incumbent company, Prime Vision, requested adding potential technologies for the planning system. Their request led me to a final sub-research question:

9. *What technologies should be considered to develop the components of this software architecture?*

This section comprises a literature study and the discussions I held with experts about the technologies appropriate for the components of the software architecture defined in Phase 3. In this section, I will create a shortlist of available technologies that could perform the tasks of each component defined in the software architecture. This makes the project more tangible for Prime Vision, further cementing the achievement of the main objective of this research project. Figure 1-5 describes the research question and methodologies I used in this Chapter.

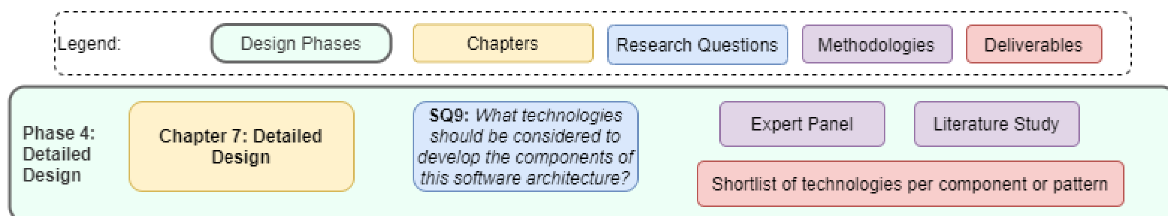

Figure 1-5. Phase 4, Detailed Design, adapted from Systems Engineering Design methodology by Dym et al. (2014)

1.5.5. Phase 5: Communication of the design

The final stage of Dym et al.'s methodology comprises communicating the final design to *the client*. In this case, the client is the incumbent company Prime Vision. I communicate the design through the writing of this thesis report. Furthermore, I include reflections on the project, discussing the contributions and limitations of the project. Finally, I recommend areas for future research. Figure 1-6 shows the deliverables of this phase.

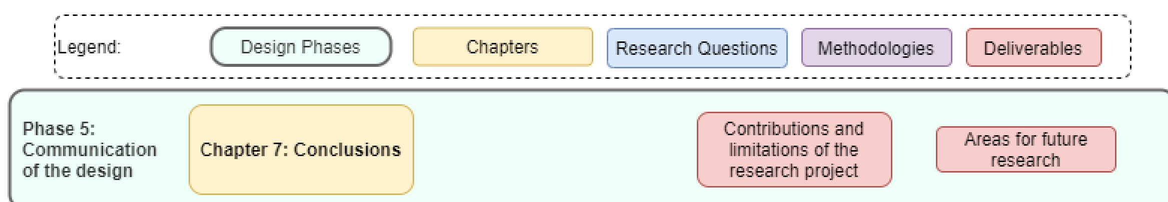

Figure 1-6. Phase 5, Communication of the Design, adapted from Systems Engineering Design methodology by Dym et al. (2014)

By combining all of the research questions and the methodologies utilized to answer them, along with the deliverables of each phase, I created a complete Research Flow Diagram of the project, visible in Figure 1-7.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

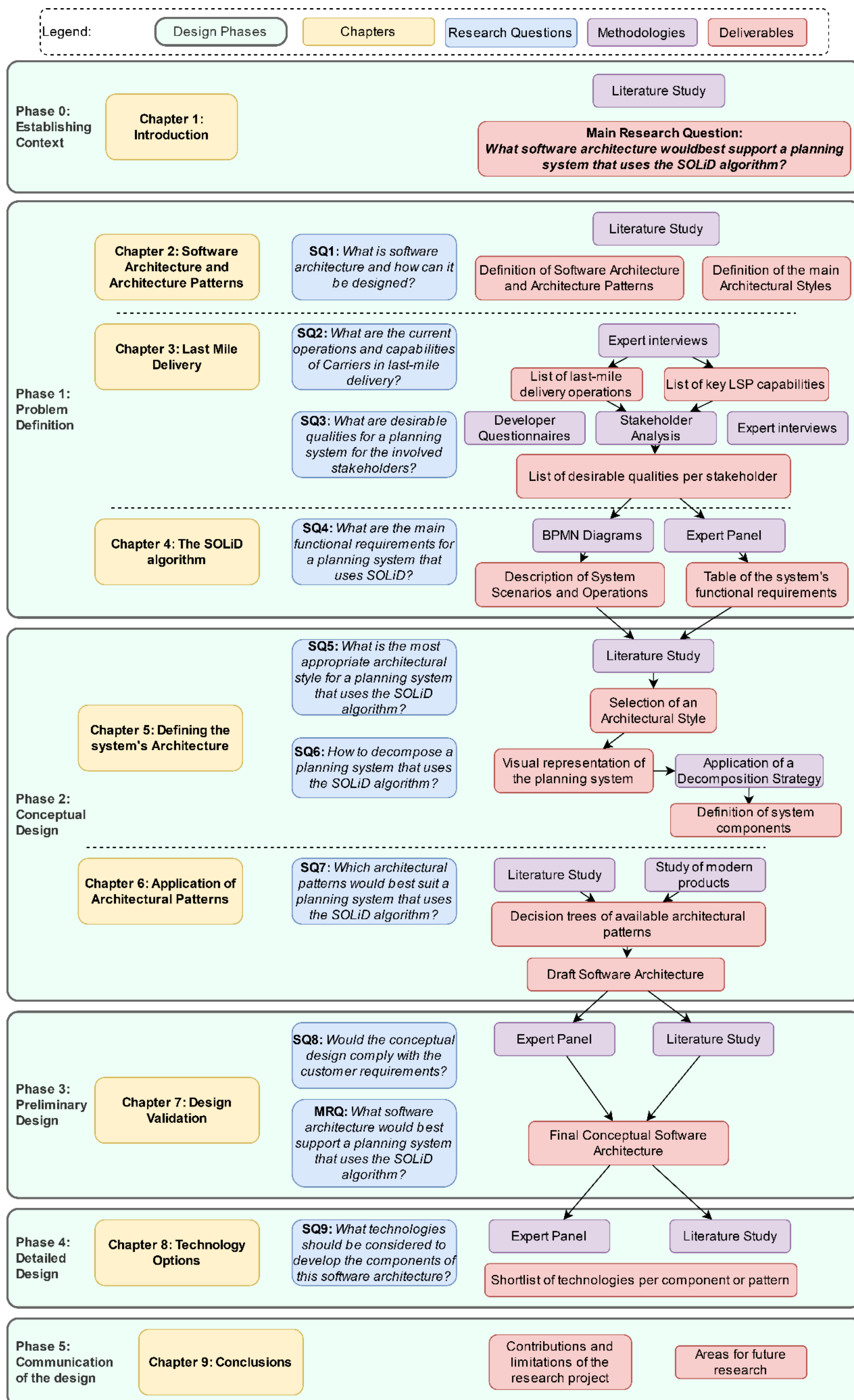

Figure 1-7. Research Flow Diagram

1.6. Scope

The objective of this thesis project is to design the software architecture of a planning system that utilizes the SOLiD algorithm for last-mile delivery. More specifically, the objective is a conceptual view of this system's architecture. The final output from this thesis can be used to develop an alternative product to current logistics planning systems in the last-mile delivery for parcels. Furthermore, this design should be able to be developed by the incumbent company Prime Vision as a commercial product. I design this product to comply with customer requirements I defined from interviewing members of Prime Vision and Industry Experts who work for major Dutch Carriers in the parcel logistics sector.

In this study, a parcel enters the planning system at the moment it is scanned by a carrier, reading its destination, dimensions, and other information, and the parcel leaves the planning system at the moment it is scanned by a delivery driver when delivered to the customer's doorstep or an alternative Collection and Delivery Point (CDP). In this thesis, I design the software architecture of a logistics planning system that handles the information from those two points. Figure 1-8 describes the Delivery chain:

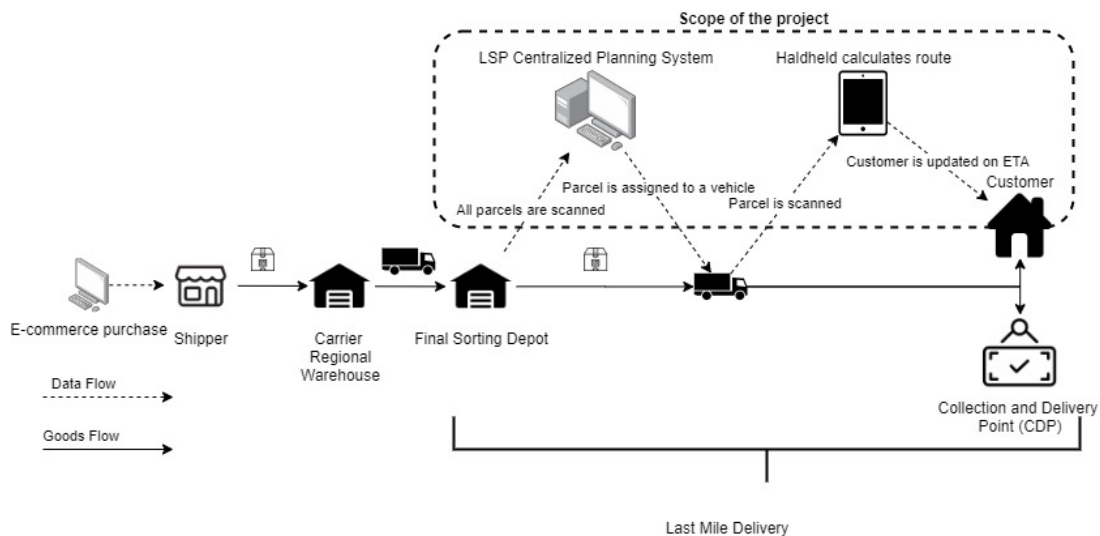

Figure 1-8. Simplified delivery chain (Interviewee D, personal communication, July 22, 2020; Interviewee E, personal communication, July 30, 2020).

The original idea of this project was to find an architecture to implement the SOLiD algorithm by Thymo Vlot. However, this project ends up suggesting certain changes that better suit Prime Vision's and its client's requirements. For this reason, the planning system extends the scenarios of the system to the entire last-mile delivery process as it would occur in practice, thus including the interactions with other agents like depots.

Finally, the project must structure the available information and generate a conceptual design based on customer and company requirements. If the SOLiD algorithm does not cover a necessary point of concern for achieving successful last-mile delivery, this project will point out the deficiencies and offer a solution for Prime Vision at the time of development. Importantly, these solutions will be conceptual, I do not write a second version of the algorithm in this project.

I utilize software architecture patterns found in literature and in practice to design an architecture that complies with customer requirements. This architecture is called the *conceptual view* of the software architecture, and it concerns the major design elements of the system and the relationships among them. The details of each of the elements and certain connections inside them are therefore hidden and will be replaced by abstract entities called *components* and *connectors* (Zhu, 2005). The details inside each component are therefore outside of the scope of this thesis project and will be left to the expertise of the developers that take on the project later. Finally, I include a shortlist of technologies that could be used to develop and deploy these *components*.

1.7. Thesis Outline

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

This thesis report consists of nine chapters. To understand the design process that comprises this thesis project, I define a set of preliminary concepts in Chapter 2. This chapter focuses on Software Architecture, Architectural *patterns*, and Architectural *styles*. Once these are understood, it will be possible to understand the requirements sought in Chapter 3, which sets the context of design by explaining last-mile delivery and its main actors, and their customer requirements for a logistics planning system. Afterward, I describe the logic of the SOLiD algorithm in Chapter 4. The main tool I used for this were BPMN diagrams of the key system scenarios. This led me to the functional requirements of the planning system. With this, I begin the conceptual design phase in Chapter 5. In this Chapter, I discuss the main architectural styles and select one, and I then decompose the system into separate elements called *services*. After I decompose the system in *services*, I discuss the specific architectural *patterns* that solve the challenges of distributed systems in Chapter 6. By selecting these architectural *patterns,* I create a draft architecture that will be tested through discussions with experts in Chapter 7. After revising the architecture, I develop the final conceptual software architecture. Next, Chapter 8 discusses the potential technologies that could be applied to the main components of the system. Finally, Chapter 9 includes the conclusion of the project, where I discuss the contributions made by this thesis project, give recommendations to Prime Vision, and recommend areas for future research.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

2. Software Architecture and Architecture Patterns

"Design is as much a matter of finding problems as it is solving them."

— Bryan Lawson

Software Architecture lies at the heart of this project, and in this I chapter will introduce the core concepts necessary to initiate the design process for a system that utilizes the SOLiD algorithm. In this Chapter, I address research sub-question 1:

1. *What is software architecture and how can it be designed?*

I conduct a literature study throughout this chapter to answer this question. Section 2.1 discusses the beginning of software architecture and finishes with the working definition for this thesis project. Section 2.2 discusses the importance of software architecture, and Section 2.3 introduces quality attributes, a key concept in building customer requirements. I use section 2.4 to discuss the architectural and non-architectural aspects of quality attributes, and I explain the relationship between functions, quality attributes, and software architecture in Section 2.5. Then, I present the concept of architectural *patterns* in Section 2.6, a structured problem-solving tool in software architecture. This section also defines architectural *styles*, the most iconic of which are explored in more depth in Section 0. These are the monolithic, service-oriented, and microservices architectural *styles*. Architectural patterns and styles are the main tools for designing software architecture and defining them thus answers research sub-question 1.

This section is part of Phase 1 of the Systems Engineering Design Methodology by Dym et al. (2014), specified in the Research Flow Diagram of Figure 2-1:

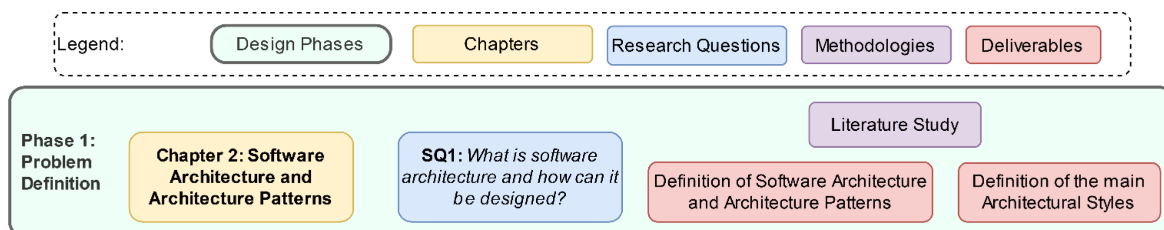

Figure 2-1. Phase 1, Problem definition, adapted from Systems Engineering Design methodology by Dym et al. (2014)

2.1. *Defining Software Architecture*

The concept of architecture is an extension of building design in the world of computer science. Software Architecture comes from replacing the usual elements that comprised a building, such as floors, panels, and beams with the components that comprise a computer system, such as databases, data processors, and messaging protocols. Many researchers have their own definition for it, Perry and Wolf (1992) had a prescriptive definition, that is to determine how a system is to be built. They construed software architecture as a set of architectural *elements* that have a particular *form* and a *rationale* behind them. This prescriptive approach implies a granular view of the elements. Other researchers preferred a more abstract view of software architecture. Shaw and Garlan (1996) defined it as *"involving the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns"*. This is a descriptive definition. Therefore, it refers to software architecture as a description of the high-level structure of elements and their exchanges. Their definition had a simple separation between elements, defining them only as:

- *Components*: a unit of software that performs a task at run-time. These can be objects, programs, processes, databases, etc.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

- *Connectors:* These are mechanisms that mediate communication and coordination between *components*. These normally span many components. Some examples are procedure calls, message protocols, and data streams.

Some researchers believe that a single way of observing and organizing software architecture is insufficient; and just like architectural views in building architecture, Bass, Clements, and Kazman (2003) believe that a system can be observed from many perspectives, and different information can be gathered from its different views. However, the conceptual view of software architecture is persistent across researchers. As synthesized by Hong Zhu (2005), this thesis will use the following definition for software architecture:

"Software architecture is an abstract representation, or model, of a software system in terms of a structure that consists of a collection of elements together with the relationships among them to achieve software design purposes and to manifest a certain set of design properties of the system. The details of the elements and relationships are hidden and replaced with abstract computational entities called components and connectors, respectively. These abstract entities are either represented by several characteristic properties that affect the system or an architectural model of the lower level."

- Hong Zhu (2005, p. 91)

2.2. ***The Importance of Software Architecture***

Software architecture has a pivotal role in easing communication with stakeholders (Bass et al., 2003). As observed in the previous section, certain components are left undefined on purpose. This is beneficial for several reasons. First, this allows non-experts to grasp what the system will accomplish, easing the communication of the design, a crucial aspect of architecture in general.

Secondly, there are usually many solutions to a single problem, and in the field of informatics, new solutions come at a fast rate. Leaving certain modules unspecified leaves room for the developers to take the latest technologies and incorporate them into the system. This gives the system a degree of *transferrable abstraction* (Zhu, 2005), which means that the system can be reenacted in other projects that show similar requirements.

Thirdly, a software architecture also shows early design decisions. Most of the challenges in software development come when adding new functionalities to existing software. This usually occurs because the original system was not intended to perform such tasks. This is a consequence of the original architecture of that software. Therefore, design decisions in software architecture carry great weight in the way the system will be developed, implemented, maintained, and upgraded (Bass et al., 2003; Richardson, 2019; Zhu, 2005).

A good example of this is the complexity of replacing a component in an existing system. To carry out such a task, the original system must have been designed for *modularity*, which is always a challenge (Bass et al., 2003; Richardson, 2019). Thankfully, this is a problem that can be avoided by planning a system's architecture for *modularity* beforehand.

Fourthly, a clear software architecture preserves the integrity of the system's intended purpose. Practitioners in the field have acknowledged that once a project has been continued for several years, the identity of a system can be lost (Richardson, 2019). This is especially prevalent for systems with a *monolithic* architecture, that is, systems whose code is in one hermetic package, including all its components. This creates long and complicated lines of code that probably no individual member in an organization knows in its entirety (Richardson, 2019). Other types of architecture, like the Software-Oriented architecture and the Microservices architecture, separate a software into smaller components, thus being different from the monolithic architecture.

Software architecture also affects how work is divided in an organization (Bass et al., 2003). By designing and dividing a system into modules, an architecture naturally sets a for the work-breakdown structure in an organization. An important corollary to this idea is that if work has already been divided and worked upon, it becomes extremely costly to make significant changes to a system's architecture. This is why an architecture must be extensively analyzed before committing to it (Bass et al., 2003).

Furthermore, as is explored in this thesis project, software architecture allows planning the development of a product. A skeletal system can be drawn for a product in the earliest portion of a product's life cycle, as is with the SOLiD-based product. This allows easier prototyping and testing of Minimum Viable Products (MVPs), which reduce the overall risk of the project since early problems can be addressed.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Finally, software architecture is a vital instrument to achieve a software product's *Quality Attributes*, which will be discussed in the following section.

2.3. Quality Attributes and Software Design

Quality attributes are the essential desired characteristics of a system. By summing all quality attributes, one can infer the quality of a software design (Zhu, 2005). Many researchers have delved into the definition of quality attributes. In terms of engineering design, quality attributes of software design are *design objectives*-characteristics a design should *have* (Dym et al., 2014). Simply put, quality attributes are customer requirements for software systems. The iconic work of Witt, Baker, and Merrit (1993) used the same terminology and defined four *design objectives*, interpreted as quality attributes:

- *Modularity:* A system should consist of a set of self-contained entities that together form the system. These entities must be replaceable, thus aiding the *maintainability* of the system (the ease with which a system can be maintained).
- *Modifiability:* The design should be able to cope with changes in the user requirements. This is particularly important for systems that are expected to be implemented for several years.
- *Portability:* The individual elements of a design should be capable of being reused in different settings or brought from one software/hardware platform to another.
- *Conceptual integrity:* The system should adhere to a single concept. In other words, the system should be consistent, predictable, and symmetrical.

Bass, Kazman, and Clements (2003) recognize other important Quality Attributes relevant to software architecture:

- *Availability:* The system should be ready to execute its tasks when needed, and it accomplishes this by being resilient to system failures. *Availability* thus contains the concept of *reliability* in software design and is related to the time it takes a system to recover from a failure.
- *Performance:* The system must respond in time and meet timing requirements. *Performance* refers to the speed with which a system can perform its tasks. There are several important elements of performance: the number of responses a system can respond to in a minute (also called throughput), the time between an event and the system's response to it (latency), the variation of this latency (also called jitter), and the number of events not processed due to the system being too busy (miss rate).
- *Scalability:* Being one of the most sought-after quality attributes, *scalability* determines a system's ability to increase its working capacity while still performing well. Scalability is then closely linked to *performance*, but also *modifiability* since a *scalable* system is easy to adapt to higher capacity requirements.
- *Testability:* Finding the system's faults should be easy. A *testable* system would be one that "gives out" its faults easily when tested. Empirically, *testability* refers to the probability that a fault in the system will be detected by a test.
- *Deployability:* It should be easy to publish a software's newest version onto the platform that hosts it. A highly *deployable* system would perform updates automatically, and not burden current executions or require significant downtime between updates.
- *Security:* The system should be able to protect its data and information from unauthorized users while still providing its data and information to authorized ones. An action taken against a software with harmful intentions is an *attack*. There are several approaches to design a *secure* system, and they can focus on detecting attacks, resist attacks, react to attacks, and recover from attacks.

Defining these quality attributes gives me a clearer idea of the type of customer requirements in software design projects. I will utilize these concepts when defining customer requirements in Chapter 3.

2.4. Architectural and Non-Architectural Aspects of Quality Attributes

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

The achievement of Quality Attributes in a software product comes throughout the entire lifecycle of the system: design, implementation, and deployment (Bass et al., 2003). It thus follows that no Quality Attribute is entirely dependent on the system's design (architectural aspect) or entirely dependent on the implementation and deployment (non-architectural aspects) (Bass et al., 2003). As software architecture refers to the decomposition of a system in components and how these are connected, not all aspects inside each component are taken into consideration. For example, there is a new system to be developed and high performance is an important requirement. To achieve these high levels of performance, the software architect might consider dividing the software functionality across various component arrangements, streamlining communication between these components, and decide which resources are shared between these components. All these decisions would be architectural, but they do not guarantee that the result will be a system with high performance. This occurs because the system also depends on the quality of the algorithms in the system logic and how the algorithms are coded. These aspects are all non-architectural, and they appear in the development and deployment phases of a software's lifecycle. And in this example, if the coding and quality of the algorithms utilized are subpar, then the system will not achieve high levels of performance, no matter the system's architecture. The opposite is also true, as the best algorithms and clearest codes cannot compensate for a poorly designed system.

For the remainder of this thesis project, this distinction of architectural and non-architectural aspects will be important to determine the accomplishment of several Quality Attributes in the design phase. For example, the type of output an algorithm has does not depend on its software architecture. Therefore, there will be customer requirements that are simply not architectural and will not be able to be tackled directly by the design of the system.

2.5. Functional requirements, Quality Attributes, and Software Architecture

Functions, or things a system *does*, are different from Quality Attributes but are also requirements of a system (Bass et al., 2003). Functions are inherently more linked to the business logic of a software system and are thus dependent on the algorithms of a system. A system's functionality could theoretically be implemented without minding its architecture (Richardson, 2019). Architecture, however, does impact *how* a function is performed. An algorithm can perform a task successfully, but can the algorithm perform said task *securely* and *efficiently*? This is the relationship between Quality Attributes and system functionality. And since quality attributes are deeply linked to software architecture, a software architect must pay close attention to the links between a system's functional requirements and its software architecture.

One can think of software architecture as the chassis of a car. The chassis must be able to support all the other parts of a car, but a good chassis does not guarantee that a motor does its job. One could say that the engineers that designed and manufactured said motor are responsible for its proper functioning. However, the chassis can affect the functioning of the motor. For example: Does the motor fit well into the chassis, or is it difficult to bolt it into the chassis? Does the chassis leave enough space for the motor to perform well, or does it not allow enough flow of air? Are the mechanics familiar with the type of bolts used to connect a motor to that chassis? The functions of a software system, the motor in this example, lean on the software architecture, the chassis, to perform its functions properly. Thus, one needs to understand what a software wants to achieve, and how its architecture can help.

The basis of architectural problem-solving is found in architectural *styles* and *patterns*, which I will discuss in Section 2.6.

2.6. Architectural Styles and Patterns

Architectural *Styles* are commonly found structures and ways of connecting elements in a family of software systems (Bass et al., 2003). An architectural style gives a specific set of allowable structural patterns, design vocabulary, properties, and underlying computational models. Thus, they become building blocks to solve an architectural design problem and include the earliest design decisions of a system (Bass et al., 2003; Richardson, 2019; Zhu, 2005). To design software architecture, you first decide on an architectural *style* and then select architectural *patterns* to solve the problems inherent to that *style*.

An architectural pattern, in contrast, is a more defined package of design solutions that are often found in practice. Architectural patterns, therefore, contain several solutions in a specific arrangement. Each architectural pattern is

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

a package of design decisions, has known properties that allow it to be reused, and specifies a *class* of architectures (Bass et al., 2003).

Architectural patterns associate a *context* with a *problem* and a *solution* (Bass et al., 2003; Richardson, 2019). In this definition, *context* refers to the recurring situation that gives rise to a *problem*. The pattern description summarizes the *problem* and its different forms. Finally, the *solution* documents a successful architectural resolution to the problem. Solutions are documented with their incumbent element types, connectors, topological layout, and important semantic constraints that describe how elements interact in the shown topology and other extra mechanisms that need further explanation. For example, there can be challenges to ask information from a system (called *querying*), especially if the system is distributed. To tackle this problem, there are several architectural *patterns*, like the API Composition *pattern*, shown in Figure 2-2:

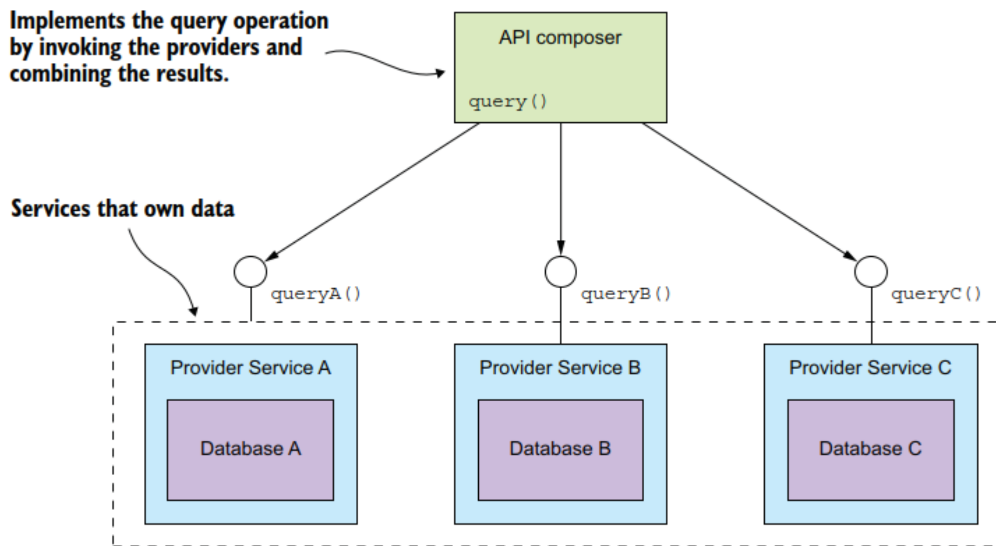

Figure 2-2. The API Composition pattern*, retrieved from Richardson (2019, p. 223).*

The API composition *pattern* consists of having one actor contact several components and aggregating their response. This actor is called the API composer. Therefore, with the API composition pattern, it is possible to solve the problem of queries in a distributed system by contacting one API composer. There are many *patterns* for different problems, and they will be the main tool with which I solve architectural problems.

*2.7. **The main architectural styles***

The main architectural *styles* that I will consider in this thesis project are the *monolithic*, *service-oriented*, and *microservices* architectural *styles*. The *monolithic* style is the simplest form of developing a software and consists of packing software in a single executable file. The Service-Oriented Architectural *style* separates software into clear services that have different responsibilities. Finally, the Microservices *style* separates software into even smaller services with very few responsibilities.

2.7.1. The Monolithic Architecture

The monolithic architecture mentioned in Section 2.2 is an architectural *style*. The idea behind the monolithic style is to create entire applications as a single executable component (Richardson, 2019). This has many benefits. First, the application covers all the business logic of the program. The business logic of the program is what the software wants to *do*. Second, the development, testing, and deployment of a monolithic application are relatively straightforward, as long as the application is not too complex (Bass et al., 2003; Richardson, 2019). When scalability is an important design objective, the monolithic *style* quickly finds a development ceiling. This happens because everything is under the same source code, meaning that the work of many individuals is stretched through an application with a million lines of code, if not more. This hampers an organization's ability to restructure the application, update it and redeploy it, and it is also difficult to assess the impact of certain changes since the code base is so large (Richardson, 2019).

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

2.7.2. The Service-Oriented Architecture (SOA)

There are two alternatives to a *monolithic* architectural *style: service-oriented* architecture (SOA) and *microservices* architecture. Both styles separate the software's business logic into a set of services. In this context, a *service* is a contained piece of software that has a particular function. In the SOA architecture, there are distributed components that provide or consume services (Bass et al., 2003). These services are largely independent, and communication between them is established through an *Enterprise Service Bus* (ESB), which routes messages between providers and consumers. An ESB is a *smart pipe*, which means that it can do additional transformations to the data between services. This characteristic lowers the overall performance of the system and introduces an additional point of failure (Bass et al., 2003).

Another key feature of SOA is the usage of the Simple Object Access Protocol (SOAP) communication protocol. This protocol gives the sender the capacity to route its messages through a series of services before reaching its final destination, allowing secure connections, controlled access, and reliable failure recovery (Bass et al., 2003; Richardson, 2019; TechTarget, 2019). While these capacities are very useful, they make this protocol *heavyweight,* which means it is not optimal for decentralized, real-time communication (Richardson, 2019; TechTarget, 2019).

SOA usually utilizes a global data model across services, and databases are usually shared between services. This eases data access, so it is simpler to do transactions and query data (Bass et al., 2003; Richardson, 2019). Doing transactions means executing the logic of a software, and querying data is asking for a software's information. These characteristics make SOA's transactions ACID (atomicity, consistency, isolation, durability), the most reliable type of transactions in the event of errors and other failures (Haerder & Reuter, 1983). Given the shared database model and the heavyweight protocols used, services in a SOA architecture are often entire monolithic applications, bigger than those found in a *microservices* architecture.

2.7.3. The Microservices Architecture

Similar to SOA, the *microservices* architecture separates software in components. However, in the microservices architecture, services only have a few functions. Thus, a full *microservice* architecture is a collection of many interconnected *microservices* (Richardson, 2019). These microservices have a narrowly focused functionality and communicate through their Application Programming Interfaces (APIs). This helps the separation of concerns within an application, and it also determines the level of granularity for the system's architecture to the microservice. This greatly aids the *modularity* of a software system. Besides the technical benefits of the software being neatly separated into modules, microservices also help the organization of development teams for the company that will develop the software (Richardson, 2019). This enables *agile development* and state-of-the-art development philosophies like *DevOps*, a software development style that focuses on continuous planning, testing, and deploying (Digital.ai, 2020). Prime Vision and many other companies strive to develop software under this motto (Interviewee 1, Personal communication, July 15, 2020). Since this company is the one that would develop the Self-Organizing Logistics Planning System, this consideration is of special importance. I will return to this in the design phase of the project.

In the microservices architectural *style*, each service has an Application Programming Interface (API) address. This API address can be thought of as a phone number. Each service that wants to communicate with said service and the components within must call the correct number and wait for the service to pick up. The caller cannot pick up the phone for the service it is calling. Similarly, a service's API is the only way to access the service's resources. This boundary is difficult to bypass, thus helping the isolation and modularity of the application over time (Richardson, 2019).

Another key feature of a microservice architecture is that every service has a separate database, which contrasts with SOA's use of shared databases. As each service has its own database or *datastore,* services become properly isolated from one another. Microservices communicate through their APIs, and therefore their databases cannot be blocked by other microservices, like other request/response communication systems do (Richardson, 2019). This characteristic is called *loose coupling* and is essential for distributed systems. Figure 2-3 shows a comparison between the Service-Oriented Architecture and the Microservices architecture.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

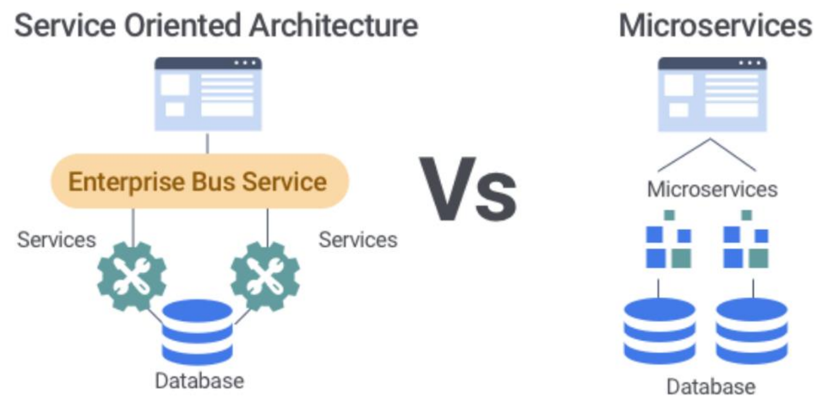

Figure 2-3. Comparing Service-Oriented Architecture and Microservices Architecture. Retrieved from XenonStack (2020).

As visible in Figure 2-3, both the SOA and Microservice architectures separate software functionality in services. However, services in a SOA architecture are bigger than microservices and often have a shared database between them. Furthermore, they communicate through an Enterprise Service Bus (ESB) and use heavyweight protocols (Bass et al., 2003; TechTarget, 2019). In a microservices architecture, however, microservices are loosely coupled, meaning they have separated databases and communicate directly using lightweight protocols (Richardson, 2019; XenonStack, 2020). From this view it can be inferred that both SOA and microservices function well for their desired purposes: SOA was developed to connect multiple coarse-grained services that can even cross organizations, while microservices aim to separate the functionality of a single piece of software into finely-grained services.

A microservices architecture grants certain managerial advantages that are unattainable for a monolithic application. If an application is a single executable file, then creating milestones for development is challenging. Companies in software development, including Prime Vision, utilize a best practice called continuous delivery and deployment (CI/CD), which focuses on continually delivering updated software during the development phase (Interviewee 1, Personal communication, July 15, 2020). CI/CD belongs to DevOps, the philosophy behind current best practices for software development (Digital.ai, 2020).

DevOps aims to streamline any software project's implementation phase, basing its philosophy on the principle of agile development. To comply with DevOps's mindset, pieces of software must have high levels of testability and deployability, and teams must be autonomous enough to deliver these updates without depending on other teams within the company (Digital.ai, 2020; Richardson, 2019). A company that correctly applies continuous delivery and deployment achieves automatically tested and deployable applications (Shahin et al., 2017). One way to achieve these quality attributes is to have the software divided into aptly sized services that teams can undertake on their own and program in a way that is highly testable and deployable. Microservices can thus be a driver for ease of development in the implementation phase of the project.

In the same manner that microservices offer many benefits, any distributed system also faces many challenges, so there are clear trade-offs in this design decision. In conclusion, choosing an architectural style is a crucial decision in the development of any software because it carries with it the earliest and most fundamental design decisions. I will revisit this question in the Conceptual Design phase of Chapter 5.

I synthesize the comparison between these 3 architectural styles in

Table 2-1:

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Table 2-1. Comparison between the Monolithic, Service-Oriented, and Microservices Architectural Styles.

Key Characteristics	Architectural Style		
	Monolithic	SOA	Microservices
Granularity	Single Package	Coarsely Grained	Finely Grained
Communication Protocols	Request/Response	Heavyweight Smartpipe	Lightweight messaging
Databases	Single Database	Shared Database	Database per Service
Transaction consistency	Transactions are local and guaranteed	Smart pipe orchestrates, a shared database makes transactions local	No orchestrator, the system depends on messaging
Security	Simple	Moderately Simple	Complex
Modularity	-	+	++
Modifiability	-	+	++
Performance	-	+	++
Availability	-	+	++
Ease of Development	-	+	++
Scalability	-	+	++
Deployability	+	++	++
Testability	-	+	++
Ease to integrate different software languages	-	+	++
Portability	-	+	++

2.8. *Summary*

In this chapter, I explained the main concepts of software architecture. The main objective was to illustrate these concepts, which will be fundamental in understanding the next Chapters of this thesis. The working definition of software architecture in this Thesis Project is "an abstract representation of a software system that showed a collection of components that are related to one another, that achieve software design purposes and show certain key characteristics of the system". This definition implies that the level of abstraction hereby used leaves out the specific workings of each component, and it focuses more on being comprehensible to all stakeholders. Next, I defined the main quality attributes of software design. These are important to define customer requirements in Chapter 3. Furthermore, I concluded that quality attributes depend on both the architectural and non-architectural aspects of a system and that some characteristics rely more on one type of aspect than the other.

I defined architectural *styles* as commonly found structures and ways of connecting elements in a family of software systems that solve specific architectural challenges, and architectural patterns were defined as a specific tool to solve a problem in a given context. By defining software architecture and discussing this strategy of selecting software patterns to design software architecture, we answer research sub-question 1.

Finally, I discussed more specific information about three architectural styles: the *monolithic* architecture, the *service-oriented* architecture, and the *microservices* architecture. I described the main benefits and drawbacks of each style and the hefty design decisions that come with them. The choice of one *style* over the others was not yet made, as choosing an architectural *style* can only be done once we know the desired Quality Attributes of the system. This is only possible after discussing the customer requirements, which I explore in Chapter 5.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

3. Last-Mile Delivery

To be able to design a software architecture that harnesses the SOLiD algorithm, I must first understand how the current last-mile delivery paradigm works. In this chapter, I study the technologies used today and the infrastructural capabilities of the usual sorting depot, which are important to design the software architecture of a feasible new system. The research questions I address in this chapter are:

2. *What are the current operations and capabilities of Carriers in last-mile delivery?*
3. *What are the desirable qualities of a logistics planning system for the involved stakeholders?*

To tackle these questions, I begin Section 3.1 with an explanation of the current operations for last-mile delivery and its shortcomings. I recollected this information by interviewing experts in the field. Furthermore, I find the key operational capabilities of the current hardware and software used in the last mile of delivery, which in sum answer research sub-question 2. In Section 3.2, I discuss the main stakeholders of the last-mile delivery environment, including their interests and desirable qualities in a logistics planning system. I gathered this information through expert interviews, a stakeholder analysis, and a literature study. From a design perspective, this discussion is crucial, as it sets the customer requirements. With this, I answer research sub-question 3. This chapter belongs to Phase 1 of the Systems Engineering Design Methodology by Dym et al. (2014), which is specified in the Research Flow Diagram of Figure 3-1:

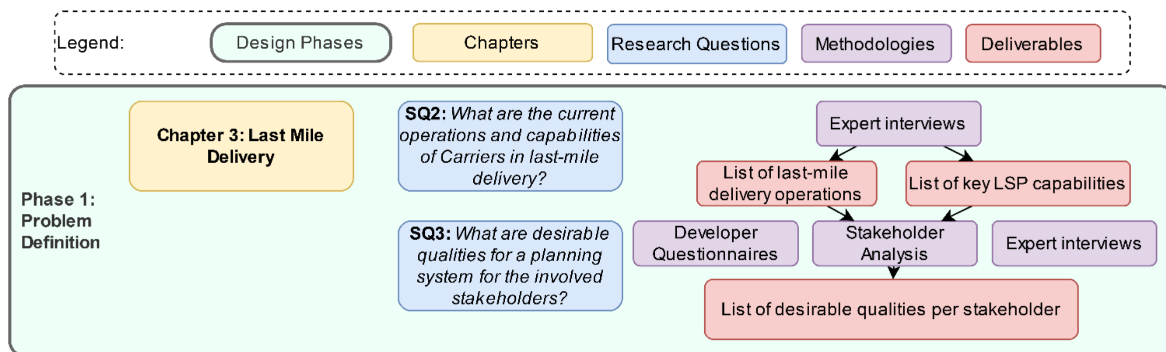

Figure 3-1. Phase 1, Problem definition, adapted from Systems Engineering Design methodology by Dym et al. (2014)

3.1. *Overview of the Last Mile of Delivery*

Before delving into the SOLiD algorithm, I will introduce the main activities of parcel logistics. More specifically, the area of interest is last-mile parcel delivery. The parcel delivery chain begins with the purchase of an article by a customer on an e-commerce platform. The e-commerce platform is owned by a shipper, which brings the customer's parcel to the carrier's warehouse. This is called the first mile of delivery. The carrier then transports the parcel from the Regional Warehouse to the Final Sorting depot, a segment called the middle-mile. Then, the last mile delivery process begins, where the carrier checks in the parcel into their system, which processes the transport request as shown by the data lines in Figure 3-2.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

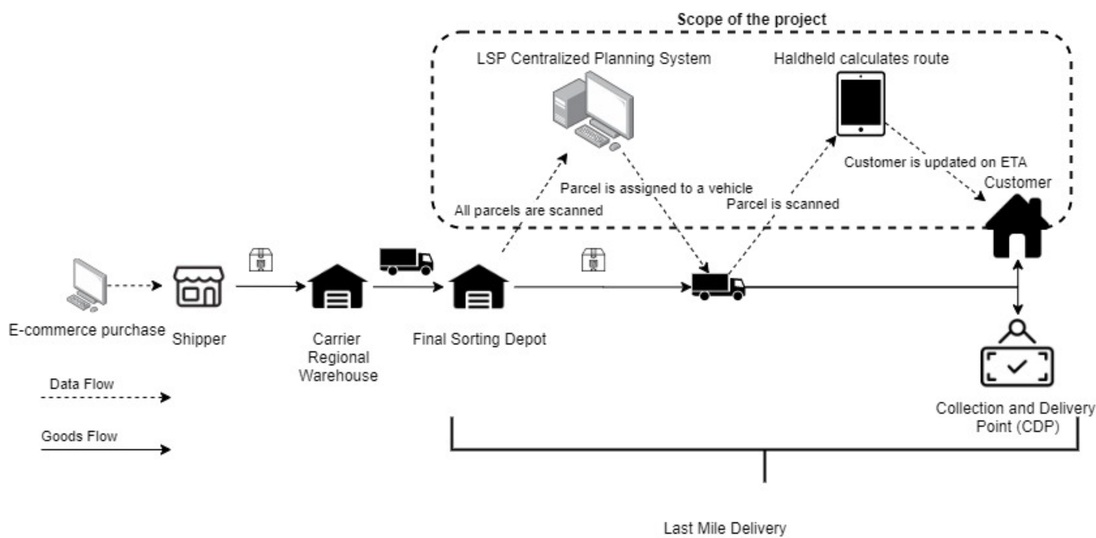

Figure 3-2. Simplified Parcel Delivery Chain (Interviewee D, personal communication, July 22, 2020; Interviewee E, personal communication, July 30, 2020).

To further understand the typical operations that span the last mile, I conducted several interviews with senior project managers at major Dutch Carriers (Interviewee D, personal communication, July 22, 2020; Interviewee E, personal communication, July 30, 2020). The sorting depot operations were defined as follows:

1. Parcels are loaded onto a conveyor belt where their barcodes are scanned.
2. The centralized planning system assigns the parcels to a vehicle, choosing from all available vehicles.
3. This selection is currently based on a set of pre-stipulated "delivery zones", which are aggregates of postal codes. These areas are fixed for various periods of time, ranging from 1 to 4 months, depending on the Carrier.
4. Then, parcels are loaded onto each vehicle. At this moment, parcels are scanned again with the driver's handheld device. This device calculates the optimal route for the driver for that day given the vehicle's parcels, minimizing travel distance.
5. The driver carries out the delivery to the end-customer, following the given route. If the delivery fails because a customer cannot receive the parcel, the parcel is left to a neighbor or is brought back to a Collection-and-Delivery Point CDP.

Figure 3-2 shows that when a parcel is sorted in a depot, a parallel digital process begins when the parcel is first scanned. This digital process is where the logistics planning system operates, which is my focus in this Thesis Project. Evidently, the transport of physical goods is crucial in the delivery process. However, in the logistics business, computer processing and advanced algorithms are often what give a competitive advantage to operators (Ergun et al., 2007). This project will focus on the software architecture of a planning system that utilizes the SOLiD algorithm and will extend the activities encapsulated in the algorithm to fit the entire last-mile delivery process in practice. For this design project, it is important to know the capabilities of the carriers in last-mile delivery, because it sets restraints for what is feasible in practice.

From this process, a few key capabilities are highlighted in Table 3-1:

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Table 3-1. Key Capabilities of Dutch Carriers in last-mile delivery

Key Capabilities
Carriers can scan a parcel in three moments: arrival at the sorting depot, loading onto a vehicle, and unloading from the vehicle.
The handheld devices used by the drivers can scan parcels and calculate daily routes based on the parcels held. This calculation tool is distinct from the centralized planning system used in the depot.
The handheld devices are connected to a Global Positioning System and can give out their position in real-time.
The centralized planning system in the depot is connected to the conveyor belts, which carry the parcels to their correct loading areas.

This list of capabilities of Carriers in the Dutch market answers Research Sub-Question 3.

After examining this method for last-mile delivery, several shortcomings can be identified:

- The delivery process is considered static due to these characteristics of a fixed delivery zone and one route calculation exclusively at the beginning of the day. This has two implications:
    a. The current system cannot cope with sudden changes in conditions, such as traffic jams and vehicle breakdowns.
    b. The current system uses static routes based on postal code groupings. From these postal groups, carriers make standard routes that work for an *average day*. This implies that the routes used daily are not specific to that day's parcels. Therefore, these routes are not optimal for each day's parcels.
- The delivery zones imply an extra layer of work for both the carriers and the shippers. The delivery zones dictate a certain organization inside the sorting depot that makes it effortful to change them. Furthermore, in the current system, the carriers send shippers a list of the current delivery zones. Shippers then mark the labels of their parcels with the correct delivery zone to simplify the delivery process for the Carrier. This implies more work and responsibilities for the shipper while adding a layer of complexity where mistakes could be made.
- Furthermore, the vehicle selection process does not occur simultaneously with the route calculation, rather, the selection is made first, and then the routes are made. Ideally, this would be calculated simultaneously, with a set goal to minimize the total combined distance traveled or cost among all vehicles. This separation in set delivery zones causes suboptimal routes for vehicles and uneven distribution of work for drivers and vehicles (Vlot, 2019).

Having explored the current operations, capabilities, and challenges of the last mile of delivery, I will now describe its main stakeholders.

3.2. Key stakeholders in Last-Mile Delivery

The parcel sector in logistics is considered separate from postal services. In a nutshell, the parcel sector deals mostly with small parcels (up to 31.5kg) and delivers them all across the world through sophisticated hub and spoke delivery networks (Ducret, 2014). These players perform the following activities: couriers perform point-to-point same-day delivery, express firms offer one to two-day delivery options with fixed time windows common in e-commerce, and the parcel sector focuses on the consolidation of lightweight parcels and usually has longer delivery times (Deckert & Görs, 2019). Lately, however, these services have grown increasingly indistinguishable, as many operators provide all three and use similar networks for all of them (Ducret, 2014). While the parcel sector also participates in the B2B (business-to-business) industry, the B2C (business-to-consumer) sector is more important for last-mile delivery. Due to the rise of e-commerce, this sector has grown at an unprecedented rate, and many companies are racing towards market dominance (Ducret, 2014). For example, due to the coronavirus

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

crisis, the e-retail business grew 44% in the Netherlands and 188% in Belgium in 2020 alone (Emarsys, 2020). The rising profitability of the e-commerce market has also meant that many post operators who previously focused on physical mail are also diversifying into parcel delivery and express services, creating a blurred lines environment. I group all these players under the name of *Carriers*.

As seen in the delivery process of Figure 3-2, many actors interact to deliver a parcel. The three most important actors are the customer, who places the order, the shipper, who sold the product to the customer, and the carrier, who transports the shipper's parcel to the customer. There are more stakeholders, as Carriers also sub-contract drivers, and many authors also add the government and other local authorities as indirectly related entities in the process (Harrington et al., 2016; Russo & Comi, 2011; Taniguchi & Thompson, 2002). However, these actors are not as prevalent for the design of a new logistics planning system that uses the SOLiD algorithm, and thus their interests will not be explored in detail.

In this Thesis Project, I will consider a special stakeholder: The Carrier's software provider, a role hereby filled by Prime Vision. I will describe the main characteristics of the principal actors and their desired characteristics in a logistics planning system. These will be separated into *customer requirements* and *functional requirements*. Customer requirements refer to system *characteristics*, and functional requirements refer to things a system *does*. In essence, functions are things that the planning system should be able to *do*, and characteristics are things the planning system should *have* (Dym et al., 2014).

3.2.1. Customers

Customers initiate the delivery chain by purchasing an item and its supplementary services at an e-commerce or retailer. In the large-scale survey done by Okholm et al. (2013), customers' main interests are based on service quality, speed, reliability, flexibility, and cost. Research shows that cost is the customer's decisive factor for delivery alongside home delivery, followed by track and trace options and digital notifications for their shipments (Okholm et al., 2013).

To comply with the customer's requirements, Carriers need to have responsive logistics networks and clear information flow in their logistics planning systems. These requirements are further pressured by the increasing demand for next day delivery. This demand is sharper in urban areas, which also host a denser number of customers (Harrington et al., 2016; Okholm et al., 2013). Flexibility has become more valuable for customers, be it regarding payment options, range of services, and home delivery. Extra services are welcome as long as they give customers more choices and flexibility, such as "green delivery" and other options (Harrington et al., 2016). In this context, Green delivery refers to delivery that focuses on the lowest possible environmental impact. The SOLiD algorithm developed by Vlot exploits this opportunity by giving all customers a choice between the lowest cost, fastest delivery, and lowest carbon emissions (Vlot, 2019).

While costs and next day delivery affect customer's requirements, they do not depend on the architecture of the planning system. The customer's requirements are:

- The system needs to include Track & Trace functionality with digital notifications of the Estimated Time of Arrival (Information availability and an accurate delivery time window to receive the packet).
- The system could support additional options like Green Delivery.

3.2.2. Shippers

Shippers are the owners of the platforms through which customers initiate the delivery chain. Example shippers are Amazon, Bol.com, Cool Blue, and other e-retailers. While some shippers decide to deliver items themselves, the majority subcontracts a Carrier for delivery (Okholm et al., 2013). To maintain their competitive advantage, they must streamline the purchase process and offer the best possible service to their clients, which translates into a careful selection of Carriers. Therefore, Shippers have significant market power in the delivery process. Like customers, they prefer fast, low-cost delivery, and added features such as insurance and track & trace. Information availability and a predictable delivery window are also key features for shippers. In the tightly knit market of the Netherlands, shippers want to show the expected time window of delivery with an accuracy of 3-4 hours **at the**

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

moment of purchase, and Carriers in the Netherlands try to accommodate for these features (Interviewee D, personal communication, July 22, 2020; Interviewee E, personal communication, July 30, 2020).

To cope with the increasingly demanding shipments for their customers, above half of the European Shippers utilize more than one carrier to perform deliveries (Okholm et al., 2013). Therefore, the combination of carriers leads to superior transport options, although it can be time-consuming for Shippers.

From the literature study, the relevant functional requirements for the shippers are:

- The system needs to include Track & Trace functionality with digital notifications of the Estimated Time of Arrival (Information availability and an accurate delivery time window to receive the packet) because it improves customer experience.
- The system could support additional options like Green Delivery because customers favor additional options.
- A system that combines carriers under a single platform is desirable because it would lower the prices for shippers.

3.2.3. Carriers

Carriers include all transportation and logistics companies that perform courier, express, and parcel delivery services. They are central to the delivery chain because they perform most of the transportation activities within it. In general, carriers seek to leverage their assets to provide satisfactory services to their customers while complying with local regulations (Gevaers et al., 2011; Harrington et al., 2016; Litman & Burwell, 2006). In later years, competition has grown considerably, causing significant duplication in the last mile (Allen et al., 2018). This implies that many carriers offer last-mile services to the same neighborhoods, which greatly increases the total amount of kilometers driven in the city (Gevaers et al., 2011).

To find key customer requirements for a new planning system for carriers, I conducted semi-structured interviews with two project managers at major Dutch Carriers. With this activity, I can also corroborate certain Carrier interests found in the literature. I supported the interviews with a small survey that listed customer and functional requirements for a logistics planning system and attached a ranking system to them. I only mentioned the items of this list after asking the interviewees directly for their own customer requirements, to avoid biasing the interviewees. This ranking system went from 1 (not important) to 4 (very important). A final rank of 5 (indispensable) was utilized to describe *constraints*, or characteristics that would be absolutely necessary for a logistics planning system. The Customer and Functional Requirements I highlighted during the interviews with the Carriers are listed in Table 3-2:

Table 3-2. Customer and Functional Requirements highlighted in the interviews with the Carriers

Customer and Functional Requirements
Compatibility with current hardware/legacy systems
Compatibility with current operational philosophy
Capacity to enable multi-modal transport
24/7 Availability
Robustness
Minimization of Costs
Ease Depot Operations
Security
Accurate Estimated Time of Arrival (ETA)
Capacity to create Data for KPIs
Data Transparency
Addition of Environmentally friendly options for delivery

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Carriers have a straightforward objective of reducing operational costs and increasing revenue, considering minimization of costs as one of their largest priorities (Interviewee D, personal communication, July 22, 2020; Interviewee E, personal communication, July 30, 2020). However, other governmental pressures have stirred carriers towards controlling their environmental and societal impact (Gevaers et al., 2011). This leads to a scenario where carriers must minimize costs while also lowering their carbon footprint. Green options for delivery become a desirable option to have to satisfy customers and comply with regulations (Interviewee D, personal communication, July 22, 2020; Interviewee E, personal communication, July 30, 2020; Okholm et al., 2013).

For carriers, distance is not necessarily the best parameter to calculate costs (Interviewee D, personal communication, July 22, 2020). To minimize costs, Interviewee D suggested designing a system to achieve equal flow, which refers to distributing workloads throughout the day. In his opinion, this design choice had more impact on the overall costs of operations than minimizing distance in the routing algorithm (personal communication, July 22, 2020). The efficiency of the system was considered a vital means to achieve cost minimization. Furthermore, there are several operational inefficiencies like empty-running. Empty running typically occurs after a vehicle has delivered all of its parcels and returns empty to the depot (Ergun et al., 2007). This is a major inefficiency that can have a significant effect on the overall profitability of the system (Allen et al., 2018; Özener & Ergun, 2008). Thus, carriers are also interested in reducing the amount of empty running, a fact corroborated by Interviewees D and E (personal communication, July 22, 2020; personal communication, July 30, 2020).

Field experts also mentioned that leveraging the current hardware infrastructure is an important requirement for a new planning system (Interviewee D, personal communication, July 22, 2020; Interviewee E, personal communication, July 30, 2020). The current hardware infrastructure refers to the capabilities carriers have in the last-mile of delivery, which were listed in Table 3-1. Both Carrier project managers agreed that hardware changes were possible but slower. Thus, having a system that can leverage their current assets has more chances to be acquired in the medium term. Interviewee E considered this to be a characteristic of a *modular* system, that can be adapted easily into existing infrastructure (personal communication, July 30, 2020). I will emphasize this aspect on the design phase. In the same vein, these experts considered compatibility with current operational philosophy as important (Interviewee D, personal communication, July 22, 2020; Interviewee E, personal communication, July 30, 2020). Both experts acknowledged that it is impossible to innovate without changing operational philosophy, but that changes need to be executed progressively. This will be part of the recommendations left for Prime Vision in the implementation phase, detailed in Chapter 9.

From these discussions, I conclude that a new system must accomplish two distinct objectives: to leverage current operator capabilities and generate results without altering operations completely. In the opinion of major Dutch Logistics Operators and experts at Prime Vision, the ideal new system would be easier to operate than its predecessor and that offers benefits unattainable by the current system (Interviewee A, personal communication, July 15, 2020; Interviewee B, personal communication, July 22, 2020; Interviewee C, personal communication, July 31, 2020; Interviewee D, personal communication, July 22, 2020; Interviewee E, personal communication, July 30, 2020). In this context, the current system is the system described in Section 3.1.

The capacity to enable multi-modal transport, or changing from one mode of transport to another, was not valued as highly, being considered "somewhat important" and "important" by Interviewees D and E, respectively (personal communication, July 22, 2020; personal communication, July 30, 2020). This could be explained by the fact that current systems do not combine these alternative modes of transport because the handling is too complex for a centralized system. With a self-organizing system, the capacity to change from one mode of transport to another could well become a leverageable characteristic and have more perceived value.

Several other points were tackled in the interviews with these experts. Carriers considered the 24/7 availability of the system as a constraint, along with data transparency, and the overall security of the system (Interviewee D, personal communication, July 22, 2020; Interviewee E, personal communication, July 30, 2020). When asked about security, respondents mentioned that they required compliance with security standards from their software providers. Therefore, security must comply with current best practices.

Furthermore, the ability for a system to generate data to then build KPIs and other analytics was considered *important*, and accurate estimation of the Estimated Time of Arrival was considered *very important* because it is a

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

customer "touchpoint". Furthermore, giving predictable time windows reduces the chance of failed deliveries, a major source of additional costs for Carriers (Florio et al., 2018).

When delivery fails, the driver must leave the parcel with a neighbor or transport the parcel to a nearby depot or alternative pick-up location. This is costly to carriers because of double handling, and it also damages customer satisfaction and firm reputation, especially when customers did not explicitly agree to these alternative delivery locations (Florio et al., 2018; Okholm et al., 2013). Hit-rate can be improved by creating "customer availability profiles" based on historical data and by performing delivery at the promised time to customers (Florio et al., 2018). Interviewee E corroborated this practice in his company (personal communication, July 30, 2020). Carriers also valued reliability against unexpected events like traffic jams and vehicle breakdown very highly (Interviewee D, personal communication, July 22, 2020; Interviewee E, personal communication, July 30, 2020). Note that this reliability concerns the routing algorithm, and not the robustness of the planning system's software. The planning system's robustness is included under the term *availability*, as discussed in Section 2.3.

In the current practice, the handheld device that calculates the last-mile route for each vehicle triggers the notification sent to the customer with the Estimated Time of Arrival of the parcel. Interestingly, Interviewee D stated that the inner-city routes were left entirely to the expertise of the driver, not relying on the compliance with the digitally sketched route (Personal Communication, July 22, 2020). This could become a barrier to the implementation of a new system that does rely on digitally sketched routes.

The two interviewees had mostly matching preferences for the characteristics of a new logistics planning system. However, when considering the capacity for a planning system to ease depot operations, Interviewee D considered it *very important,* while Interviewee E considered this to not be the responsibility of the planning system, therefore being qualifying this characteristic as *unimportant.* This point will hereafter be included under "compatibility with current operational philosophy", as operations span depot operations, and the envisioned planning system would affect operation within it. I summarize the desirable qualities and functions of a logistics planning system for Carriers in Table 3-3.

Table 3-3. Summary of requirements for a logistics planning system for Carriers.

Customer Requirements	**Importance**
24/7 Availability of the service	**Indispensable**
Security	**Indispensable**
Data Transparency	**Indispensable**
Modularity	**Very important to Interviewee E**
Compatibility with current hardware/legacy systems	**Important**
Compatibility with current operational philosophy	**Important**
Functional Requirements	**Importance**
Accurate prediction of ETA within a 1-2 hour window.	**Indispensable**
Minimization of costs	**Very important**
Ability to increase capacity utilization	**Very important**
Robustness against traffic jams, vehicle breakdown, etc.	**Very important**
Minimizing costs by distributing workloads	**Very Important to Interviewee D**
Environmentally friendly options for customers	**Important**
Capacity to enable Multi-Modal Transport	**Somewhat important**

3.2.4. Delivery Drivers

Delivery Drivers often have conflicting interests to those of carriers, especially if the driver does not belong to the large organization the parcels belong to. In these agreements, sub-contracted drivers are often paid by each parcel delivered and the number of stops made, with some compensation for distant stops (Krajewska & Kopfer, 2009). Furthermore, delivery drivers often know more about the terrain than the standardized platforms that create their optimal routes (such as their particular vehicle not being able to cross a particular lightweight bridge), which is why they sometimes do not follow the routes calculated exactly (Interviewee D, personal communication, July 22, 2020). Delivery drivers also need to report if a delivery failed and a parcel needs to go to a Collection and Delivery

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Point (CDP). Therefore, the need for an effortless way to report this to the overall system becomes important. Another point is the driver's right to privacy, as mentioned by Interviewee D (Personal Communication, July 22, 2020). If data is tracked 24/7 on each vehicle, the privacy of each driver would be compromised.

From the literature study and the interviews carried out with the project managers at major Dutch Carriers, the two main points to stress from delivery drivers are:

- Ease of reporting failed deliveries.
- Protecting the privacy of the drivers.

3.2.5. Software Providers for Carriers (Prime Vision)

Prime Vision is in a unique position in the last-mile delivery market because it provides software for multiple Carriers. This implies that the quality of Prime Vision's software products directly affects the efficiency of its customer's delivery operations. Prime Vision functions like a provider and a consultant at the same time, helping customers with new products, and also testing and developing products with its customers (Interviewee C, personal communication, July 31, 2020).

Current Prime Vision products include an existing centralized planning system that works in the "middle mile", or the point between the Regional Sorting Center and the Final Sorting Depot in Figure 3-2. This is called a sorting system and it is the fixed routes systems that the Carriers from Interviewees D and E employ. Other products are the barcode technology utilized to scan parcels, and the handheld apps for the use by drivers (Interviewee C, personal communication, July 31, 2020). In the last mile delivery paradigm, Prime Vision wants to bring innovative products for its customers and expand its current wallet with its customers.

Prime Vision wants to utilize the SOLiD algorithm as an all-encompassing platform that could accommodate several Carriers and eventually achieve collaborative delivery. This would put Prime Vision in a privileged position in the market as a broker between Carriers and a facilitator for end customers. In the short and medium term, however, Prime Vision's main customers are Carriers and it designs software to suit their needs (Interviewee C, personal communication, July 31, 2020). Therefore, the planning system must be designed to comply with Carrier requirements while keeping it agnostic or compatible with many types of information. For Prime Vision, the best-case scenario is for the logistics planning system to work directly from the moment an item is bought at an e-commerce platform, thus uniting the first and last mile of delivery (Interviewee A, personal communication, July 15, 2020; Interviewee B, personal communication, July 22, 2020; Interviewee C, personal communication, July 31, 2020).

From the interviews with project managers and the commercial director of Prime Vision, I highlighted several important aspects of a new logistics planning system that Prime Vision would want to develop. I only mentioned the items of the list after asking interviewees for their requirements directly. This was done to avoid biasing the interviewees. In agreeance with the managers of major Dutch Carriers, the managers at Prime Vision anticipated that the compatibility with current operational philosophy would be important —and that Prime Vision would have an important role in facilitating the change between the two systems. The Customer and Functional Requirements used to aid the interviews are listed in Table 3-4:

Table 3-4. Customer and Functional Requirements highlighted in the interviews with Prime Vision

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Customer and Functional Requirements
Capacity to integrate multiple carriers under one platform
24/7 Availability of the service
Performance and Efficiency
Modularity
Modifiability
Testability
Deployability
Conceptual integrity
Security
Portability (from one software/hardware platform to another)
Compatibility with current operational philosophy
Use of standard languages and elements
Compatibility with current hardware/legacy systems

Following their business plans to offer more service and subscription-based offerings, Prime Vision managers also rated 24/7 availability as *very important*, as dictate the modern standards for software products. Billing said product would also have to be in accordance with that. In my interview with Interviewee C, the Commercial Director of Prime Vision, I suggested a usage-based subscription business model for the Self-Organizing Logistics Planning System. This proposition was agreed upon enthusiastically (Interviewee C, personal communication, July 31, 2020). I will revisit this point in Chapter 7.

Other *very important* characteristics were the Testability and Deployability of the system, which goes with Prime Vision's philosophy to comply with *DevOps* best-practices, as mentioned in Section 2.7.3 (Interviewee A, personal communication, July 15, 2020). Performance was also mentioned as a key priority, as Prime Vision considers it to be the point where the new planning system can outdo the competition (Interviewee B, personal communication, July 22, 2020). Time is a scarce resource in everyday operations, and the velocity at which the software can compute the routes is paramount to reducing the time between drivers reaching the depot in the morning and leaving for deliveries. This is consistent with the experience and opinion of Interviewees D and E from major Carriers (personal communication, July 22, 2020; personal communication, July 30, 2020).

Portability between software platforms was considered *important* to avoid vendor lock-in, especially when it comes to potential cloud solution vendors. As I describe later in section 6.1.5, this is the most logical solution for deploying most large-scale decentralized systems. Prime Vision also considered Conceptual integrity *very important*, and as I explained in Section 2.2, software architecture is key in achieving this characteristic. The use of standard elements was deemed *somewhat important* by Interviewee A, and *important* by Interviewee B.

The SOLiD algorithm was also discussed with the development team at Prime Vision, and potential implementations were theorized. After several conversations and explanations about the algorithm's functions, a questionnaire was sent to the developers to rank the importance of several quality attributes and functions for a logistics planning system that used this algorithm. This questionnaire and its results are found in Appendix A. The opinion of software developers familiar with the logistics field helps understand technical priorities for a logistics planning system.

The developers anticipated that Carriers would have high availability and security requirements; and they also considered that having high performance and the capacity to host multiple carriers under one platform would be very important, coinciding with Prime Vision's management team. Furthermore, developers noted that *modularity* and *modifiability* would be key for the development and maintenance phase of the project. Interestingly, developers did not consider compatibility with legacy systems or operational philosophy to be a factor worth designing for. I revisited this question with the developers in person, they explained that in their industry most pieces of software become obsolete within some years and that it is normal for full overhauls to happen. This also explains the low value assigned to the use of standard languages. This particular point was corroborated by Interviewees B and E, enhancing the internal validity of these findings.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Table 3-5 summarizes the important characteristics for a planning system that harnesses the SOLiD algorithm for Prime Vision's management and development team.

Table 3-5. Customer and Functional Requirements for a logistics planning system for Prime Vision's management and development team.

Customer Requirements	Importance
Conceptual integrity	Very Important
Subscription-based billing	Very Important
Compatibility with current operational philosophy	Very Important
24/7 Availability of the service	Very Important
Performance and Efficiency	Very Important
Modularity	Very Important
Modifiability	Very Important
Testability	Very Important
Deployability	Very Important
Security	Very Important
Portability (from one software/hardware platform to another)	Important
Compatibility with current operational philosophy	Important
Compatibility with current hardware/legacy systems	Somewhat important
Use of standard languages and elements	Somewhat important
Functional Requirements	Importance
Capacity to enable Multi-Modal Transport	Very Important
Capacity to integrate multiple carriers under one platform	Very Important

3.3. Summary

In this Chapter, I described the main characteristics of the last-mile delivery scenario, supported by a study of the logistics literature and interviews with industry experts. Then, I discussed the main operations for last-mile delivery in the Benelux area. Through this investigation, I gained important insights into the capabilities of Carriers in the last-mile delivery, an important aspect of designing the software architecture of a new system, answering research sub-question 2. This sub-question pertained to the capabilities of Carriers in last-mile delivery. Importantly, my answer to this question is inherently linked to the Benelux area.

Afterward, I carried out a stakeholder analysis for the last-mile delivery scenario, highlighting their customer and functional requirements for a logistics planning system. I found that Availability, Security, and Data Transparency were the most important requirements for Carriers and that the most important functional requirement for the system is being able to calculate the Estimated Times of Arrival with a 1 or 2-hour window. Prime Vision, on the other hand, values scalability by future-proofing the planning system, regarding the capacity to integrate multiple

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

carriers under one platform and enabling Multi-modal transport as the most desirable functions for the system. With regards to the Quality Attributes, Prime Vision valued Performance, Modularity, Modifiability, Testability, and Deployability as the most important characteristics along with the capacity to have a subscription-based billing. These customer requirements answer research sub-question 3.

In the next Chapter, I discuss the SOLiD algorithm and extract its functional requirements.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

4. The SOLiD Algorithm

Before designing a logistics planning system that uses the SOLiD algorithm, I must first understand the algorithm's logic. The SOLiD algorithm was developed to cope with the increased complexity of collaborative delivery and the limited scalability and resource utilization that current operations have. Vlot's algorithm would theoretically help address part of the shortcomings of current last-mile delivery operations shown in Section 3.1. Basing the design on the state of the art in self-organizing systems, the algorithm changes the point of view from which last-mile delivery is calculated. Namely, this algorithm focuses on each individual parcel, with its own preferences for transportation modes.

In this chapter, I describe the SOLiD algorithm and develop several BPMN diagrams that illustrate its stepwise functioning. The research sub-question I address in this chapter is:

4. *What are the main functional requirements of a self-organizing logistics planning system?*

To answer this question, I look into the source material of Thymo Vlot (2019). From his descriptions of the algorithm's main functions, I develop a series of graphical representations in Business Process Model Notation (BPMN) diagrams, which I verify with an interview with Vlot himself. I combine the BPMN diagrams with the creation of user *stories*, which help me understand the functional requirements of the algorithm when applied in last-mile delivery. The concept of user *stories* comes from agile software development, and it represents crucial system scenarios where a series of actors perform certain activities. *Stories* give a clear idea of what needs to occur given a particular situation and are thus very helpful in decomposing the system. In this thesis, I am following the guidelines to create these stories from Richardson's Microservices Patterns (2019). The Research Flow Diagram of Figure 4-1 summarizes the research questions, activities, and deliverables for this chapter.

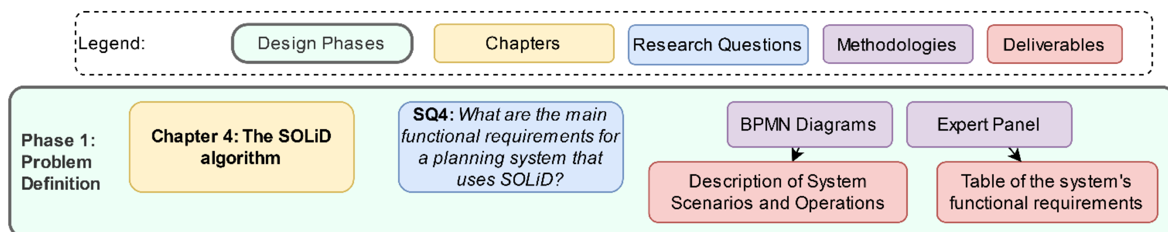

Figure 4-1. Phase 1, Problem definition, adapted from Systems Engineering Design methodology by Dym et al. (2014)

4.1. *The main functionality of the Algorithm*

In the SOLiD algorithm, Vlot redefined the delivery process of sorting, loading into a vehicle, and delivering a parcel. Vlot (2019) established three main actors in the process: parcels, vehicles, and a central platform. Both vehicles and parcels are intelligent, and they negotiate with one another, connected via a central platform. This results in a decentralized system that does not depend on a single entity to compute all routes and make all decisions. It does, however, count on one entity that knows the position of all vehicles at all times: the central platform. I will expand on this subject in later sections.

In the algorithm, vehicles start the day with empty itineraries, and they must register as "available" to a central platform, thereby announcing that they want to be notified when there are parcels to be delivered. Furthermore, vehicles have a preferred "center of delivery", which helps determine if they are going to be considered for a particular parcel's auction (Vlot, 2019, p. 45). After they are notified, they can elicit a bid to transport the parcel. If a vehicle wins the auction, it can take the parcel. Figure 4-2 shows the basics of the algorithm's auctioning process.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

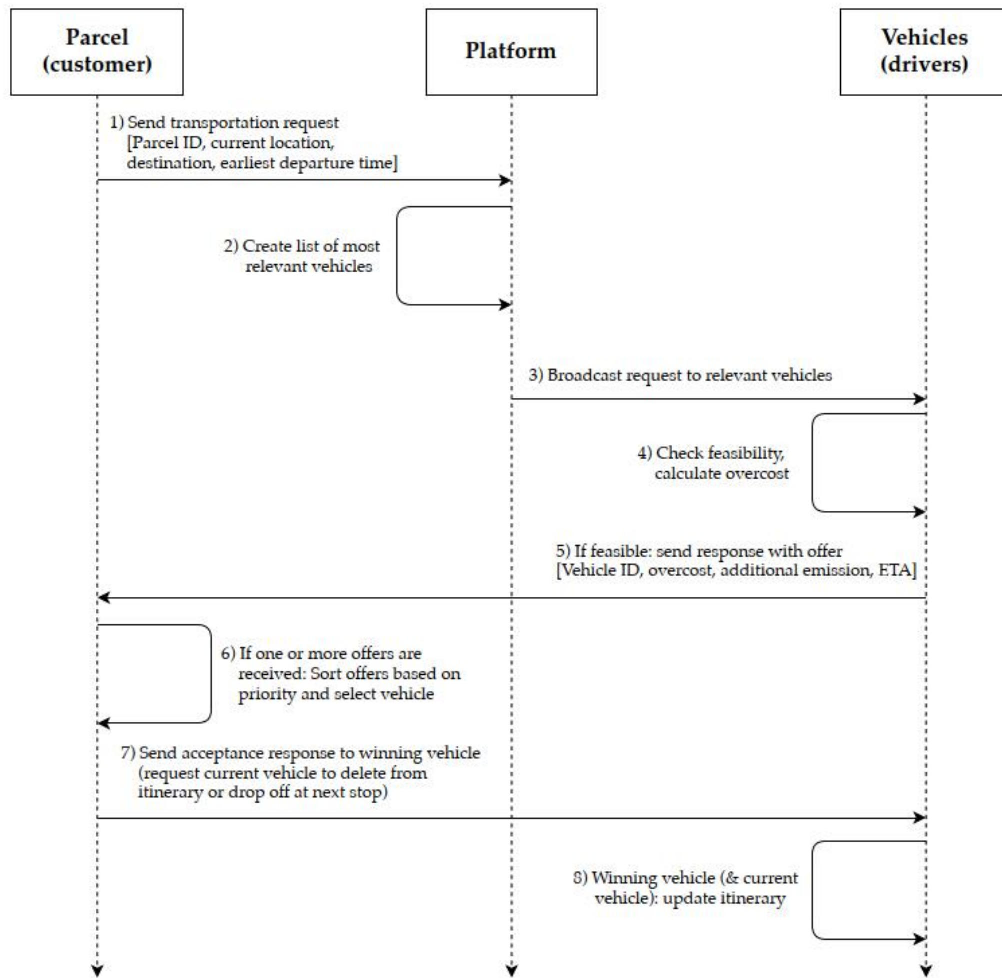

Figure 4-2. UML Sequence Diagram for the SOLiD algorithm's auctioning process. Retrieved from Vlot (2019, p. 63)

In the original algorithm, the process begins with the parcels, who send a transportation request to the central platform, indicated by activity 1 in Figure 4-2. The central platform then filters the "relevant vehicles" for a particular transport request, denoted as activity 2 in Figure 4-2. Then, the central platform notifies the "relevant vehicles" about that particular parcel, which then gets a chance to send their transport bids. Then, vehicles check the feasibility of taking said parcel on their trip, and if the parcel's pick-up and drop-off points are feasible, they will send a bid, encapsulated in activities 4 and 5 in Figure 4-2. Parcels then must choose from all bids received from vehicles (activity 6), which is done on a first-come-first-serve order. Parcels then send an acceptance response to the winning vehicle, denoted by activity 7 in Figure 4-2. There is an important characteristic of the self-organizing algorithm at this point, which is that parcels can request for transport while in a vehicle that is already transporting them. Therefore, if the parcel is "en route" and the parcel accepts a bid from another vehicle, then activity 7 must include a drop-off request from the current vehicle. The final activity in the auctioning process is the update of the itinerary for the winning vehicle.

4.2. Describing the activities of the SOLiD algorithm in a real delivery scenario

The UML sequence diagram of Figure 4-2 gives a good overview of the auctioning process that occurs in Vlot's algorithm (2019). However, to design the software architecture of a logistics planning system based on the algorithm, a more granular exploration of the algorithm's functions is needed. Furthermore, the process needs to be expanded for the whole delivery process, not only the auctioning process. Therefore, I developed several BPMN diagrams to show these activities in greater detail, adding activities from the real delivery process that are missing in the SOLiD algorithm. I extract *stories* from each BPMN diagram, which will help me define system operations. System operations are necessary to decompose the system in the following chapter.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

In BPMN diagrams, each color-coded horizontal lane represents an actor, and the boxes represent its activities. Dotted lines refer to messages sent from one actor to the other, and solid lines refer to the main flow of activities. Circles determine the start and the end of the process.

Step 1: Parcel Creates Transport Request

The activities of the SOLiD algorithm begin when the parcel requests transport, as shown in Figure 4-3:

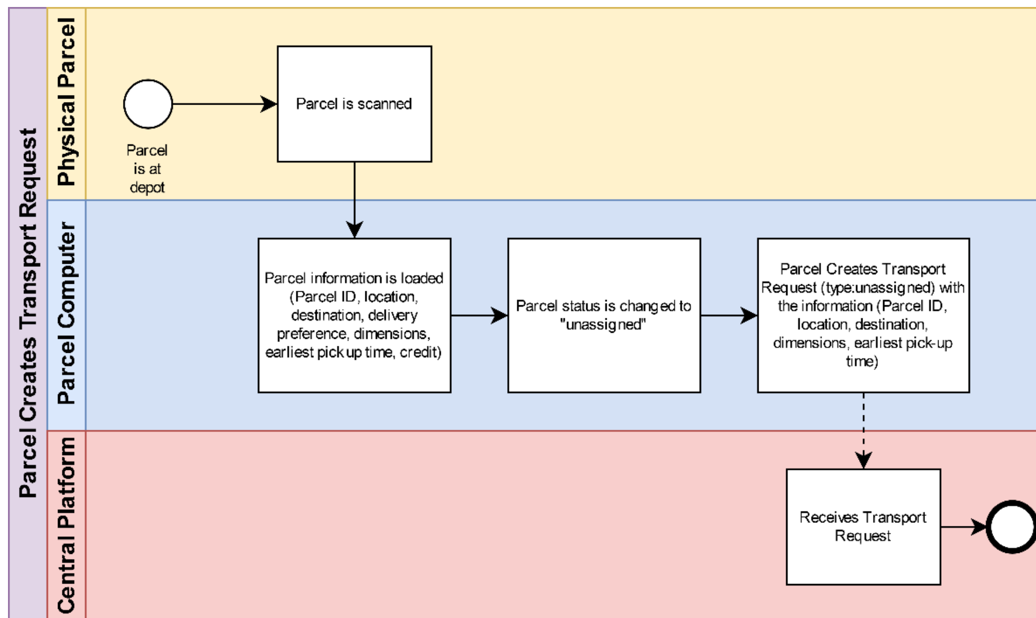

Figure 4-3. Step 1: Parcel Creates Transport Request

As visible in Figure 4-3, the precondition for this process is that the parcel is at a depot. The parcel is then scanned, and its information is loaded. This information includes its Parcel ID, location, dimensions, earliest pick-up time. After the parcel knows its own information, it can create a transport request with the relevant information, which is its Parcel ID, location, dimensions, earliest pick-up time. Because this parcel is not yet assigned to any vehicle, the transport request is of "type: unassigned" which means that it will have priority over requests by parcels that are already assigned. This request is received by a central platform, which marks the end of this process. As I show in the next step, this central platform is responsible for communicating this request to vehicles.

From this process, I extract the "Parcel Creates Transport Request" *story*:

Given a parcel

And a medium to send transport requests to vehicles

And a set of parcel preferences

And a delivery destination

And a set amount of credit available for the parcel

Then the parcel creates a "type_unassigned" transport request

And the transport request is sent to the central platform.

And the central platform finds relevant vehicles to send the request to

This is the beginning of the transport auction process that was defined in Section 4.1, written in a format that shows the relationships that are necessary to carry out the main action of the story, in this case, *creating the "type_unassigned" transport request*. The "medium to send transport requests to vehicles" is the Central Platform.

This story hints that parcels must know their destination, dimensions, and preferences. Therefore, each parcel should be able to store said information somewhere. Furthermore, it also hints that parcels must have a way of

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

holding *credit*. In other words, parcels should have a *wallet*. It also shows that parcels must be able to create a transport request by checking its own information, so parcels need to be able to create and send its transport requests somehow. In other words, parcels need the ability to create a transport request from its information (destination, dimensions, etc.). Thus, parcels need processing power to create these requests, which should be created in a readable format for vehicles.

Step 2: Central Platform Finds Relevant Vehicles

The next step involves the communication of the transport request to relevant vehicles, which is performed by the Central Platform. This platform knows where vehicles are and where they want to perform deliveries (delivery center). Figure 4-4 details this process:

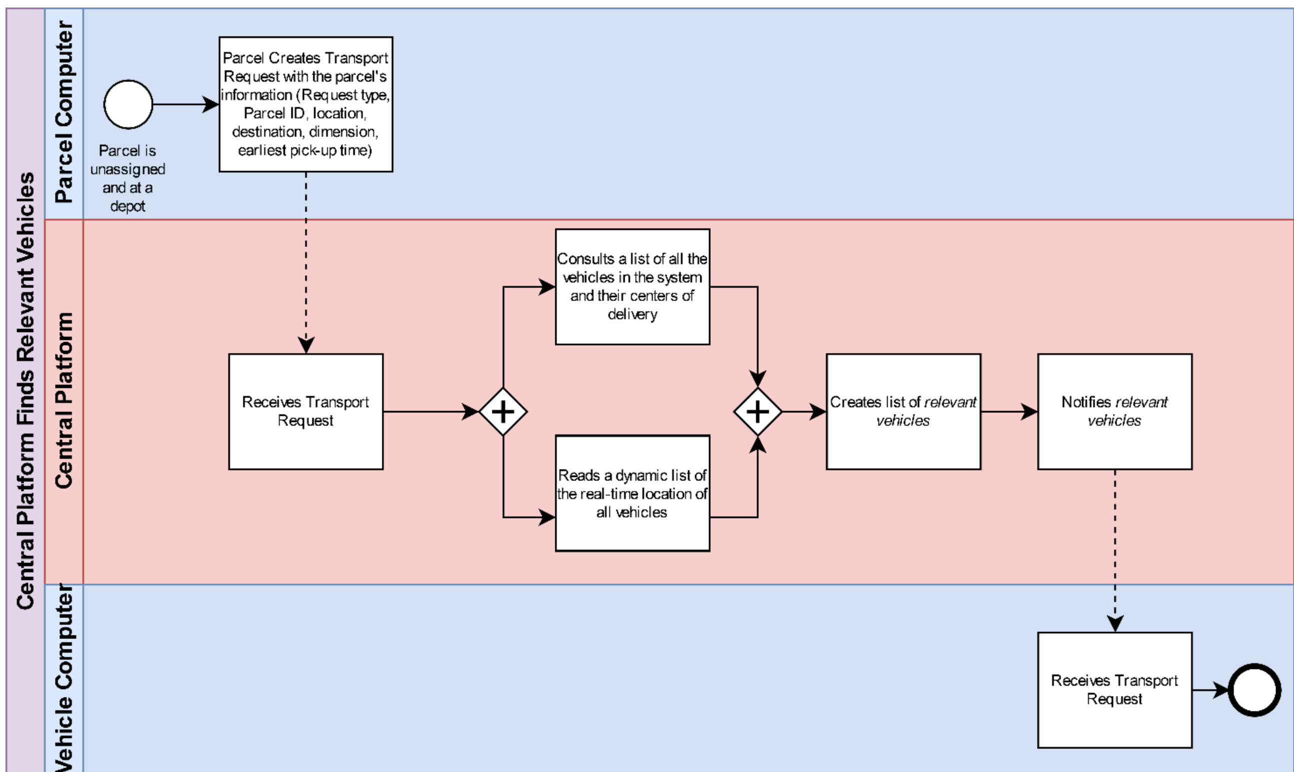

Figure 4-4. Step 2: Central Platform Finds Relevant Vehicles

The process begins when a parcel creates a transport request. The Central Platform then consults a list of all the vehicles in the system and their centers of delivery, which approximate to their preferred delivery zone. This is carried out while simultaneously checking every vehicle's location, stored in a dynamic list that is updated by the vehicles in real-time. The Central Platform then can create a list of *relevant vehicles* to which to send the transport request to, doing so based on the read position of the vehicles and their delivery center. The process ends when the *relevant vehicles* receive the transport request.

From this process I extract the *Find Relevant Vehicles* story:

Given a central platform

And a "type_unassigned" or "type_assigned" transport request

And a list of all vehicles that are participating in parcel delivery with their delivery centers

And a dynamic list with the position of all vehicles that are participating in parcel delivery

Then the central platform executes the "create list of most relevant vehicles" algorithm, filtering vehicles based on the distance between the vehicles and the parcel.

And a message is sent to the "most relevant vehicles" with the "type_unassigned" or "type_assigned" transport requests

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

This story hints that the central platform must be able to receive transport requests from parcels and send these to vehicles. Thus, the Central Platform must be able to communicate. Furthermore, the central platform needs to be able to compute the *"create list of most relevant vehicles" algorithm*, and thus would require some processing power.

Step 3: Vehicle announces participation in the system

To be able to be considered by the central platform, vehicles must first announce their participation in parcel delivery, shown in Figure 4-5:

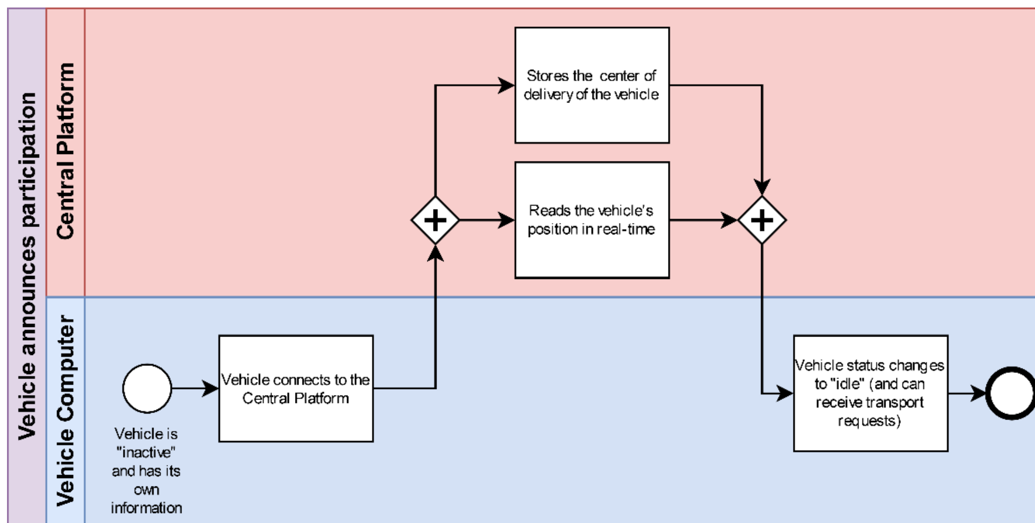

Figure 4-5. Step 3: Vehicle announces participation in the system.

There are two preconditions for this process. First, the vehicle is not yet in the system and thus its status is "inactive". Second, the vehicle knows its own information. The Central Platform then reads the vehicle's position in real-time and stores the vehicle's center of delivery. This way, it can consider the vehicle as a potentially *relevant vehicle* when it receives a transport request from a parcel, denoted by the "idle" status.

With this I wrote the *Vehicle announces participation* story:

Given a vehicle

And the vehicle status is inactive

And the vehicle stores its information (Vehicle ID, location, center of delivery, speed, emissions, etc.)

And a Central Platform

Then the vehicle connects to the Central platform

And the Central platform reads the vehicle's position in real-time

And the Central Platform stores the vehicle's Center of Delivery

And the vehicle status changes to "idle"

These last two *stories* contain one of the fundamental assumptions behind the SOLiD algorithm. As defined by Vlot (2019, p. 65), the *central platform* requires a full list of available vehicles to run the *"create list of most relevant vehicles" algorithm*. In Section 1.1, I mentioned that one of the main barriers to collaborative delivery between carriers is the lack of trust between competitors (Cruijssen et al., 2007; Daudi et al., 2016; Martin et al., 2018b). To implement the algorithm as is, all carriers would have to disclose the delivery zone and real-time position of their vehicles to a Central Platform. This is sensible commercial data, and thus it is unreasonable to assume that carriers will willingly share this information. I developed an alternative that does not rely on full information of the fleet, and I discussed it with the author of the original algorithm, Thymo Vlot. He verified that this approach does indeed cover from needing full information, and it also still works with the original intent of the algorithm. I explain my alternative in the following section.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

An alternative to Steps 2 and 3: Using a Depot Platform, Depot Digital Twins, and a Vehicle Filter

An alternative to this story is to shift the point of view to the vehicles. This alternative relies on three different agents: A Depot Platform, a Digital Twin of the Depot, and a Vehicle Filter. The Depot platform would be the medium between vehicles and depots, and the digital twin of a depot would be the medium between vehicles and the parcels within it. Finally, the Vehicle Filter would filter vehicles per Transport Request, thus performing a similar task to the Central Platform's in Step 2. This is an alternative offered by this thesis project and is not part of the SOLiD algorithm as it stands.

The BPMN diagram of this alternative would look as follows:

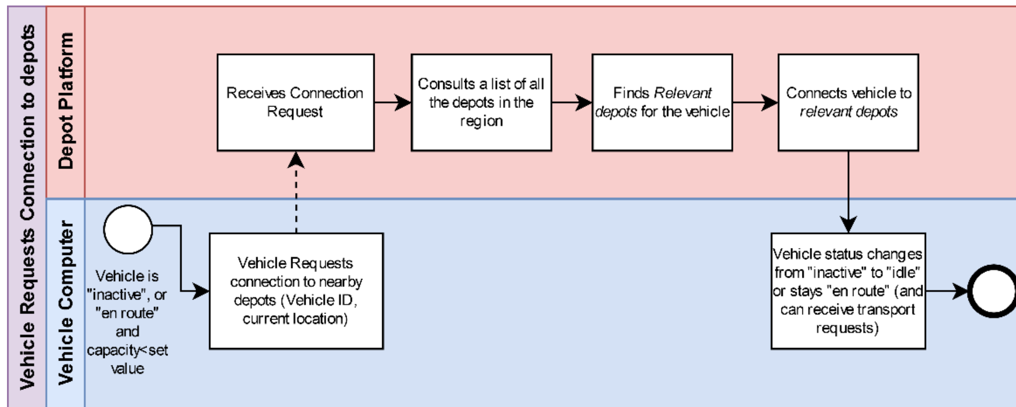

Figure 4-6. An alternative to steps 2 and 3: Vehicle Requests Connection to Depots

This process can start in two ways. First, this process can begin with the vehicle being "inactive", similar to the "Vehicle announces participation" process of Figure 4-5. Second, this process can also start if the vehicle status is "en route" and its capacity is under a set threshold. Then the vehicle creates a new type of request: a connection request. This connection request contains the vehicle's current location and the vehicle's ID. Then, the Depot Platform receives this request and consults a list of all the depots in a region. Thus, this Depot Platform only knows the location of the depots in a region. Then, this Depot platform finds relevant depots for the vehicle, using an algorithm that would be similar to the find relevant vehicles algorithm of Figure 4-7.

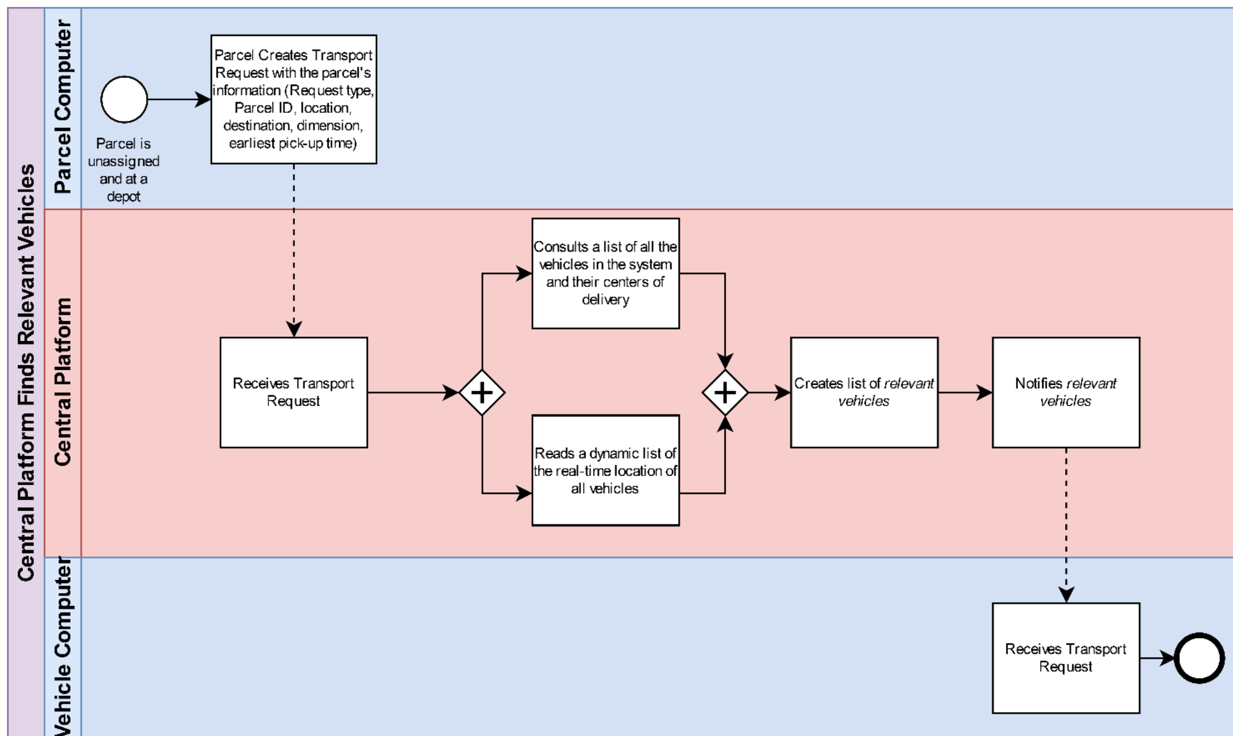

Figure 4-7. Central Platform Finds Relevant Vehicles

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

After finding *relevant depots* for the vehicle, the platform connects the vehicle to these depots. When this happens, the vehicle can receive transport requests from the parcels within these depots.

This process can be condensed in the *find relevant depots story*

Given a depot platform that knows the location of the depots

And can connect vehicles to the digital twins of these depots

And a vehicle

With a known real-time location

And an "inactive" status

Or an "en route" status and enough capacity to take in more parcels

And a depot digital twin that receives the requests of the parcels within

When the vehicle asks the platform to connect it to nearby depots

Then the platform finds "relevant depots" that are close to the vehicle (which could be done by a distance heuristic similar to the "Find relevant vehicles" *algorithm)*

And the platform connects the vehicles to the nearby depots

And the vehicle can receive the transport requests from the parcels within that depot

Then, an algorithm similar to the relevant vehicles algorithm of Step 2 would be implemented by a Relevant Vehicles Filter, with a slight simplification. Since vehicles that were close to the depot were connected by the depot platform, then this Relevant Vehicles Filter would not need to know the location of the vehicles, and thus only need to check their center of delivery and filter parcel requests to them. Figure 4-8 shows this process.

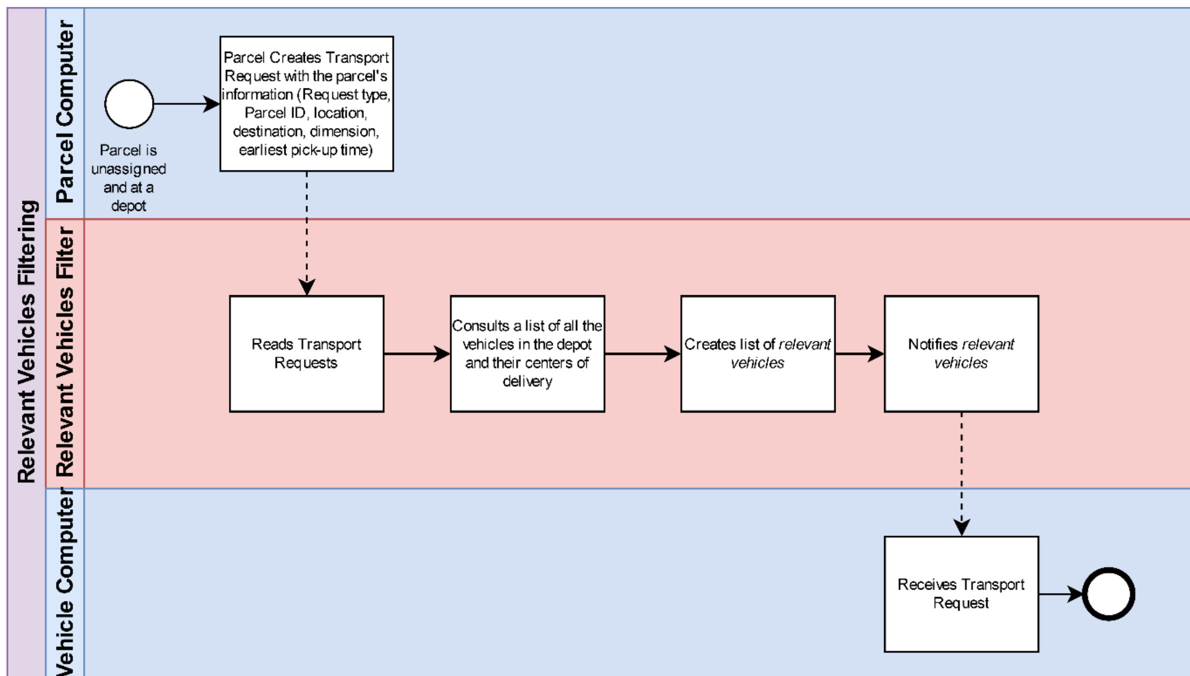

Figure 4-8. Alternative to Steps 2 and 3: Relevant Vehicles Filtering

Implementing this filter would significantly reduce the algorithm's computing time while avoiding the need to know the real-time locations of all vehicles. This alternative has a slightly different approach. Instead of having a central platform that knows where all vehicles are, there is a platform that knows where depots are. Then, each vehicle can contact this platform to be connected to nearby depots and receive the transport requests from the

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

parcels within that depot. To speed up the auctioning process, a vehicle filter can be implemented at the depot level.

In this alternative, there would be a need for a digital representation of each depot that would have access to the transport requests of the parcels within it. This is a more feasible scenario when considering collaborative delivery and avoids the necessity to know the position of all subscribed vehicles to begin auctions. The information of the centers of delivery is also spread out throughout depots and is not centralized into one agent, which is preferable to the Central Platform scenario.

The Filtering Relevant Vehicles *story* would be a simplified version of the original Find Relevant Vehicles *story*:

Given a depot

And a Relevant Vehicles Filter

And a "type_unassigned" or "type_assigned" transport request

And a list of all vehicles that connected to the depot with their delivery centers

Then the Relevant Vehicles Filter creates a list of the most relevant vehicles, filtering vehicles based on their center of delivery.

And a message is sent to the "most relevant vehicles" with the "type_unassigned" or "type_assigned" transport requests

The Relevant Vehicles Filter would have to read the centers of delivery of the vehicles connected to the depot and execute the relevant vehicles algorithm. Thus, the Relevant Vehicles Filter only needs computing power.

With this alternative, there is **no need for full information of a fleet**, or a central agent that knows where all parcels or vehicles are. There would be a need for a different platform that only knows the location of the depots, and helps the vehicles connect to them, which would require less sensitive information from Carriers. In practice, this alternative allows the creation of Depot Digital Twins for the Carriers. This has impactful implications because it is possible for this system to link the first mile of delivery with the last mile of delivery. The way it would work is the following:

1. A depot digital twin is created for the shipper's warehouse
2. The parcels within that depot come directly from the shipper's e-commerce platform. Thus digital twins for these parcels would be created at the moment of purchase.
3. Then the parcels would request transport
4. Nearby vehicles from associated Carriers would connect to the depot and be able to participate in auctions to deliver the parcel.

Prime Vision's customer requirements stressed the importance of housing multiple carriers under one platform and breaching the gap between the first and last mile of delivery. Therefore, this alternative better fulfills Prime Vision's objectives and customer requirements. From this point onwards, I will include the depot digital twins, the depot platform, and the vehicle filter as actors in the remaining BPMN diagrams.

Step 4: Vehicle creates bid

Both the Central Platform approach and the Depot platform and Depot Digital Twins approach are the medium through which a vehicle can receive transport requests. The next step in the delivery process is the response of the vehicles to these requests, which is independent of the approach taken, be it the original Central Platform approach or the Depots Platform approach I suggest. The process is shown in Figure 4-9:

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

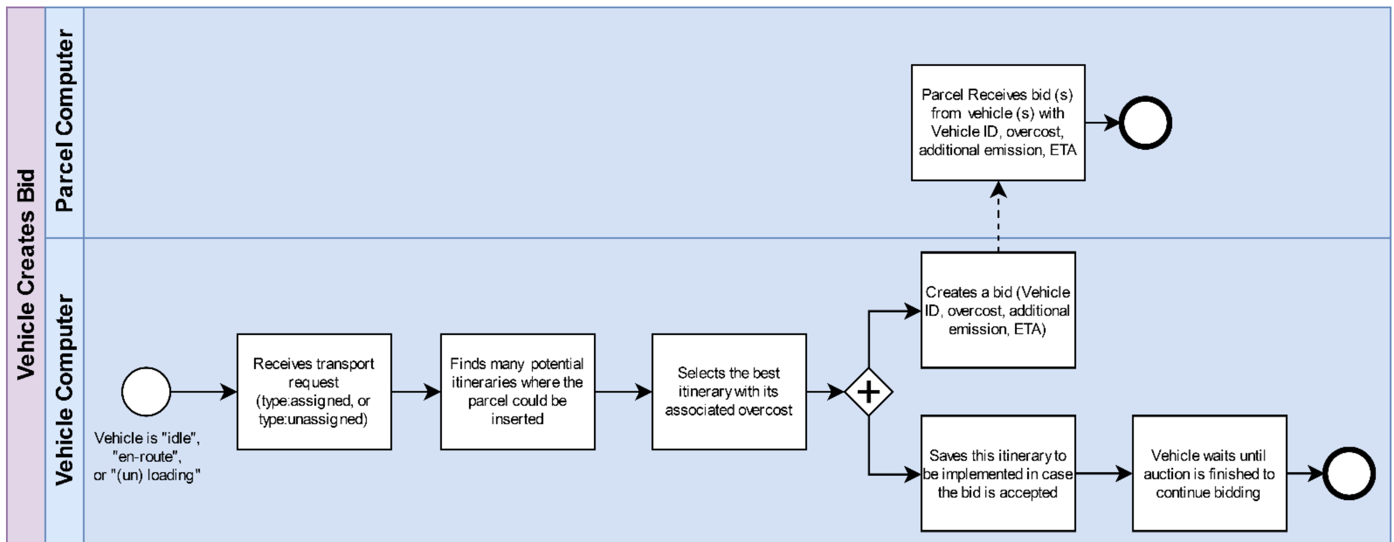

Figure 4-9. Step 4: Vehicle Creates Bid

The vehicle can receive transport requests in the "idle", "en route", "loading" and "unloading" statuses. When this happens, the vehicle creates many potential itineraries that include the parcel, and then select the best one. Each of these itineraries has an associated *overcost* - the price of the transport offer. After selecting the best offer, it creates a bid that has the vehicle's relevant information for the parcel, namely the vehicle ID, the trip's *overcost,* the additional emissions if the vehicle were to deliver the parcel, and the Estimated time of arrival (ETA). The itinerary associated with this bid is stored in case the bid is accepted. In this case, the vehicle would adopt this itinerary and pick up the parcel.

This leads to the *Vehicle Creates Bid story*, (Modified from Vlot's pseudocode for the *Insertion* Heuristic (Vlot, 2019, p. 52)):

> *Given a vehicle*
>
> *With an ongoing itinerary, in the "idle" status (waiting for parcels to transport) or an "en-route" status (delivering parcels)*
>
> *And a known real-time location*
>
> *And a parcel transport request (with delivery time limit and dimensions)*
>
> > *If the time windows are coherent (pick up time is before the expected arrival time of the vehicle, and the delivery time limit is not violated)*
> >
> > > *Calculate overcost of pickup and delivery (price for parcel) (many potential itineraries are generated)*
> > >
> > > *And Store all overcosts in the "insertions list" with the vehicle ID*
> > >
> > > *And every "insertion" is paired to an itinerary (route)*
> >
> > *Then evaluate all insertions from the "insertions list" against the capacity of the vehicle at the moment of pickup*
> >
> > *And select the best insertion (with its paired route).*
> >
> > *And save this route (to that it can be implemented if the bid is accepted)*
> >
> > *And create the bid with the best offer.*
> >
> > *And save this bid's overcost as a quote for later use*
> >
> > *And send the bid to the parcel.*

In this *story*, the vehicle has many opportunities to fit a parcel in its itinerary, and thus many potential *insertions* are created. These are then listed and the best one is sent as an offer to the parcel. It might seem intuitive to calculate capacity first and then calculate an offer, but the issue is that the vehicle's capacity is a dynamic value that changes

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

during the day as the vehicle picks up and drops off parcels. Therefore, capacity must be checked at all possible "insertion points", which leads to the final bid that reaches the parcel.

This story hints that vehicles, just like parcels, must hold some of its key information somewhere. This would be its center of delivery, its capacity, the start and end of its delivery window, and its status. To comply with the Data Transparency requirement from Carriers, it would be important to log this data as well. To create its bids, vehicles would have to compute the "*insertion heuristic",* and thus like parcels and depots, vehicles would need processing power. It also shows that the vehicles need a medium to both to receive transport requests from the parcels and send their offers to them. Also, the route that is created with the bid must be stored temporarily, so that it can be implemented if the parcel accepts the bid. Thus, this route needs to be stored somewhere. Lastly, each bid's *overcost* represents its quote as well, which is what the parcel would pay if that vehicle does transport it. This information needs to be saved for later use as well, be it by the vehicle or the parcel.

Step 5: Parcel selects the best vehicle

The next step in the delivery process is the parcel's evaluation of the received bids, seen in Figure 4-10:

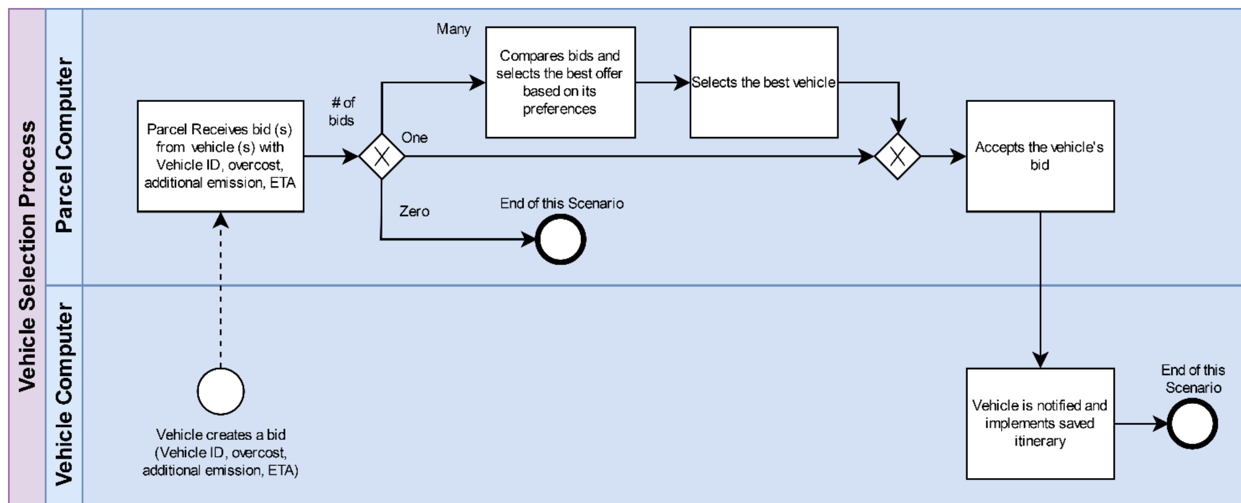

Figure 4-10. Step 5: Vehicle Selection Process

The Vehicle Selection Process begins with the bids created by the vehicles. Depending on the number of bids (zero, one, or many) the parcel executes different activities. If the number of bids is zero, the process stops. If there is only one bid, the parcel automatically accepts it, and if there are more than one bids, the parcel evaluates all bids and selects the best offer based on its preferences. Let us remember that each parcel has a set of preferences for delivery based on cost, environmental emissions, and speed of delivery. After accepting a bid, the vehicle is notified, and the vehicle implements the saved itinerary that was created in the *Vehicle Creates Bid* process, and thus begins to drive to pick up the parcel.

The *Vehicle Selection story* would be as follows:

Given a parcel

And the parcel has a set of preferences

And a set of bids from one or multiple vehicles

And a bid by a vehicle (bid status is pending)

Then the parcel evaluates bids based on its preferences and accepts one bid

And the parcel must have enough credit to accept the bid

And a message is generated for that vehicle

And the best bid is accepted (bid status changes to accepted)

Then the accepted vehicle implements the itinerary that includes the parcel

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

And a message is generated for all rejected vehicles (bid status changes to rejected)

And the vehicle continues to participate in future auctions

From this story, I deduce that each parcel needs processing power for both creating requests and accepting bids. Just like in the "create transport request" story, this story also relies on communication with other agents in the system.

Step 6: Parcel requests transport when already assigned

The SOLiD algorithm allows parcels to find even better means of transport, even when the parcel already has an assigned vehicle, visible in Figure 4-11.

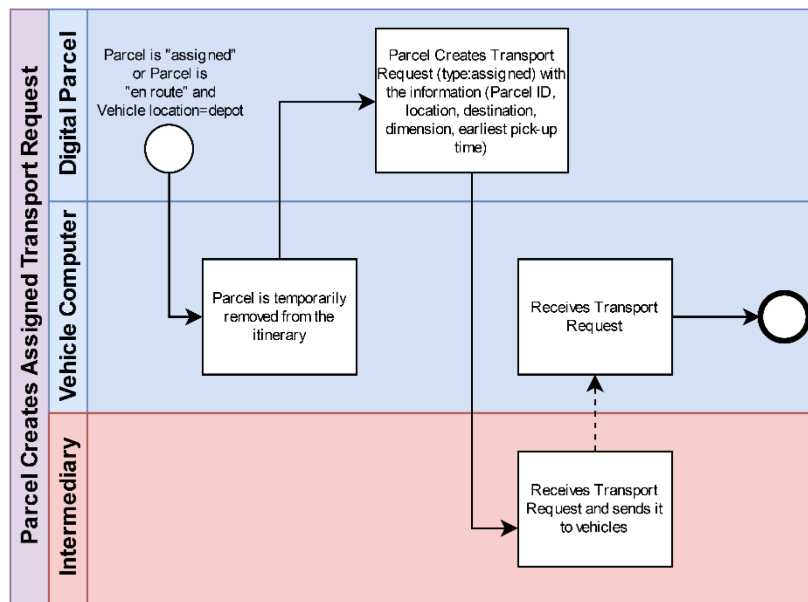

Figure 4-11. Step 6: Parcel Creates Assigned Transport Request

This process begins when the parcel is "assigned". The first step for creating an assigned transport request is temporarily removing the parcel from the vehicle's itinerary. Then, the parcel creates a new transport request, an *assigned* transport request. This request type means that the parcel will have less priority for vehicles. Then, an intermediary actor ensures that this request reaches vehicles. This process is repeated a limited number of times after the parcel was assigned to a vehicle. In the original SOLiD algorithm, this intermediary would be the Central Platform; in the alternative I suggested, this intermediary would be the **digital twin of the depot** where the parcel is. In the case of a transfer, which is when the parcel status is "en route" and the vehicle is at a depot, the parcel should have a way of communicating its transport requests to the other vehicles. Thus, the parcel must also have a way of connecting to the digital twin of the depot, where other vehicles can receive its requests.

From this process I develop the *Create Assigned Transport Request story:*

Given a parcel

And the parcel status is assigned

And a set of parcel preferences

And a delivery destination

And a set amount of credit available for the parcel

And a medium to send transport requests to vehicles

Then the parcel creates a "type_assigned" transport request

And the transport request is sent to an intermediary

And vehicles receive the transport requests through the intermediary

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

This process is the same as the *create transport request* story, with an added step of being temporarily eliminated from the itinerary of the current vehicle. This *story* points towards the parcel capabilities that were previously mentioned: communication, data storage, and computing power for creating the transport requests. Also, the vehicle that temporarily removes the parcel from its itinerary needs to store this itinerary to be implemented if the parcel does not accept another bid.

Step 7: Parcel loading process

After the parcel has accepted a transport request, the parcel must be loaded onto the correct vehicle. For that to happen, the incumbent vehicle needs to know where the parcel is located within a depot, and the parcel needs to be moved to a loading dock where it can be picked up by its assigned vehicle. To do this, the parcel requests transport to a loading dock. This process works under the assumption of Depots that have digital twins, as described in the alternative to steps 2 and 3. Thus, this process is not yet in the SOLiD algorithm. The detailed process would be as follows:

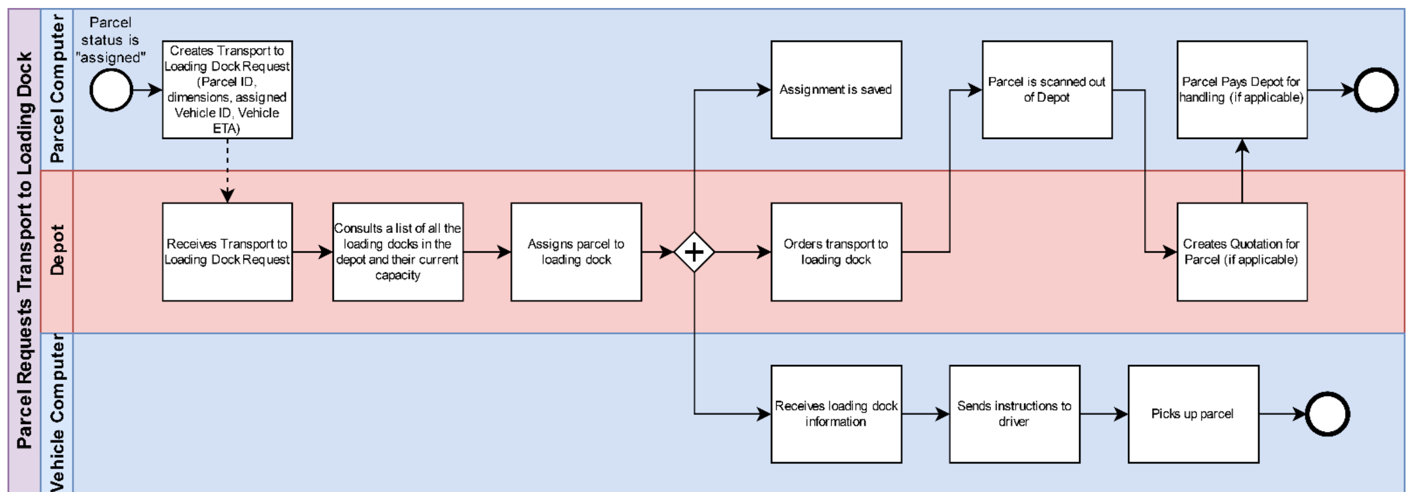

Figure 4-12. (added) Step 7: Parcel Requests Transport to Loading Dock

In this process, when the parcel becomes "assigned" to a vehicle, it requests transport to a loading dock within its current depot. The depot then receives this request and checks the current capacity of its loading docks. After finding an available spot for the parcel, it assigns the parcel to a loading dock, and it orders the transportation of the parcel to this loading dock. The depot also notifies the incumbent vehicle of the exact pick-up location. After a parcel is transported to its correct loading dock, it is scanned out of the Depot. At this moment, the depot creates a quote for the parcel, depending on the time it spent on the depot and other handling costs, which the parcel pays from its credit.

This same procedure for requesting the assignment to a loading dock could also be followed for dropping off a parcel at a depot. As I describe in Step 9, this is necessary for the parcel to transfer from one vehicle to another.

The corresponding "Parcel Requests Transport to Loading Dock" *story* would be:

Given a Depot with and its Depot Digital Twin

And the depot has a set of loading docks with its correspondent capacities

And a vehicle with known ID and ETA to the parcel

And a parcel

With given dimensions

And the parcel is assigned to a vehicle

And a set amount of credit available for the parcel

And a medium to communicate with the depot

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

And the parcel is in a depot

Then the parcel creates a "transport to loading dock" request

And the transport request reaches the Depot Digital twin

And the Depot Digital Twin the capacity of its loading docks

And assigns the parcel to one of them

And orders transport of the parcel to the assigned loading dock

And informs the assigned vehicle of the loading dock where the parcel can be picked up

And the driver receives pick up instructions (if needed)

And the parcel is picked up

And the parcel is scanned out of the depot

And the Depot Digital Twin creates a quotation for the parcel for handling costs

And the parcel pays the Depot Digital Twin

This *story* shows that there are many activities during the stay of a parcel in a depot and actors require capacities to perform them. Specifically, the parcel needs to be able to communicate with the Depot Digital Twin and request transport from it. Additionally, the parcel and the depot digital twin must have a way of knowing the ID of the vehicle that will pick up the parcel. The parcel, for example, could communicate this in the "Transport to loading dock" request. Furthermore, the depot digital twin should be able to give instructions and detailed pick-up information to the vehicle. Again, the parcel must have a way of paying for handling, and in this case, the depot must also be able to receive such payments. Finally, the depot digital twin needs computing capabilities to create quotations, assign parcels to loading docks, and order their transport to the assigned loading dock.

Step 8: Parcel pick-up process

When the parcel is picked up at a depot, two important updates must be made, as shown by Figure 4-13:

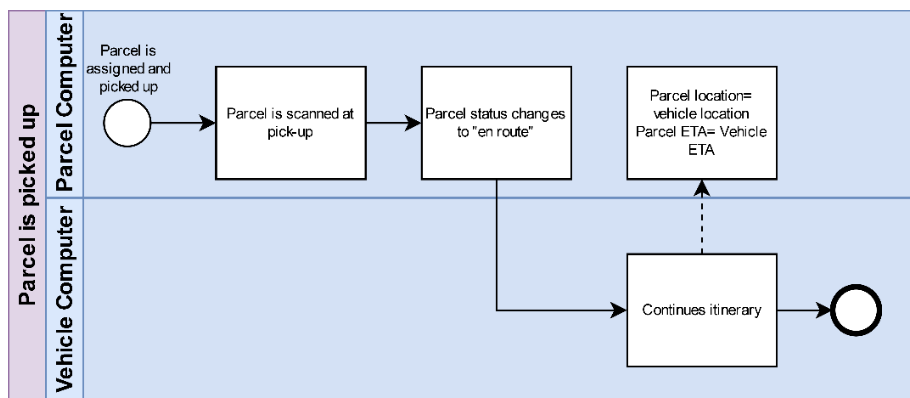

Figure 4-13. Step 8: Parcel is Picked up

This straightforward process shows that the parcel status only changes from "assigned" to "en route" when it is scanned into the vehicle. This would be done by the driver's handheld device. Furthermore, when the parcel is scanned, its location and Estimated Time of Arrival (ETA) can be read as that of the vehicle.

The "Parcel Pick-up" story would be as follows:

Given a vehicle

And a parcel

And the parcel is assigned to the vehicle

And the parcel was picked up at a depot by the vehicle

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Then the parcel is scanned into the vehicle

And the parcel status changes to "en route"

And the parcel's location becomes the vehicle's location

And the vehicle continues its itinerary

From this *story* follows that the vehicle must be able to track its own location and that the parcel's location and Estimated Time of Arrival can be loaded from this location. If this location is logged, the system could easily comply with the Data Transparency constraint set by the Carriers (see Section 3.2.3). Importantly, if the parcel's location is the same as the vehicle it is assigned to, there is no need to track the physical location of the parcel, simplifying asset tracking tremendously.

Step 9: Transfer Evaluation at Depots

After pick-up, the parcel is finally "en route" towards its destination. If a vehicle goes through a depot, however, parcels in the SOLiD algorithm evaluate potential vehicle transfers. This becomes a different type of assigned transport request that involves more actors and thus is more complex. In the BPMN diagram for this process, I color-coded the activities that are not part of the original SOLiD algorithm. Figure 4-14 shows the complete process.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

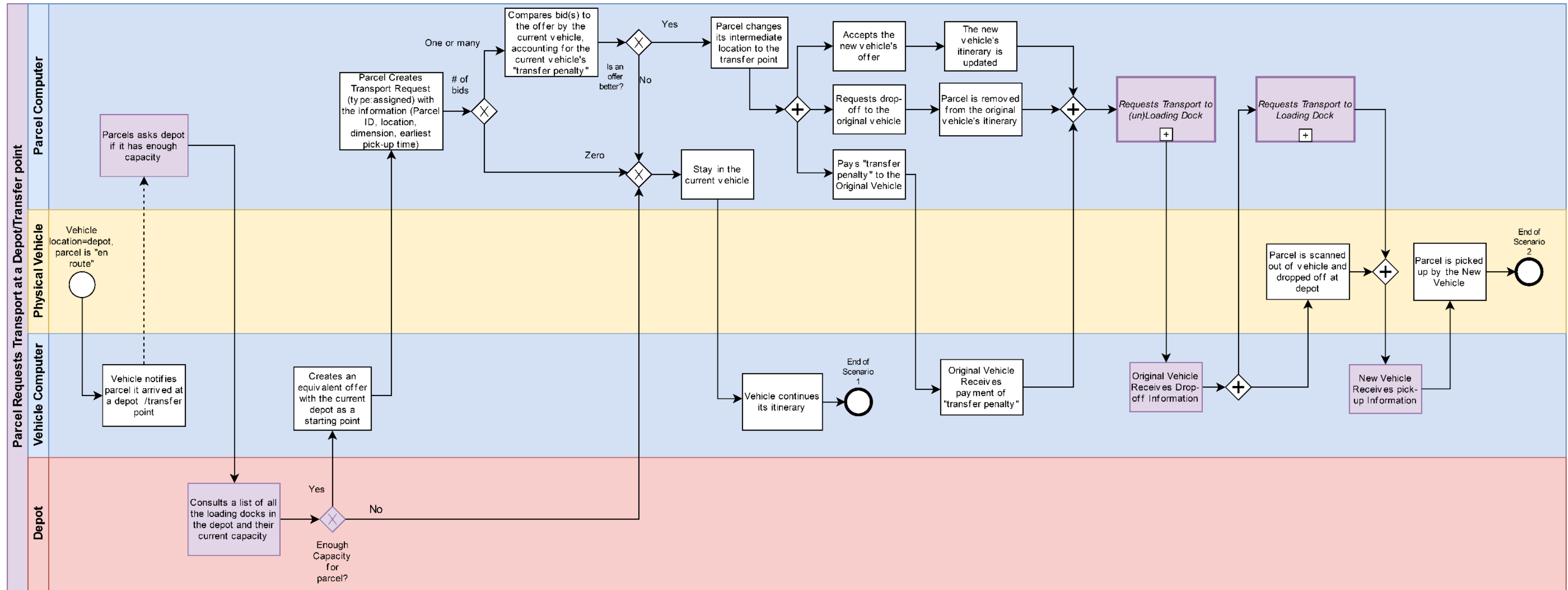

Figure 4-14. Step 9: En route Transport Request

In essence, the "en route" transport request happens when a vehicle passes through a depot and the parcel evaluates new vehicles that might suit its preferences better. The process begins when the vehicle notifies the parcel that it is at a depot. The parcel then asks the depot if there is enough capacity for the parcel, and if the answer is yes, the parcel will evaluate a transfer. To do so, the parcel asks its current vehicle for an adjusted offer of its current trip. This adjusted offer is a bid that counts as if the parcel had begun its travel with the original vehicle at that particular depot. The parcel then creates an assigned transport request and compares the received bids to this adjusted offer of the original vehicle. If there is a better offer by another vehicle, the parcel changes its position from the original vehicle's position to the depot's position, so that the new vehicle knows where to pick it up. Then, the parcel does three things: First, the parcel accepts the new vehicle's offer, and thus that vehicle adopts the itinerary with the parcel and begins traveling towards the depot where the parcel will be dropped off. Second, the parcel requests drop-off to its current vehicle, which triggers its removal from the original vehicle's itinerary. Finally, the parcel pays a transfer penalty to its original vehicle.

After all these events, the parcel creates a "Transport to (un)Loading Dock" Request. This is the same process I described in Step 7 but used for assigning the parcel to a loading dock where it can be dropped off. Then, the original vehicle is notified of this drop-off location and the parcel is dropped off. Then, the parcel repeats the same procedure, this time to be assigned to a Loading Dock to be picked up by the second vehicle. This could be the same Loading Dock where it was dropped off, but it does not have to be. Finally, the parcel is picked up by the new vehicle.

The "En route Transport Request" *story* would be as follows:

Given a depot and its depot digital twin

Given an original vehicle

And the Original Vehicle is at the depot

And a set of new Vehicles

And the New Vehicles are connected to the depot digital twin

And a parcel

And the parcel is assigned to the original vehicle

Then the parcel asks the depot digital twin if there is capacity for the parcel

If there is capacity

Then the parcel asks the Original Vehicle to create an equivalent bid with the depot as the starting point

And the parcel Creates an assigned Transport Request

If there are one or many bids

Then the parcel compares the bid(s) to the equivalent bid by the original depot plus a transfer penalty

If there is a better offer

Then the parcel Changes its position to the depot's position

And accepts the New Vehicle's bid

Then the new vehicle's itinerary is updated

And the parcel requests the Original Vehicle to drop it off at the depot

Then the parcel is removed from the Original Vehicle's itinerary

And the parcel pays the Original Vehicle for the transfer penalty

Then the Parcel requests transport to an unloading dock to the depot digital twin

And the Depot Digital Twin assigns the parcel to a drop-off location in the depot

Then the Original Vehicle receives the Drop-off Information

And the Original Vehicle scans the parcel at the drop off point at the depot (and the parcel is dropped off)

And the Parcel Requests Transport to a Loading Dock

Then the Depot digital twin assigns the parcel to a loading dock and transports it to said loading dock

Then the New Vehicle receives the pick-up information

And the parcel is picked up by the new vehicle

If there are no bids

Then the parcel stays in the current vehicle

And the vehicle continues its itinerary

If there is no capacity for the parcel at the depot

Then the parcel stays in the current vehicle

And the vehicle continues its itinerary

The most important lesson from this story is the orchestrated nature of the algorithm. When the actors interact, it is specific actions that trigger the next sequence of events. If that sequence is misconstrued and the events are triggered in the wrong order, then the algorithm will not work, and the software will not do its job. Therefore, there must be special attention to the sequencing of events when designing the software architecture of the system. For functional requirements, all agents must have communication capacities, and that both the original vehicle and the new vehicles must be connected to the depot digital twin. Thus, the "Vehicle requests connection to depots" process might have to be revised to ensure all vehicles are connected to the relevant depots where parcels can evaluate transfers.

The capabilities of the depot are consistent with those seen in the "Parcel Requests Transport to Loading Dock" story: the depot can check its own capacity, assign parcels to loading docks, order their transportation, and receive payment for handling. Furthermore, this procedure applies to both pick-up and drop-off.

Step 10: Parcel Drop-off

Parcels are dropped off when they reach their destination. The original SOLiD algorithm by Vlot assumed that all customers were always available to receive their parcels (Vlot, 2019, p. 48). In the BPMN diagram of this process, I included the other activities that would be necessary if this assumption were not made, color-coded in purple. Figure 4-15 shows the complete process:

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

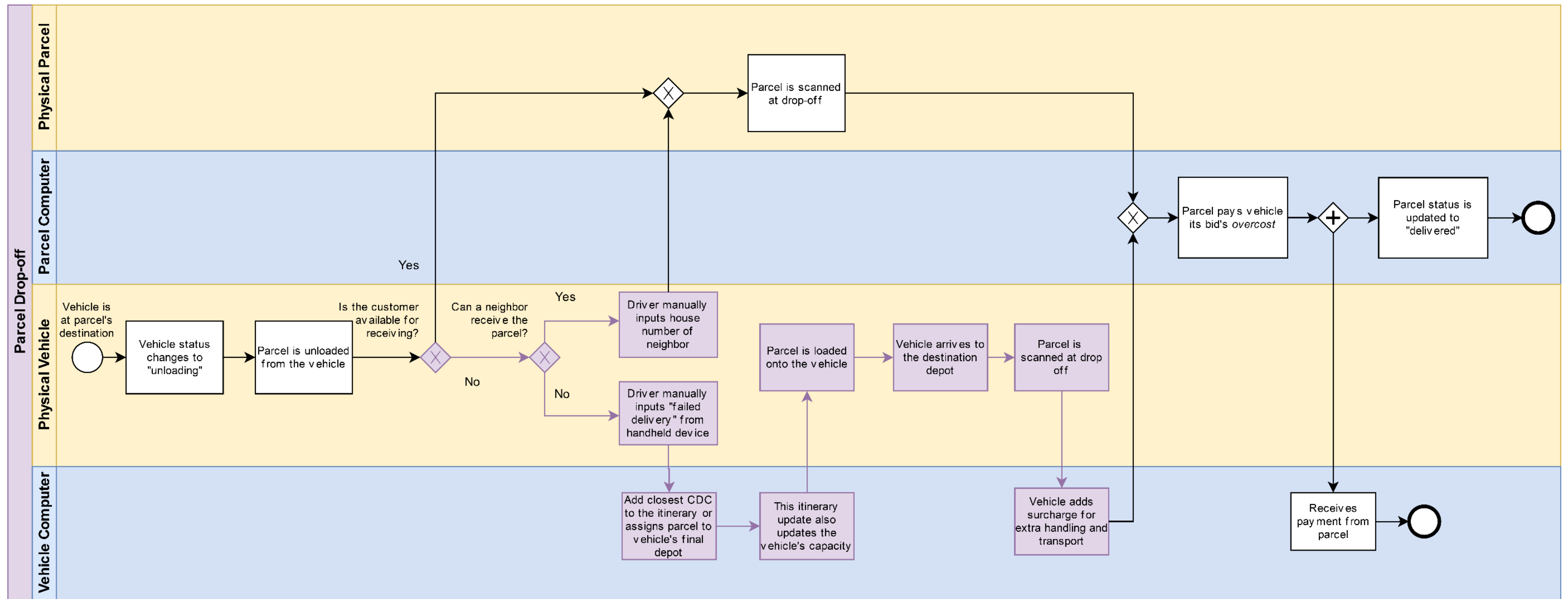

Figure 4-15. Step 10: Parcel Drop-off

In summary, the delivery driver will attempt to deliver the parcel to the customer, but if the customer is not available, the driver will first try to give the parcel to a neighbor. When successful, the driver adds the neighbor's information on his handheld. If unsuccessful, the driver must take back the parcel and transport it to a depot. In this last case, the parcel must pay a surcharge for the added handling.

The "Parcel drop-off" *story* would be as follows:

Given a vehicle

And a parcel

And the parcel is assigned to the vehicle

And the vehicle is at the parcel's destination

Then the vehicle's status changes to "unloading"

And the parcel is unloaded from the vehicle

If the customer is available for receiving the parcel

Then the parcel is scanned at drop off

And pays the vehicle's overcost

And the vehicle receives the payment by the parcel

And the parcel status is changed to "delivered".

If the customer is not available for receiving the parcel

Then the driver will check if a neighbor can receive the parcel

If there is an available neighbor

The driver manually inputs the house number of the neighbor on his handheld

And the parcel is scanned at drop off

And pays the vehicle's overcost

And the vehicle receives the payment by the parcel

And the parcel status is changed to "delivered".

If there is no available neighbor

The driver manually inputs "failed delivery" on his handheld

And the vehicle adds the closest Collection and Delivery Point to the itinerary

And the parcel is loaded back into the vehicle and is delivered to its new destination

And the delivery driver scans the parcel at drop-off

And the vehicle adds a surcharge for extra handling to the parcel

And pays the vehicle's overcost

And the vehicle receives the payment by the parcel

And the parcel status is changed to "delivered".

This story highlights that the handheld device for the delivery drivers must have an effortless way of inputting the case for failed deliveries and drop-off at a neighbor. Furthermore, the vehicle adds a surcharge for extra handling if the delivery fails, and thus should be able to compute how much this surcharge should be.

After going through all the stories, I summarized the key capabilities and functional requirements per entity d in Table 3-1. I show all the entities discussed in this chapter, including the Central Platform of the original SOLiD algorithm.

Table 4-1. Functional Requirements of the SOLiD algorithm.

Capacity/Entity	***Parcels***	***Vehicles***	***Depot Platform***	***Depot Digital Twin***	***Vehicle Filter***	***Central Platform***
Compute Algorithms	***X***	***X***	***X***	***X***	***X***	***X***
Store Information	***X***	***X***	***X***	***X***		***X***
Communicate	***X***	***X***	***X***	***X***		***X***
Track its Position	***X***	***X***				
Make Payments	***X***					
Receive Payments		***X***		***X***		
Log Events	***X***	***X***		***X***		
Check its capacity		***X***		***X***		
Connect Entities			***X***			
Know the location of all depots		***X***	***X***			
Know the Center of Delivery of vehicles					***X***	***X***
Orchestrated activities between entities	***X***	***X***	***X***	***X***	***X***	***X***

All entities must be able to compute algorithms, store information, and communicate. Vehicles and parcels need to be able to track their position, the vehicles use a Global Positioning System that aids this task. For the parcel, if they are scanned into a vehicle, the parcels could load their location without having to track the individual parcel's location. This would simplify tracking parcels tremendously. As many processes involve transactions, vehicles and depots need to be able to receive payments done by the parcels. The parcels, vehicles, and depot digital twins need to be able to log their significant events for later tracking of KPIs, as specified in section 3.2.3, and this function is useful for many other uses, especially recovery from crashes, and security. Depot and Vehicle digital twins need to be aware of their own capacity to admit parcels. The special Depot Platform that I suggested would need to compare the distance between a requesting vehicle and the known depots in a region so that it can connect the requesting vehicle to them. Thus, it needs to know the depot locations and computing power. The vehicles need to know the location of depots because they need to let the parcels know that they could potentially transfer when it goes through a depot.

As I mentioned in Step 2: Central Platform Finds Relevant Vehicles, the Central platform would need to know the center of delivery and center of delivery of all vehicles to list relevant vehicles, which would be troublesome given the lack of trust between competitors. In this system, vehicles also need to go through Step 3: Vehicle announces participation in the system to be considered by the Central Platform in the first place. The alternative I proposed has vehicles ask a Depot Platform for connection to Depot Digital Twins, and a local Vehicle Filter that speeds up auctions in depots.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

4.3. *Summary*

In this chapter, I described the SOLiD algorithm using granular BPMN diagrams. I did not limit to explaining the SOLiD algorithm as it stands but extended it with the activities of the complete last-mile delivery process. I discussed that the current version of the algorithm uses a central platform that knows where all vehicles are. As this is averse to the interests of carriers in a collaborative delivery scenario, I offer an alternative solution that uses a Depot Platform, Depot Digital Twins, and a Vehicle Filter. These three entities replace the central platform and do not require full information of the fleets. Furthermore, this alternative can hypothetically link the first and last mile of delivery, a powerful implication for the logistics sector, improving the potential commercial value of the Logistics Planning System. I also describe the hypothetical activities of these added entities with BPMN diagrams.

From each BPMN diagram, I developed *stories* that synthesize the crucial scenarios of the system. Stories clarify the context and the executor of the actions, which will be useful for decomposing the system in the next chapter. From these stories, I extracted key functional requirements for the application of the algorithm. These functional requirements are found in Table 4-1, answering the research question for this chapter.

This ends Phase 1 of Dym et al.'s Systems Engineering Design methodology, *Problem Definition.* Since customer and company requirements are now set, it is possible to continue onto Phase 2, which begins the *Conceptual design* of the SOLiD-based planning system's software architecture.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

5. Conceptual Design

The first half of this chapter concerns the beginning of the conceptual design of the planning system. The first topic I discuss in Section 5.1 is the selection of an architectural *style*, choosing between the *monolithic style*, the Service-Orientated Architectural *style*, and the Microservices *style*. Given that I found the desired Quality Attributes in Section 3.2 and the functional requirements in Section 4.2, I now have a clear criterion to choose one style over the others.

By selecting an architectural style, I answer Research sub-question 5:

5. *What is the most appropriate architectural style for a self-organizing logistics planning system?*

After selecting one architectural style, I continue to answer research sub-question 6:

6. *How to decompose a self-organizing logistics planning system?*

I answer this question by defining the software architecture in Section 5.2, which begins by constructing a conceptual view of the system in Section 0, called a *high-level domain model of the system*. To do this, I revisit the user *stories* developed in Section 4.2. I present two versions of the system: one of them has the Central Platform that Vlot suggested (2019), and the other has the Depot platform and Depot Digital twins that I suggested in Section 4.2. This comparison permits exploring the extent to which the alternative system would be more complex than the original. Furthermore, the exploration of the original system allows for an unadulterated vision of Vlot's system, something interesting from a research point of view. Then, I will expand these two sketches into a *class diagram of the system,* which is a more detailed conceptual view. This class diagram lets us define the system's operations, which are necessary to apply a decomposition *pattern*. It is important to decompose the system in services to achieve the ultimate goal of designing the software architecture of the Self-Organizing Logistics Planning System. I introduce and select the decomposition patterns in Section 5.2.2. I discuss two decomposition *patterns*: Decomposition by Business Capability and Decomposition by Sub-Domains. This section ends with the application of the former as the most suitable *pattern* for decomposition. Then, in Section 5.2.2, I apply the Decomposition by Business Capability *pattern*, which results in a segmentation of the planning system into a set of services with defined responsibilities. Finally, in Section 5.2.3, I specify the APIs or interactions of each of these services, which is done by formally assigning operations to each service.

By decomposing the system in services and assigning tasks to each of them, I answer research sub-question 6. This encapsulates the first half of Phase 2 of Dym et al.'s Systems Engineering Design methodology, *Conceptual Design* (2014). The Research Flow Diagram of Figure 5-1 summarizes the research questions, activities, and deliverables for this Chapter.

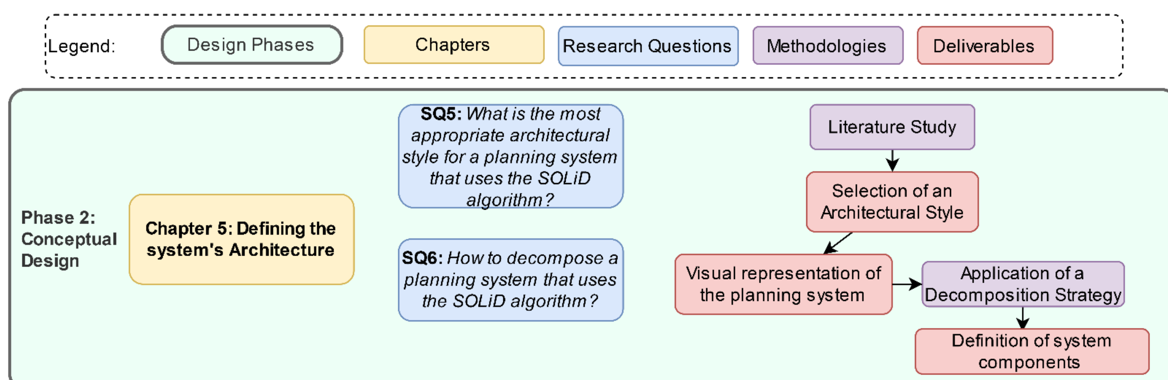

Figure 5-1. Phase 2, Conceptual Design, adapted from Systems Engineering Design methodology by Dym et al. (2014)

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

5.1. Choosing an architectural *style*

The main architectural *styles* I introduced in Section 0 are the *monolithic, service-oriented,* and *microservice* architectural *styles*. Each of them has different advantages and disadvantages. Architectural *styles* help achieve system requirements, and thus the most appropriate *style* must be picked to suit business needs. Carriers valued availability, data transparency, and security the highest, as summarized in Table 3-3. The Prime Vision team, in turn, considers Scalability, Performance, Modularity, Modifiability, Testability, and Deployability the most important quality attributes for a hypothetical software system, as compiled in Table 3-5. These requirements help prioritize architectural *styles.*

In Chapter 4, I also explored the functional requirements of the system. Table 4-1 summarized key elements of the system: First, the communication between entities and the orchestration of their activities is extremely important. Second, the system requires decentralized computing capabilities. Third, some positioning systems will be necessary for vehicles. Fourth, most actors need storage capabilities. Finally, parcels need to be able to make payments, and vehicles and depots need to be able to receive these payments.

I will explore the *monolithic, service-oriented,* and *microservice* architectural *styles* with these Quality Attributes and functional requirements in mind.

5.1.1. *Considering the monolithic Architecture*

Section 2.7.1 discussed that the *monolithic style* was good for simple applications that need a single executable file that encompasses all functions. However, scalability is greatly compromised in a *monolithic* application. The reason is that different types of components scale in different ways. For example, databases scale with memory, and image processing scale with CPU capacity. If all the functionality of an application is contained in a single executable file, it is impossible to cater to the needs of each component, hence the scalability ceiling for *monolithic applications*. Furthermore, testability, modifiability, and deployability are also impaired because a large application quickly becomes complex, worsening a development team's ability to understand and significantly improve its source code.

The characteristics of this architectural style are incompatible with the agile working philosophy at Prime Vision, and I conclude that a monolithic structure would ultimately not fulfill the system requirements. Furthermore, considering the decentralized nature of the system, with smart parcels and vehicles, it makes little sense to recommend a *monolithic* structure to house these digital entities. The *monolithic style* is thus discarded.

5.1.2. *Comparing the Service-Oriented Architecture (SOA) and the Microservices Architecture*

The next alternative to *monolithic* applications is the SOA *style*. As discussed in Section 2.7.2, SOA separates functionality in coarse-grained services connected by an Enterprise Service Bus (ESB) that standardizes communication between services and adds computational power to communication (Bass et al., 2003; Richardson, 2019). This results in communication protocols that are considered *heavyweight*, as is the staple SOAP in SOA architectures. These protocols orchestrate, secure, and log the execution of services in series, which is good for failure recovery but add significant overhead, affecting the system's performance (Bass et al., 2003). We would see that SOA could offer high availability and consistency among service, at the expense of speed. SOA is also characterized by the use of joint databases between services. This has three important limitations. Firstly, different services and types of functions make better use of certain types of databases, so a shared database with a global data model forces a one-size-fits-all scenario that compromises performance. Secondly, shared databases hinder the isolation of services, which affects its *modularity* and *modifiability* because changes in a service could compromise functionality in another, as their databases are shared. Lastly, in the collaborative delivery scenario discussed in Section 1.1, data completeness was seen as a major obstacle to collaborative delivery, as Carriers are not willing to share the full information of their fleets (Cruijssen et al., 2007; Daudi et al., 2016; N. Martin et al., 2018b). The SOLiD algorithm, implemented in the SOA *style*, would probably utilize a shared database that contains the data of all vehicles and parcels, something that Carriers would probably never agree to disclose.

The *microservices* architecture has many things in common with the SOA *style*: both are suited for decentralized systems and separate functionality in different services that communicate with one another. One of the key differences is the scale, as the SOA architecture often connects larger *monolithic* applications, while the *microservices*

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

architecture separates a system into finely grained services. Notably, these *microservices* have their own database, and thus avoid the drawbacks of the shared databases present in the SOA architecture, as discussed in Section 5.1.2. This gives *microservices* an edge over the SOA architecture in terms of performance, modularity, and modifiability, which were key Quality Attributes for the Prime Vision team. In the context of the SOLiD algorithm, the *microservices* architecture would theoretically allow databases to stay isolated, thus aiding the exploration of collaborative delivery. Housing different Carriers under one platform was one of the main functional requirements for Prime Vision, and the finely grained nature of *microservices* also suit Prime Vision's agile work philosophy.

As discussed in Section 3.2.3, I will need to take special considerations concerning security and data transparency. I will tackle this when considering *patterns* within the *microservices style* in Section 6.1.3.

We can see another comparison between SOA and microservices in Figure 5-2:

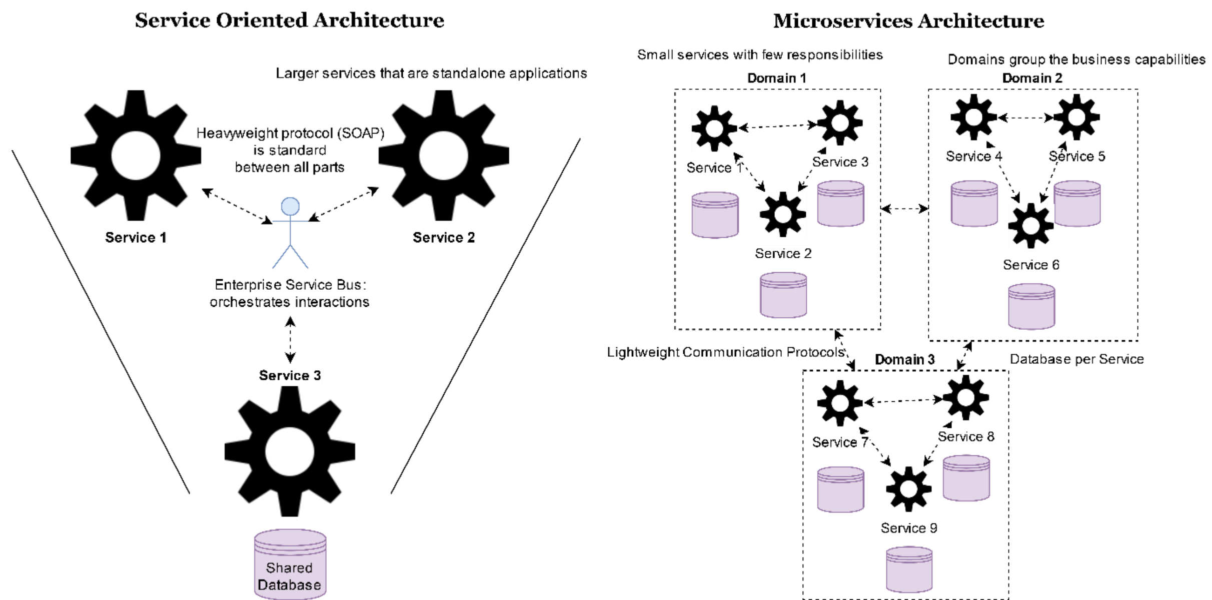

Figure 5-2. Comparison between the Service-oriented architecture and the Microservices architecture

As we can see in Figure 5-2, SOA utilizes bigger services that are basically standalone applications in their own right, and communicate with each other using standard heavyweight protocols. This protocol is usually SOAP, and it functions through an Enterprise service bus, which orchestrates interactions between services. The one thing that keeps SOA services from being standalone applications is that they share a database. Having a shared database eases having consistent data across services because the interactions of all services update the same database, which is easily accessible. This is what is considered ACID transactions. These benefits come at the cost of a single database that has to fit all services, and as discussed in Section 5.1.2, this hinders the application's scalability and performance. It also makes development more complicated, because all the teams have access to the same database. For example, if a development team for Service 1 changes something in the shared database, Services 2 and 3 could be compromised and might need to be updated to cope with the database changes. Thus, teams must convene with each other often, and the development process must be highly coordinated.

In the Microservices architecture, the services are much smaller, and they have fewer responsibilities. The communication protocols are lightweight, and orchestration between services relies solely on messaging. Thus, the performance is higher. In the microservices architecture, decomposition is done across domains, which typically mirror business capabilities. A domain could be, for example, billing or identification. The small size of microservices architecture gives it superior modularity, modifiability, testability, deployability, and general ease of development. It also helps establish small checkpoints in the software development stage, which eases the separation of responsibilities among developers. Since the communication protocol is agnostic, teams can work separately, without having to corroborate with each other constantly. Summarizes the main comparison points between SOA and Microservices.

I will describe the decomposition of the system in Section 5.2.2.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

The SOLiD algorithm could definitely be implemented with a SOA architecture. Many instances of Industry 4.0 use SOA architectures to incorporate systems of many industries under one platform (Xu et al., 2018). Comparing it to monolithic systems, SOA brought upon a new age of more modular, modifiable software. The philosophy behind SOA, which looks to coordinate between heterogeneous information systems, is just taken one step further with the microservices *style*. The once complex, hard to maintain monolithic applications moved to services in a SOA environment. This made SOA the superior choice for distributed software for a long time and is still prevalent in the industry today (Xu et al., 2018). However, microservices separate the typical SOA service into even smaller services, maximizing the simplification of the code, and thus its maintainability, modifiability, ease of development, and loose coupling.

After considering all factors, like the strong requirements in terms of performance, modularity, modifiability, testability, deployability, and availability, and the decentralized nature of each agent in the SOLiD algorithm, I believe the *microservices* architecture has an edge over the SOA architecture for a self-organizing logistics planning system. I will now dive into the definition of such an architecture. Table 5-1 summarizes the comparison between the three architectural styles.

Table 5-1. Comparison between the Monolithic, Service-Oriented, and Microservices Architectural Styles.

Key Characteristics	Architectural Style		
	Monolithic	SOA	Microservices
Granularity	Single Package	Coarsely Grained	Finely Grained
Communication Protocols	Request/Response	Heavyweight Smartpipe	Lightweight messaging
Databases	Single Database	Shared Database	Database per Service
Transaction consistency	Transactions are local and guaranteed	Smart pipe orchestrates, a shared database makes transactions local	No orchestrator, the system depends on messaging
Security	Simple	Moderately Simple	Complex
Modularity	-	+	++
Modifiability	-	+	++
Performance	-	+	++
Availability	-	+	++
Ease of Development	-	+	++
Scalability	-	+	++
Deployability	+	++	++
Testability	-	+	++
Ease to integrate different software languages	-	+	++
Portability	-	+	++

5.2. ***Defining the Microservices Architecture***

Defining the *microservices* architecture of a software application is not straightforward. It implies the separation of the software's functionality into very granular components, which could theoretically be done in many ways. In this chapter, I define the *microservices* architecture with a three-step process, as recommended by Richardson (2019). The first step is to identify the system operations. Operations, in this context, are an abstract request, that can be either a command that updates data or a query that retrieves data (Richardson, 2019). I defined system operations through the user *stories* of Chapter 4. The next step is to determine the *high-level domain model of the* system, which is a conceptual view of the system. I draw this sketch in Section 0. Since the objective is to find the system's services, the system must be decomposed. There are two main *decomposition patterns* to do this. In Section 5.2.2, I select and apply one of these *patterns*. Lastly, I assign tasks to these defined services in Section 5.2.3. This is also referred to as the definition of APIs.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

5.2.1. *Determining the high-level domain model of the system*

The first step towards defining system operations is creating a high-level domain model of the system, which comes from the functional requirements of the system. With the BPMN diagram and the *stories* of Section 4.2, we understood the different actors of the SOLiD algorithm and their responsibilities and capabilities. The first alternative of the system, as Vlot (2019) conceived it, has a central platform. We can condense the abilities of each agent into a High-level Domain Model of the system. This model would be that of Figure 5-3:

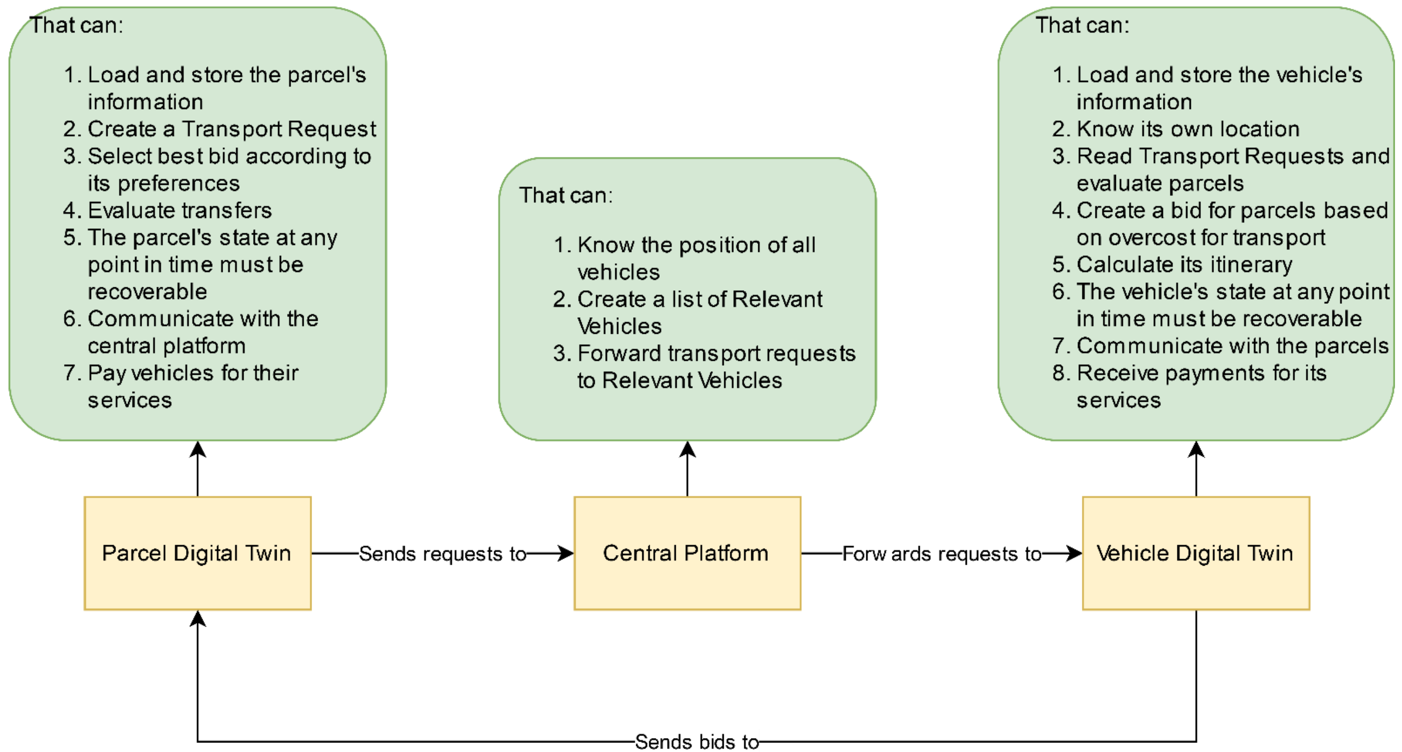

Figure 5-3. High-level Domain Model of the System with a central platform.

This sketch encapsulates the most important characteristics of the actors in the system that has the central platform. I further divide these actors into subclasses, depending on their field of expertise. For example, the parcel class can have several subclasses, like a parcel wallet subclass that handles payments, and a parcel computer subclass that generates transport requests. The parcel also must hold its information somewhere, so there could be an information subclass that does this. Furthermore, carriers mentioned that the parcel's statuses must be recoverable to create KPIs. Therefore, there must be a subclass that takes care of this task.

The central platform must have a way to forward parcel requests to vehicles and compute the relevant vehicles algorithm. Thus, there should be a central platform computer subclass. Furthermore, to compute this algorithm, the central platform must have a list of real-time vehicle positions and their centers of delivery, which could be a separate information subclass.

The digital twin of the vehicle must have a computer subclass to calculate routes and create bids. Furthermore, there should be a positioning subclass that helps it know its location and an information subclass that holds its ID, emission rate, center of delivery, etc. As with parcels, vehicle statuses should also be recoverable, so there should be a subclass that logs events. Parcels pay vehicles for their services, so vehicles should also have a wallet subclass that allows them to receive payment.

By making this separation of the actors in subclasses, I obtain the following Class Diagram of the System:

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

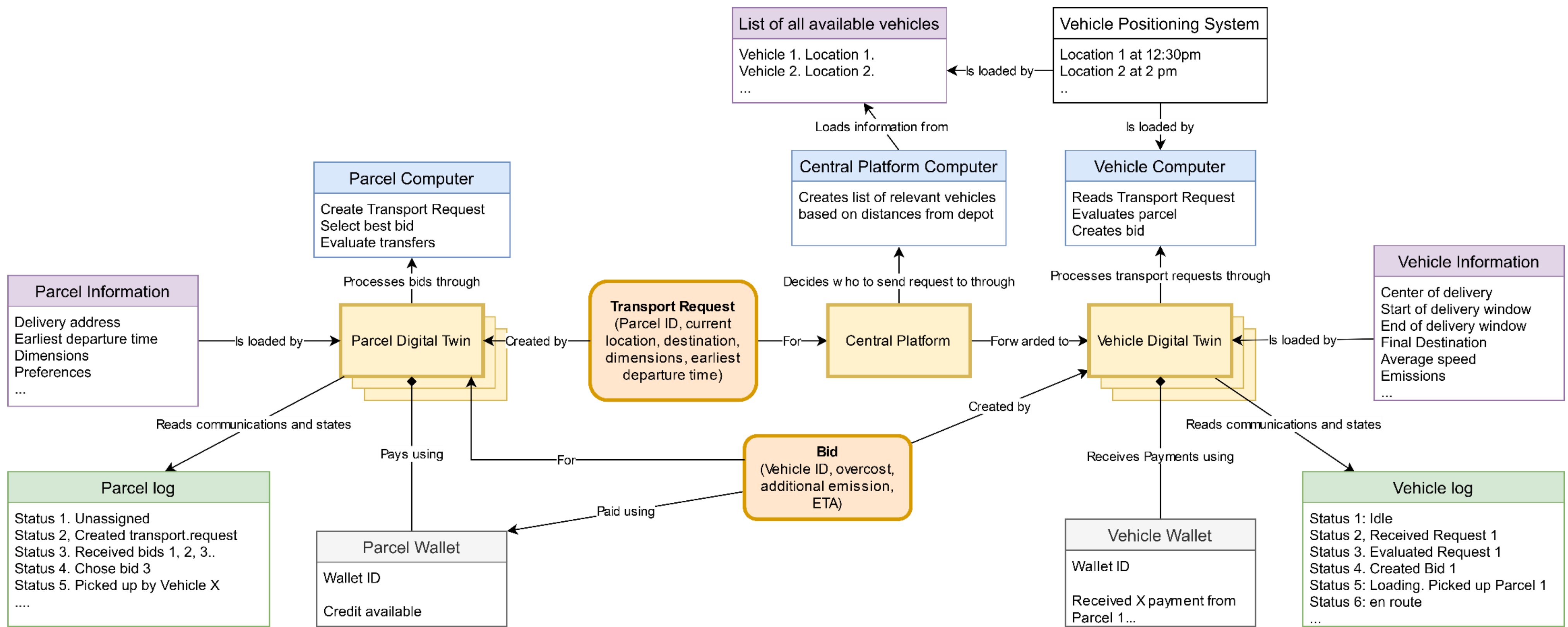

Figure 5-4. Class Diagram of the System with a Central Platform

This is an interpretation of the original algorithm by Vlot (2019). Figure 5-4 shows that the digital twin of a parcel can perform its tasks thanks to the support of several classes. The supporting classes of the parcel are:

- A *parcel information* class, from which the parcel's digital twin can load its own information, something necessary to create its transport request.
- A *parcel computer* class, through which the parcel can translate its information into a transport request and choose between the different bids it receives.
- A *parcel wallet* class, that allows it to pay the bids it accepts from vehicles.
- A *parcel log* class, that reads every significant event in the lifecycle of the parcel and logs it for future KPI analysis.

The digital twin of the parcel has two main functions: creating transport requests and evaluating bids. The digital twin of the parcel sends the transport requests to the vehicles through the central platform. This central platform also has supporting classes, which are:

- A *central platform computer* class, which creates the *list of relevant vehicles*, which are the vehicles to which the transport request created by the parcel is forwarded. To do this, the *central platform computer* class needs help from the *list of all available vehicles*. This class is similar to the *parcel computer* class, hence the matching colors.
- A *List of all available vehicles* class, from which the *central platform computer* loads its information to create the *list of relevant vehicles* to which transport requests are forwarded. In Vlot's algorithm, this is a dynamic list that is updated in real-time for all the vehicles in the system (Vlot, 2019, p. 49). Therefore, it follows that it would update its data by loading each vehicle's position from a *vehicle positioning system* class.

The supporting classes for the vehicle are:

- A *vehicle information* class, from which the vehicle's digital twin can load its own information, something necessary for evaluating transport requests and creating bids. This class is similar to the *parcel information* class, hence the matching colors in Figure 5-4.
- A *vehicle computer* class, that allows the vehicle to process transport requests, evaluating them, creating a route for itself, calculating the overcost for the parcel, and joining this information into a Bid for the parcel. This class is similar to the *parcel computer* and *central platform computer* classes, hence the matching colors. The *vehicle computer* class requires information from the vehicle's position, which is given by the *vehicle positioning system* class.
- A *vehicle positioning system* class, which updates the vehicle's position in real-time. This information is necessary for creating the real-time *list of all available vehicles* and for the *parcel computer* class to calculate its bids.
- A *vehicle log* class, that reads every significant event in the lifecycle of the parcel and logs it for future KPI analysis. This class is similar to the *parcel log* class, hence the matching colors in Figure 5-4.
- A *vehicle wallet* class, that allows it to receive payments from the parcel digital twins, specifically from their *parcel wallet* class. This class is similar to the *parcel wallet* class, hence the matching colors in Figure 5-4.

The High-level domain Model shows several rectangles for the digital twins of the parcel and vehicle, indicating that there would be many instances of them in the system.

Now, the system I suggested in Section 4.2 has a depot platform, depot digital twins, and a vehicle filter. I redraw the *High-level Domain Model of the System* in Figure 5-5:

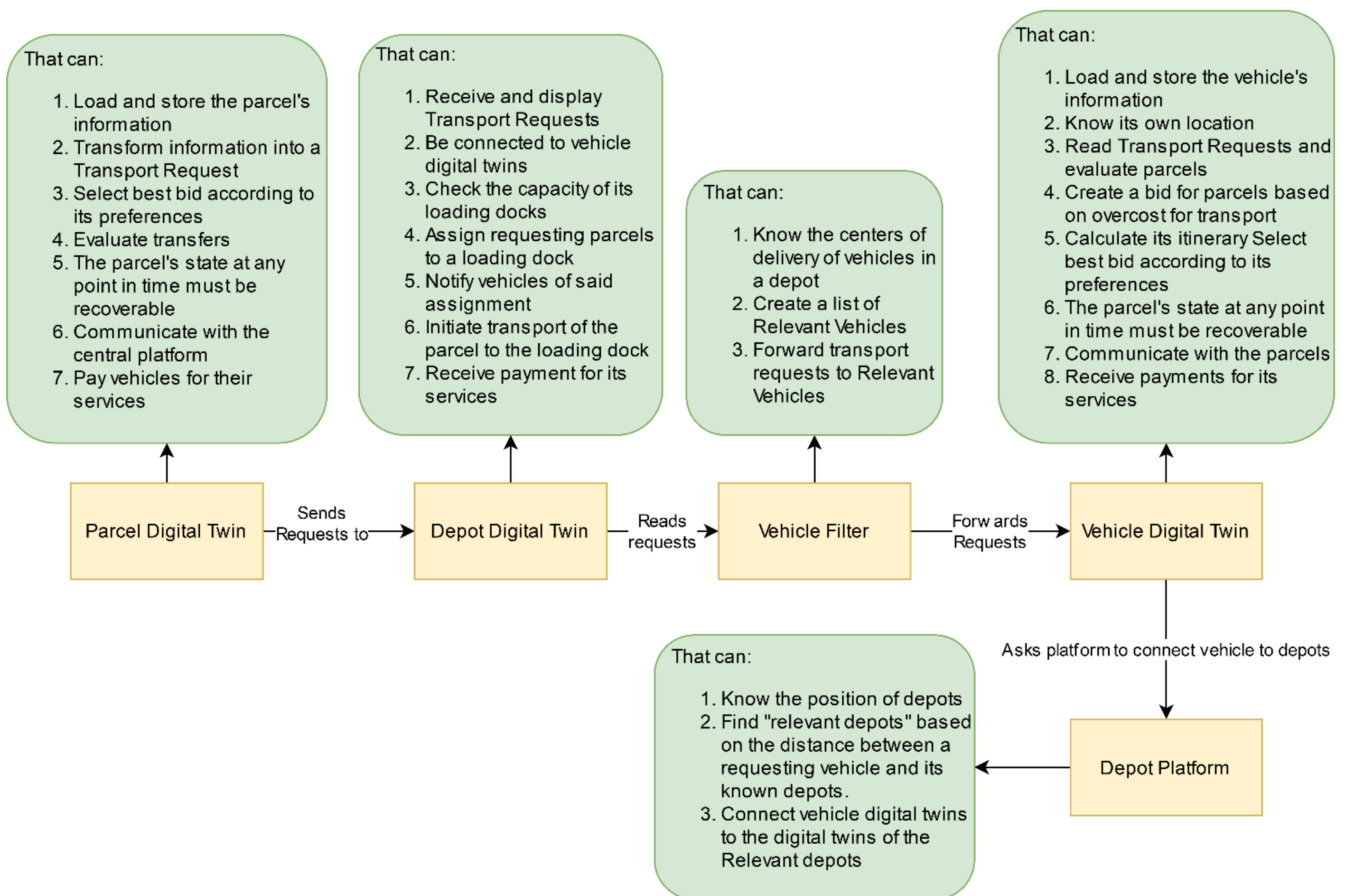

Figure 5-5. High-level Domain Model of the System with Depot Digital Twins and a Depot Platform

Figure 5-5 shows that the depot digital twins' main activities would be to receive and display the parcels' transport requests. Moreover, the Depot Digital Twin would assign parcels to loading docks within the depot and notify vehicles of this assignment. This could be done by a depot computer subclass. This computer subclass could also take on the responsibility of ordering the transport of a parcel to the correct loading dock. In practice, this would connect to the centralized planning system that governs parcel sorting, as explained in the last mile operations of Section 3.1. The requests would be read by the Vehicle filter, which would forward the transport requests to the relevant vehicles, who would consult the centers of delivery of vehicles connected to the depot.

The depot platform would have to have the information about a region's depots, which could be stored in a depot information subclass. The depot platform's main activities would be to find the "relevant depots" for each vehicle that requests to be connected to more depots, which could be done by a depot platform computer subclass. The Vehicle filter would forward transport requests only to the most suitable vehicles. Finally, to be able to be connected to a depot, vehicles request connection to a depot platform that finds the relevant depots, which in turn allows vehicles to be considered for transport requests from the parcels within.

The class diagram for this alternative would be that of Figure 5-6:

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

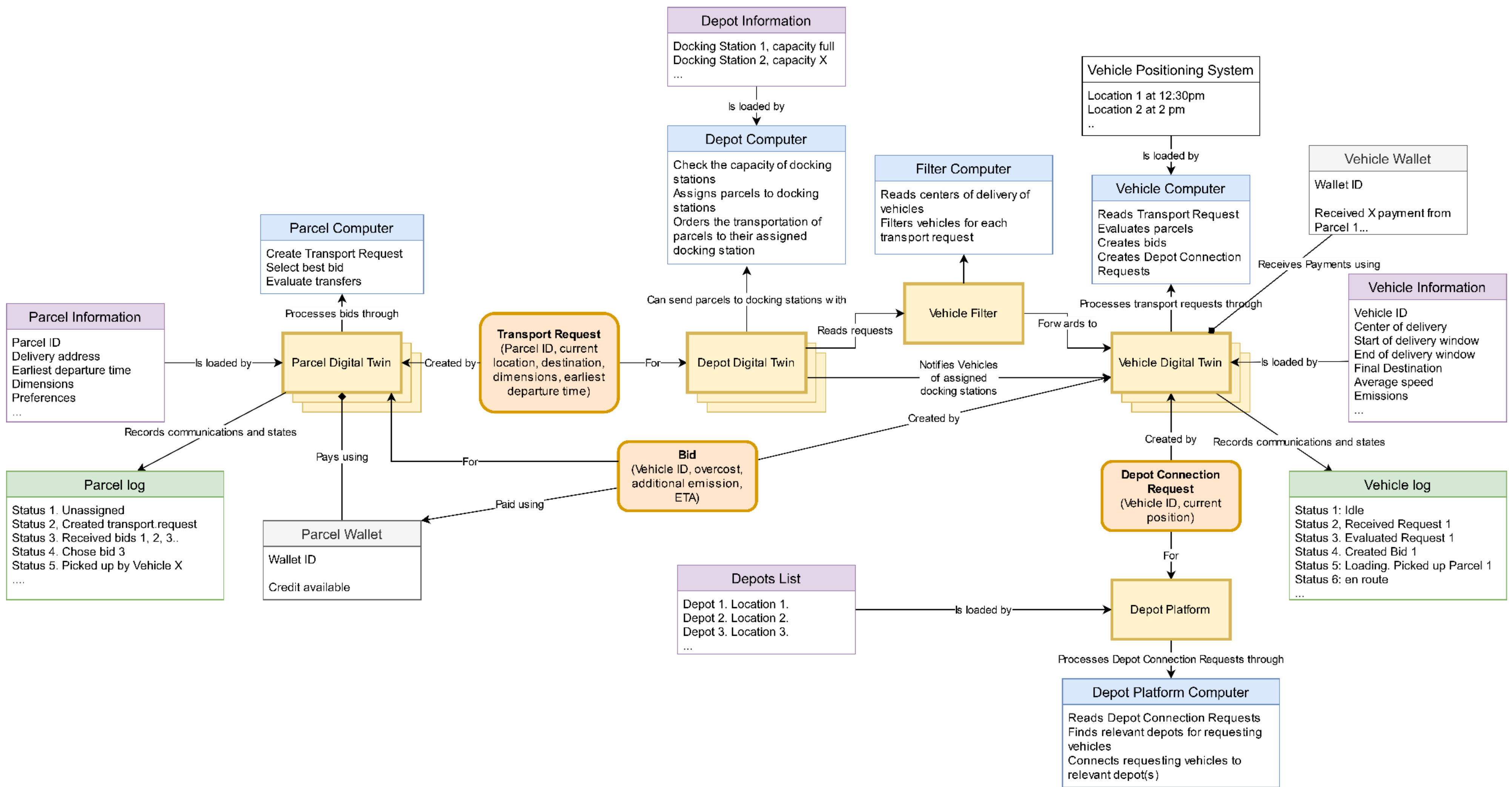

Figure 5-6. Class Diagram of the System with Depot Digital Twins and a Depot Platform

As visible in Figure 5-6, the system I suggest is more complex. While there are more actors, the parcel and vehicle digital twins are mostly unchanged. The first difference is the recipient of the parcel transport requests, which is now the Digital Twin of the Depot where the parcel physically is. This Depot Digital Twin has several supporting classes to perform its activities:

- A *Depot computer* class, which can check the capacity of the depot's docking stations and assigns requesting parcels to them. Furthermore, this computer would also inform the vehicle that is picking up the parcel of the docking station that the parcel was assigned to. Finally, this computer could order the transport of the parcel to the assigned loading dock. This *depot computer* class needs help from the *Depot information* class.
- A *Depot information* class, that would dynamically update the capacity of the depot's docking stations.

The Vehicle Digital Twin has more activities in this model, as its *Vehicle Computer* class must be able to create Depot Connection Requests. These include the vehicle's ID and position. These Depot Connection requests are sent to a new actor, the *Depot Platform*, whose main activity is to connect requesting vehicles to nearby depots. It does this thanks to two supporting classes:

- A *Depot Platform computer* class, which compares the position of the vehicle to the position contained in its *Depots list* class and finds "relevant depots". Then, this computer connects vehicles to nearby depots.
- A *Depots list* class, that holds the position of a region's depots. This would be updated every time a depot is added to the system but would be mostly static.

This sums up an important milestone in decomposing a system and gives us a visual representation of the system's architecture. However, it is still too abstract. I explored the specific operations of each of the classes and their supporting collaborators from Figure 5-6 in Appendix G.

5.2.2. Decomposing the Logistics Planning System in Microservices

Now that the system operations and the system classes have been defined, I can divide the system into services. However, before applying a decomposition *pattern*, I will introduce some basic decomposition principles developed by Robert C. Martin in the field of Object-Oriented Design (Martin, 1995). The first principle is the single responsibility principle:

A class should have only one reason to change.

-Robert C. Martin

This tells us that each class (and the microservices that arise from it) should have one responsibility, and therefore only one reason to change. If a class has more than one responsibility, there is a risk of the class being unstable, which means it can behave undesirably.

The second decomposition principle is the Common Closure Principle:

The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all classes in that package.

-Robert C. Martin

This is related to the coupling of classes. If a particular event alters more than one class, then those classes should be in the same package. If the class packages are neatly identified, it is clear for developers which classes need to be updated after a change in the code in another section. We see that the opposite is undesirable: if there is no certainty of which classes are affected by a particular change, then there will be issues at the moment of implementing said change, slowing down production and reducing maintainability and deployability.

There are other obstacles to decomposing a system. For example, the potential existence of *god classes*. A *god class* is a class that spans many parts of a given software, and thus has many relationships that make it difficult to separate the system (Riel, 1996). For example, in the High-level domain models of the system of Figure 5-4 and Figure 5-6, the Vehicle Digital Twin and the Parcel Digital Twin classes are good candidates for a *god class*, since they span multiple elements that perform very different activities.

The decomposition pattern that I will utilize in this thesis project is the Decomposition by Business Capability *pattern*. This approach separates the system into different areas of expertise, and thus it is a separation into what a system *does* (Richardson, 2019). Similar to the decomposition done in the Class Diagrams of the system of Figure 5-4 and Figure 5-6, decomposition by business capability focuses on what a system does to generate value. For example, the vehicle and parcel wallets focus on the payment capacities of the agents. The benefit of decomposing a system in this way is that business capabilities are stable because the system will not change *what* it does. However, business capabilities do not pertain to *how* a task is done, and this could be quite different across iterations. For example, a logistics company's business capability to calculate vehicle routes is a stable category, but it can be performed by a central platform that calculates all routes for all vehicles or by each vehicle itself on a bidding system, as is the case in the SOLiD algorithm. Thus, we see that for this example the business capability is the same (calculating routes), but *how* it is done can change drastically.

While this characteristic makes it difficult to do a complete overhaul to the architecture, decomposition by Business Capability is still an intuitive and powerful tool to separate software systems. One of the main alternatives to decomposition by business capabilities is decomposition by subdomains, a concept derived from Domain-Driven Design. This approach is also a powerful tool, but to separate a system using subdomains requires the understanding of many concepts that are outside the scope of this thesis project, like *domains, subdomains, bounded contexts*, *domain models*, and the *ubiquitous language* (Evans, 2004). Furthermore, the main premise of Decomposition by Business Capability *pattern* is separating a system based on functionality, which aligns perfectly with the conceptual models developed for the SOLiD project. Thus, it becomes accurate and intuitive to use this pattern to decompose the system.

The core business capabilities of each agent in the Self-Organizing Logistics Planning System are the following:

- Parcel Capabilities:
  - Parcel Information Management – Manages the parcel's information and statuses
  - Transport Request management – Enables parcels to create transport requests
  - Vehicle Selection – Enables parcels to select one vehicle based on its preferences
  - Transfer Evaluation – Enables parcels to evaluate a transfer at a depot
  - Transfer to dock request management – Enables parcels to request transport to a loading dock
  - Parcel Wallet – Enables parcels to make payments
- Vehicle Capabilities
  - Vehicle Information Management – Manages the vehicle's information and statuses
  - Bid Creation – Enables vehicles to create bids
  - Vehicle positioning – Enables tracking the vehicle's positioning in real-time
  - Vehicle Wallet – Allows vehicles to bill parcels and receive payments
- Depot Capabilities
  - Depot Information Management – Manages depot information and capacity
  - Parcel Assignment – Assigns parcels to available loading docks in the depot
  - Depot Wallet – Allows depots to bill parcels and receive payments
- Filter Capabilities
  - Filter vehicles – Reads transport requests and forwards them to the relevant vehicles
- Depot Platform Capabilities
  - Depot Location Management – stores the depot's locations, can add and remove depots from the system

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

- Find Relevant Depots – Decides what depots are relevant for requesting vehicles
- Depot connection management – Connects vehicles to the digital twins of relevant depots

I map these business capabilities onto services in Figure 5-7:

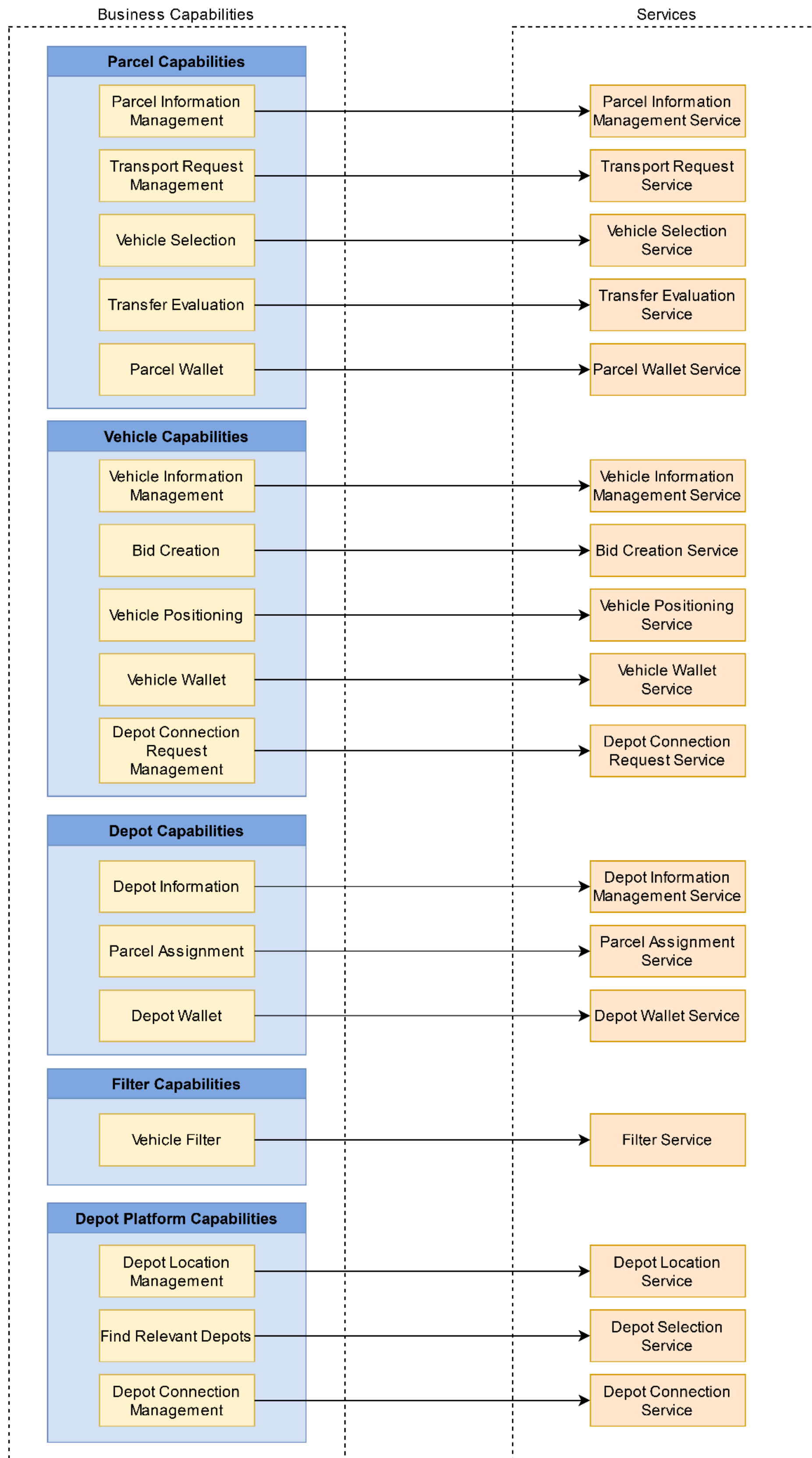

Figure 5-7. Mapping the logistics planning system's business capabilities onto services

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

The result is a decomposition in services based on business concepts rather than technical terms. This separation has many benefits. For example, if the parcel digital twin were a class on its own, it would definitely be a *god class*, since it would not comply with the single responsibility principle defined by Martin. I avoid this by separating the parcel class into 5 distinct services that are part of the same package called "parcel". I make this package comply with Martin's Common Closure Principle (Martin, 1995). By doing this, the "parcel" never becomes a single service, but a collection of loosely coupled services that perform specific actions. The same goes for the digital twin of the vehicles, depots, and the depot platform. Their *digital twins* are not a single entity but a sum of smaller parts.

With this decomposition, the parcel and vehicle functions were separated in finely grained services, such as the Parcel Information Management Service, that only holds the parcel's information, the Transport Request service, that only creates transport requests, and so on. Other problems arise, however, as the vehicles, parcels, depots, and depot platform are separated into several pieces, we must pay careful attention to the communication among the pieces. These are problems intrinsic to distributed systems. I will explore this in detail in the next chapter, where I select architectural patterns for the challenges that arise from decomposing the system.

Now that a selection of services has been done, I formally address the services' APIs (Application Programming Interfaces), which involves knowing when a service will be invoked by other services, and what conditions trigger a service's operation. Thus, I ask the following questions: When is a service invoked? What sorts of events trigger the service? Who created these events? These events are thus related to the system operations visible in the class diagram of Figure 5-6.

5.2.3. Assigning APIs to the selected services

To define the system APIs is to assign system operations to our newly defined services. I assigned all of the operations of Appendix G to the services defined in Figure 5-7. Table 5-2 contains some example operations assigned to services. The full table is available in Appendix H.

Table 5-2. System Operations assigned to the new services

Actor	***Command***	***Pre-Conditions***	***Result***	***Description***	***Collaborators***
Parcel Information Management System	Store Parcel Information	-	Parcel info can be loaded	Saves data (Parcel ID, preferences, earliest pick-up time, etc) so that it can be loaded by the parcel computer to create a transport request	-
Transport Request Service	Load Parcel Information	Parcel Information Management System: - Store Parcel Information	Transport Request Service can now "Create Transport Request"	The Transport Request Service loads the parcel's information to then create a Transport Request	Parcel Information Management System: - Store Parcel Information
Transport Request Service	Create Assignment to Dock Request	(a) - Parcel status changes from "unassigned" to "assigned" (b) Transfer Evaluation Service: - Parcel Requests Drop-off to the original vehicle	Depot Digital Twin is informed and can begin "Assign Parcel to Loading Dock"	When the parcel is assigned or when it found a better transport, it must be loaded/unloaded. This is done at a loading dock. Thus, the parcel requests to be assigned to one.	(a) Parcel Information Management System: - Store Parcel Information (b) Depot Connection Service: - Connect Vehicle to Depot (the parcel can communicate to the depot thanks to the vehicle's connection to it)

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Depot Information Management Service	Read and update capacity	Parcel Assignment Service: - Assign Parcel to Dock	The depot updates the capacity of its loading docks	By updating its capacity, the depot knows where to assign parcels to. Every time a parcel is assigned to a dock, loaded, or unloaded; this updates itself.	-
Parcel Assignment Service	Load Depot Capacity	Depot Information Management Service: - Read and Update Capacity, Transfer Evaluation Service: - Request Capacity to Depot	Sends the response to the parcel, there is capacity yes/no	If there is capacity for the parcel, then the Transfer Evaluation Service begins its operations	
Parcel Assignment Service	Assign Parcel to Loading Dock	Transport Request Service: - Create Assignment to Dock Request	The parcel is assigned to a loading dock and a message is sent to the vehicle	A parcel needs to be assigned to a loading dock to be (un)loaded. The relevant vehicles are notified, and the Depot Capacity is updated	Parcel Assignment Service: Load Depot Capacity
Parcel Assignment Service	Order transport to Loading Dock	Parcel Assignment Service: - Assign Parcel to Dock	The Assignment Service orders the transport of the parcel to the assigned loading dock	The parcel is sorted to the assigned loading dock where the vehicle will (un)load it	-

The preconditions column in Table 5-2 gives insight into the trigger of a particular operation. For example, whenever the parcel status changes from "*unassigned*" to "*assigned*", the Transport Request Service will *Create an Assignment to Dock Request*. To do this, however, it needs the help of the Parcel Information system, since it needs to load the parcel's information for the request. This is what the collaborators' column signifies: crucial operations that are necessary for the completion of a task. Furthermore, the assignment to dock request acts as a trigger for the Parcel Assignment service, which will attempt to assign the parcel to a loading dock. Just like the Transport Request service required the Parcel Information Management Service's help for creating an Assignment to Dock Request, the Parcel Assignment Services requires information from the Depot Information Management Service, from whom it loads the depot's capacity. Figure 5-8 shows this example graphically.

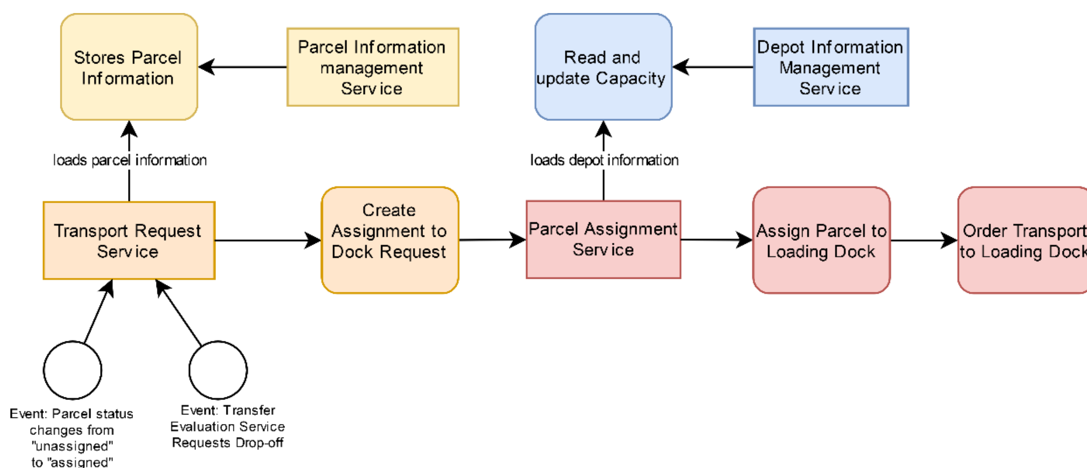

Figure 5-8. Example interpretation of operations

Theoretically, loading operations could be performed by a service whose only task is to load information. In the example of Figure 5-8, there could be another service, a Depot Information Loader, that only performed the "Load Depot Capacity" operation. However, I suggest that this operation is done by the Parcel Assignment Service. I

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

decided to keep these two operations into the same service. This achieves two things: first, it gives two clear tasks done by a single service to developers, and secondly, it avoids having an *anemic service*, which is a service that has little to no business logic (Richardson, 2019). This would be the case for the hypothetical Depot Information Loader.

Now that all operations are documented in Appendix H, all the services' APIs have been defined, including their collaborations with other services. Therefore, the decomposition of the Self-Organizing Logistics Planning System is complete, answering research sub-question 6. The next step is to can select Architectural *Patterns* to solve the inherent issues of a distributed system, such as communication, data consistency, maintainability, availability, and many more. I will explore this in Chapter 6: Application of Architectural Patterns.

5.3. Summary

In this chapter, I discussed the potential architectural styles relevant to the self-organizing logistics planning system and selected the Microservices style as the most appropriate. This answer research sub-question 5, which stated:

What is the most appropriate architectural style for a self-organizing logistics planning system?

After selecting this style, I developed two conceptual views, one of the system as Thymo Vlot proposed, and one of the system with Depot Digital Twins and a Depot Platform that I suggested in Chapter 4. Then, I specified both conceptual models into *class diagrams*. This gave a good overview of both systems and allowed a comparison of the complexity of the two, highlighting key differences. Then, I took the activities in the second class diagram and applied the Decomposition by Business Capability *pattern*, which led to a group of services in Figure 5-7 with their relevant operations and collaborators in Table 5-2. By defining these services and their individual responsibilities, I have decomposed the self-organizing logistics planning system, answering research sub-question 5, which stated:

How to decompose a self-organizing logistics planning system?

In the following chapter, I apply architectural patterns to address the problems inherent to any distributed system, which would also be present in the separation in services that resulted from this chapter.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

6. Application of Architectural Patterns

The second half of Phase 2 of the design process concerns the selection of architectural patterns to design the planning system. The microservices architectural *style* selected in Section 5.1 offers superior scalability and ease of development than the alternative *monolithic* and *service-oriented* architectures. However, choosing this *style* brings up several problems and areas of interest that need to be addressed. In this chapter I present the patterns that address these problems one by one and compare each potential solution, thus seeking to answer research sub-question 7:

7. *Which architectural patterns would best suit a self-organizing logistics planning system?*

To choose between patterns, I utilize the customer requirements of Section 3.2. I will summarize the process graphically by using a decision tree that shows available patterns applicable to the microservices architectural *style*. Figure 6-1 summarizes the research question, activities, and deliverables for this chapter.

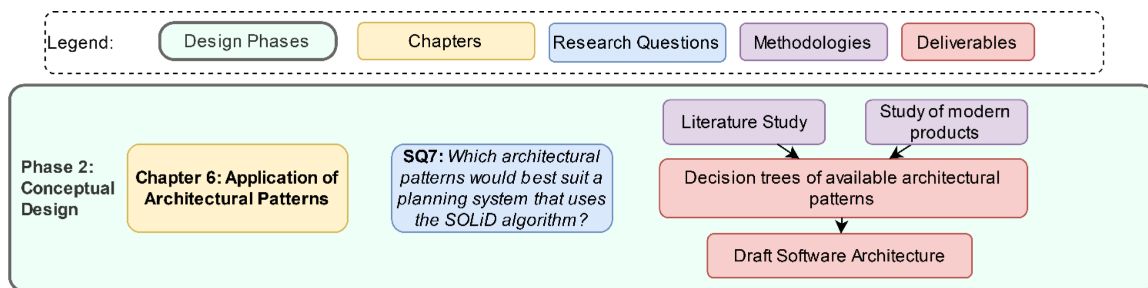

Figure 6-1. Phase 2, Conceptual Design, adapted from Systems Engineering Design methodology by Dym et al. (2014)

6.1. ***Addressing customer requirements with architectural patterns***

In Section 3.2 I found that availability, security, and data transparency were very important for Carriers and that Performance, Modularity, Modifiability, Testability, and Deployability were the most important Quality Attributes for Prime Vision. From the functional requirements of Section 4.2, I also gathered that most actors must communicate with one another and that the process has an orchestrated nature. Furthermore, by separating the system into microservices, there are several issues to tackle since the system is distributed. Each of the following sections explores the most important topics that must be addressed in our microservices architecture to comply with the customer requirements of Chapter 3 and the functional requirements of Chapter 4.

6.1.1. Interprocess Communication, Transactions, and Business Logic

Communication between services is one of the most important aspects of a distributed system. Given the small size of the services defined in Section 5.2, many services must interact with one another for the execution of all the logic of a Self-Organizing Logistics System. Services can interact on a one-to-one basis or a one-to-many basis. As the name implies, in one-to-one interactions are between two individual services. Conversely, one-to-many interactions are sent by one service and received by many.

6.1.1.1. The Synchronous Remote Procedure Invocation pattern

Interactions also differ by the timeliness of response: there are synchronous and asynchronous interactions. Synchronous messages include classic request/response interactions. Request/response interactions result from applying the Synchronous Remote Procedure Invocation *pattern*. In this *pattern,* the services involved are actively waiting for an immediate response of the other service or services. In our system, this occurs when a vehicle requests connection to more depots to the depot platform: the vehicle requests connection to the platform alone, and the depot platform performs its actions immediately. With these kinds of interactions, the involved services are usually tightly coupled, which means that a requesting service will stop its activities until it receives a response

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

from the other service. In the logistics planning system, however, only the Depot Connection Request system would be tightly coupled. The other functions of the vehicle digital twin, like creating bids, are done by other services such as the Bid Creation Service, which function independently from the Depot Connection Request system. Thus, we avoid unnecessary tight coupling.

But why is avoiding tight coupling so important? The answer is that tight coupling (and thus synchronous messaging) reduces the availability of a service (Bass et al., 2003; Richardson, 2019). And availability is an important requirement for the Carriers that will use the system. Let us show this with an example: Say Service A requests something of Service B. While Service A is waiting for a response from Service B, it stops doing its other functions. Thus, if another service, Service C, invokes Service A, it will not be available to respond to the invocation because it is waiting for the response from Service B, thus being *unavailable*.

6.1.1.2. The Request/Asynchronous response communication pattern

The alternative to the Synchronous Remote Procedure Invocation *pattern* is the Asynchronous messaging *pattern*. Asynchronous messaging does not require an immediate response from a service, and thus the receiving service, the *client*, does not block the requesting service, and if there is a response, it does not have to be immediate.

There are two types of interactions when using the Asynchronous messaging *pattern*: the request/async response interaction and the publish/subscribe interaction. The request/asynchronous response *pattern* uses a requests channel, owned by the service, and a response channel, owned by the client. The basic idea is that the client requests something of the service and specifies a return address for its message. The service then processes the request and sends a response in the specified address, which is a different channel owned by the client.

For example, when the vehicle makes a Depot Connection Request, this could very well be a request/response. However, it is possible to implement it as a request/async response. As a request/response, it would be a direct communication between the vehicle and the depot platform. As a request/async response, the request would be sent to a requests channel where the depot platform could queue up requests and attend to them as soon as possible. This would avoid locking the depot platform and the vehicle. In the case a vehicle does not get a timely response, it can default to not connecting to nearby depots. We can compare this process to the classic request/response interaction in Figure 6-2:

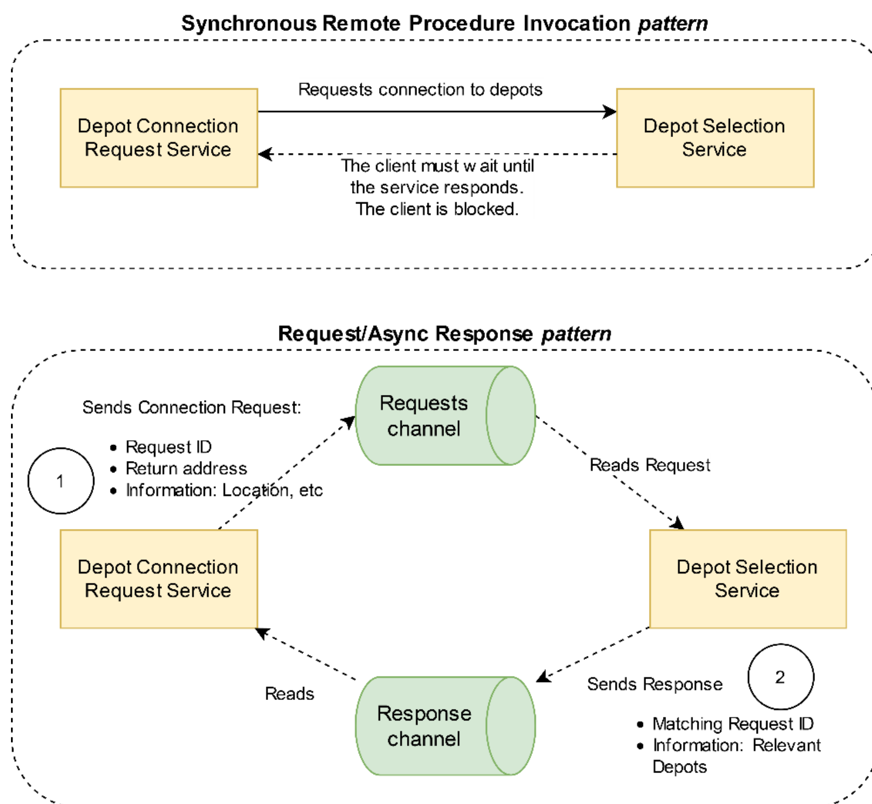

Figure 6-2. Example of Synchronous Remote Procedure Invocation pattern *vs. Request/Asynchronous Response messaging* pattern.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Figure 6-2 shows the classic request/response interaction that begins with a *client* (in this case, the Depot Connection Request Service) sending a request to a *Service* (in this case, the Depot Selection Service). Since the communication is direct, the client's resources are blocked until the Depot Selection can respond. If the Depot Selection Service is busy, then there will be a partial failure in the system or the *client* will have to repeat its request until the service can answer. In the second case, the client's request is sent to a Request Channel, which acts as a storage for the service's requests. These could be stored temporarily or permanently. The critical difference is that this channel acts as a *buffer*, which means that it holds the requests until the service can respond to them. This avoids the need for the client to resend the request many times. Furthermore, if the service is busy, there will not be a system failure. The same is true for the responses, as they can be stored and consumed whenever the client is ready to do so.

This communication *pattern* helps guarantee *loose coupling* of services, an indispensable characteristic in a distributed system. To guarantee an orderly consumption of messages, the message channels should store messages in order. Another tactic to keep orderly consumption is for each service to have a separate, partitioned channel for each service that they want to communicate with, also known as *sharded* channels (Richardson, 2019). This works with the same logic as communication between smartphones: there is an individual chatroom per each contact on a smartphone instant messaging app, as there is a *sharded* channel for every entity a service communicates with. A service can access the chatroom it has with another service and read its messages in order, thus aiding with consistent processing of events.

This interaction pattern would fit many requesting services in our self-organizing logistics planning system, like the parcel's requests: from the assignment to loading dock request, to all the requests done by the Transfer Evaluation service, like the Capacity check, the Drop-off Request, and the Equivalent Bid Request.

6.1.1.3. The publish/subscribe communication pattern

The other asynchronous communication within the asynchronous messaging pattern is the *Publish/subscribe* pattern. This pattern is very similar to the request/async response pattern, but the channels to which services send messages to are accessible by one or many services, hence *publishing* messages. Just like the channels in request/async response, the *topics* in publish/subscribe interactions are also a *buffer* to messages: they can hold requesting messages for busy services, thus aiding availability, and reducing repeated messages. Going back to the smartphone example, *topics* in a *publish/subscribe* system work like group chatrooms in an instant messaging application that services can go to and read previous events whenever they want.

With *topics*, many of the system's operations can be handled in a way that they use publish/subscribe systems instead of a request/response interactions. Thinking of our logistics planning system, the Transport Request Service needs the parcel's information to create a Transport Request. Thus, the Transport Request system needs to load the parcel's information from the Parcel Information Management Service. This operation would traditionally be a request/response: The Transport Request Service would request the parcel's information to the Parcel Information Management Service, and it would wait for this service to give it the necessary information. However, if the Parcel Information Management Service *publishes* the parcel's data on a *topic* to which the Transport Request Service is subscribed to, then it would not have to wait for the Parcel Information Management Service to load the parcel's information and then create the Transport Request. This *publishing* system would greatly aid the availability of the Transport Request Service. I compare the request/response method to the publish/subscribe method in Figure 6-3:

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

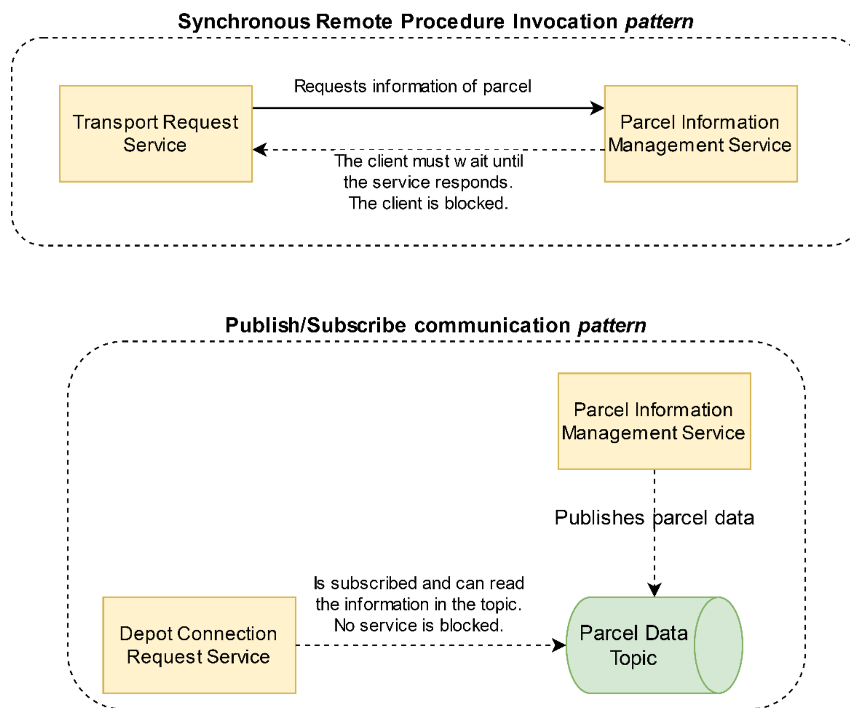

Figure 6-3. Example of Synchronous Remote Procedure Invocation pattern *vs. Asynchronous messaging* pattern.

From this example, it is possible to gather that it is advantageous to substitute request/response interaction with publish/subscribe systems or request/async response whenever it is suitable. This is especially true since the *topics* can *buffer* messages, which means to store messages they are consumed, therefore avoiding the need to repeat a request several times. Furthermore, some message brokers allow accessing information that was published before the time of connection. In our self-organizing logistics system, this would be useful for vehicles that connect to new depots, as they would be able to read unfulfilled transport requests. This functionality would give certain message brokers priority when considering technologies for the system in Chapter 7.

The *publish/subscribe* messaging *pattern* would fit other services in the Self-Organizing Logistics Planning System. For example, when the Transport Request Service creates a transport request, it does not need an immediate response from the vehicles. It also needs to communicate with many vehicles at the same time. Thus, a communication pattern that allows it to publish its request would suit this service. This is possible by applying the *Publish/subscribe* pattern, in which the requests are readable by vehicles that are subscribed. Therefore, it seems fitting that the parcel requests are *published* on a board belonging to the depot that holds the parcel. This way, all the vehicles connected to the depot could read their requests. Furthermore, if the topic allows the discovery of old messages, then the parcel does not need to repeat a request. If the message broker does not have this functionality, then the parcel should repeat this request each time a new vehicle is connected to the depot. This would continue to be a publish/async response message, and it would not be tightly coupled to any vehicles, thus solving the unavailability problem.

In the same manner as parcels publish their transport requests, vehicles can publish their bids for the parcels to read. This could be done using the same depot topic, or a response topic set up by the parcel. In either case, the vehicles would send their bids asynchronously and no service would be blocked.

6.1.1.4. Incorporating request/async response and publish/subscribe communication patterns

These asynchronous communication patterns can be applied symbiotically. If applied for the moment a parcel requests assignment to a loading dock, the result is the interplay between services visible in Figure 6-4:

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

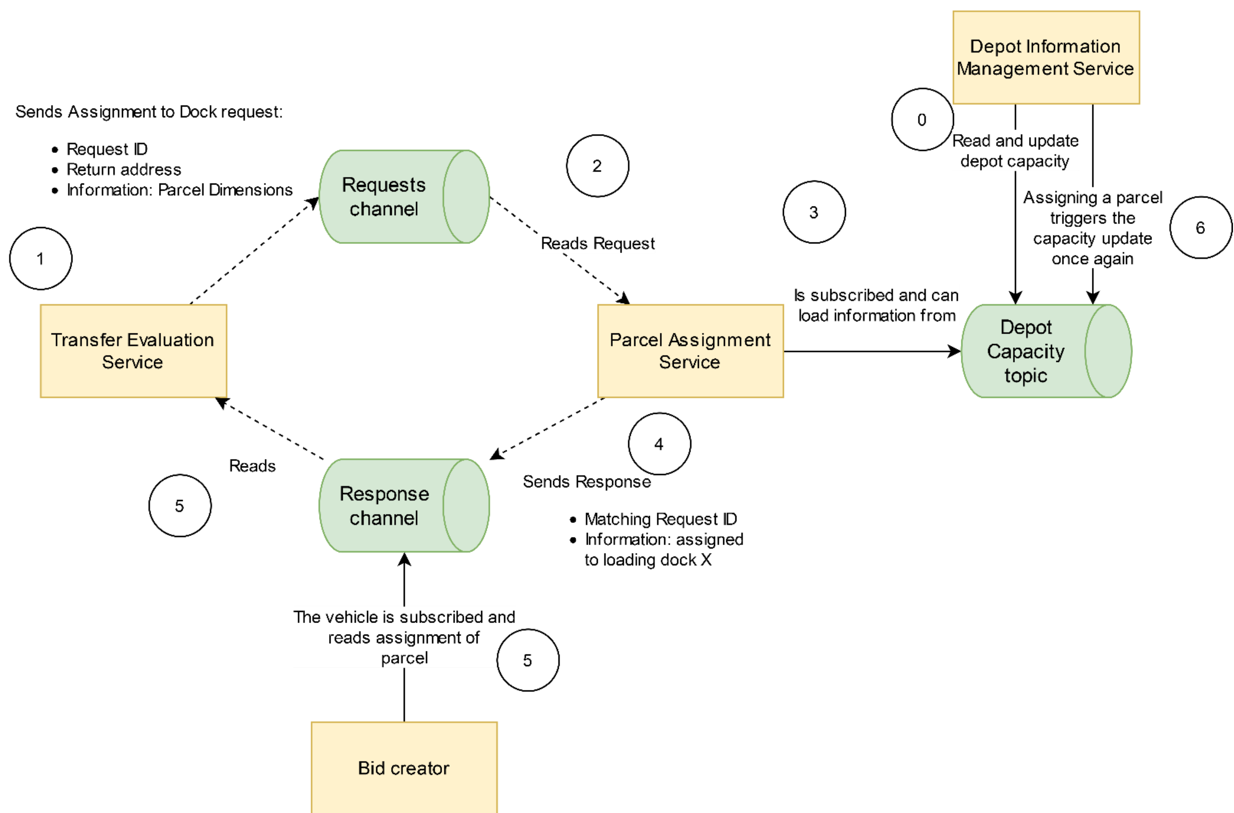

Figure 6-4. Transfer Evaluation service requests Assignment to a Loading Dock.

In this process, the Depot Information Management Service has already updated the Depot's Capacity at step 0 of Figure 6-4. The depot's capacity is therefore updated and kept in a Depot Capacity Topic that the Parcel Assignment Service can read because it is subscribed to it. Then, when the Transfer Evaluation Service Requests to be assigned to a loading dock so that it can be dropped off or picked-up, it sends this request to a request channel in step 1. Then, the Parcel Assignment Service reads this request in Step 2 and loads information from the Depot Capacity topic to which it is subscribed to in Step 3. Then, the Parcel Assignment Service sends its response with a matching Parcel ID to the response channel in Step 4. This response channel can also act as a topic to which the vehicle is subscribed to. Therefore, the vehicle can read to which loading dock the parcel was assigned to in Step 5 and begin going towards it. At the same time, the parcel can read the response and log the event if it wants to. Finally, whenever a parcel is assigned to a loading dock, the Depot Information Management Service updates the depot capacity once again in Step 6, thus maintaining the topic up to date for later queries.

Figure 6-4 shows that messaging helps orchestrate transactions between services. This is a very important aspect of a distributed system, where orchestrating operations that span many services is a challenging task. This sort of communication between many small services to carry out an operation is known as a *saga*, and they are the best solution for distributed transactions in a microservices environment (Richardson, 2019). More specifically, the type of *saga* used in Figure 6-4 is considered a *choreography-based saga*, because there is no orchestrator telling services to enact their operations. The *orchestrator saga* could be considered as an alternative in the development phase if the fulfillment of the activities becomes too complex. However, this thesis project will limit itself to the *choreography-based saga*.

Figure 6-4 also shows that *sagas* use the latest information and state changes of a service, since these act as notifications for other services. The way the services are separated, and how they work with their latest updates, is known as an event-driven architecture, an increasingly popular style of developing distributed systems (Waehner, 2020). This technique of utilizing a service's significant changes to notify other services and interested parties is known as *event sourcing* (Richardson, 2019).

Event sourcing helps maintain services up to date, and enables the usage of orchestrated operations like the ones visible in Figure 6-4. For event sourcing to be effective, it is necessary that services in a microservice environment

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

publish all their events, so that data can be eventually consistent across services (Bass et al., 2003; Richardson, 2019; Zhu, 2005). Therefore, the business logic of each service must update its state onto a visible topic, called *publishing events*. Event publishing clearly fits the Self-Organizing Logistics Planning System. Event publishing can play a key role in the Information Management Services for the parcels, vehicles, and depots. Under this *pattern,* these services basically become topics where these actors publish their important events, which are easily accessible by other services later.

One of the key success factors for effective sagas is having granular operations that can be performed by one microservice. Then, with asynchronous messaging, it can be ensured that events will be visible as messages that are also buffered in communication channels. This way, services can consume them whenever they are available, leading to the fulfillment of all necessary operations.

Most of the self-organizing logistic planning system's operations will result in a service interplay similar to that of Figure 6-4. Thus, the system will have many *sagas* that synchronize operations. However, if a service must be implemented using synchronous response/request, then developers should apply the Circuit Breaker *pattern* to avoid failure due to an unavailable service.

The Circuit breaker *pattern* is applied when using synchronous requests and it seeks to avoid failure when a service is unresponsive. The *pattern* implies adding a counter for the attempted requests a service has made and the request failures caused by an unavailable service. If the number exceeds a certain threshold, then the circuit breaker comes into action and further requests fail automatically. Then, a timeout can be implemented so that no further requests are done after a certain time, after which the circuit breaker can be deactivated, and the requesting service can try again.

Other asynchronous messages include one-way notifications, which are directed to a particular user and do not expect a response at all. In the Self-Organizing Logistics Planning System, this would be used to tell the customers the expected time of delivery of a parcel, for example. Using messaging channels like topics and services that publish their important events, it becomes simple to implement a service that only reacts to this type of event (for example, when a parcel becomes assigned to a vehicle and it can read its itinerary) and sends a notification via email or mobile app.

This Section spanned Interprocess Communication, Transactions, and Business Logic for the Self-organizing Logistics Planning System. I summarize the design decisions in a decision tree in Figure 6-5:

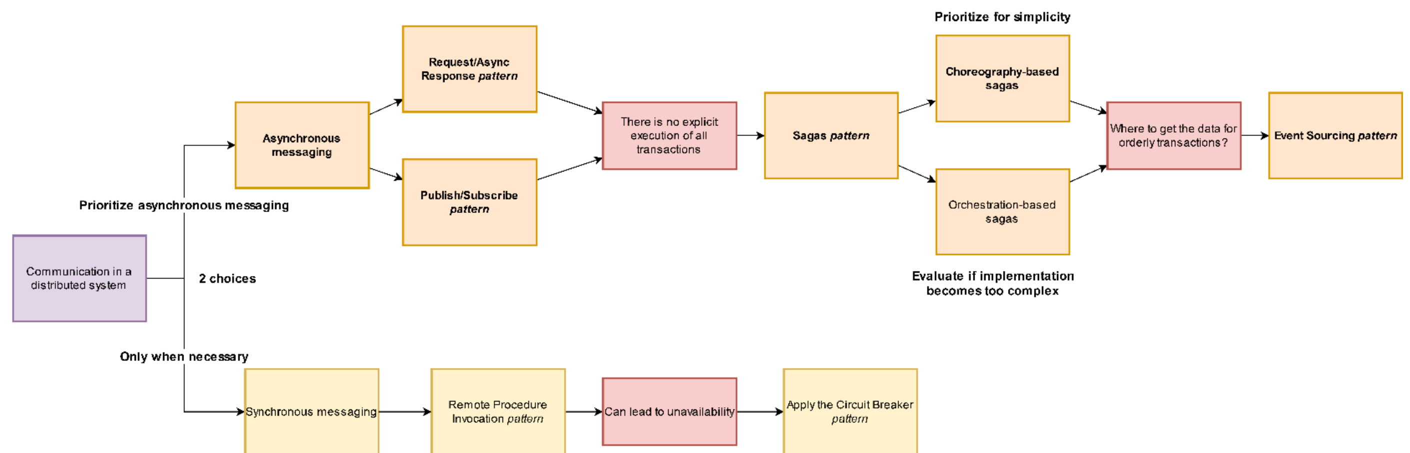

Figure 6-5. Decision Tree for communication, transactions, and business logic in a distributed system

Messaging also involves using a certain format or *language*. Furthermore, messages use an explicit structure called a *schema*, which *consumers* need to know to use the information in the message. While it is not in the scope of this project to define each schema for each API, the architecture of the system should have a mechanism that aids in handling these different message formats and languages.

The tools that allow us to handle these different languages and formats are a schema registry and a serializer. The serializer makes the data fit a particular format or schema, and the schema registry stores all the different schemas

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

and acts as a translator between the parts (C++ Foundation, 2015; Cloudera Docs, 2019). This system would be between the main message broker and the services that use the message broker's data, as shown in Figure 6-6.

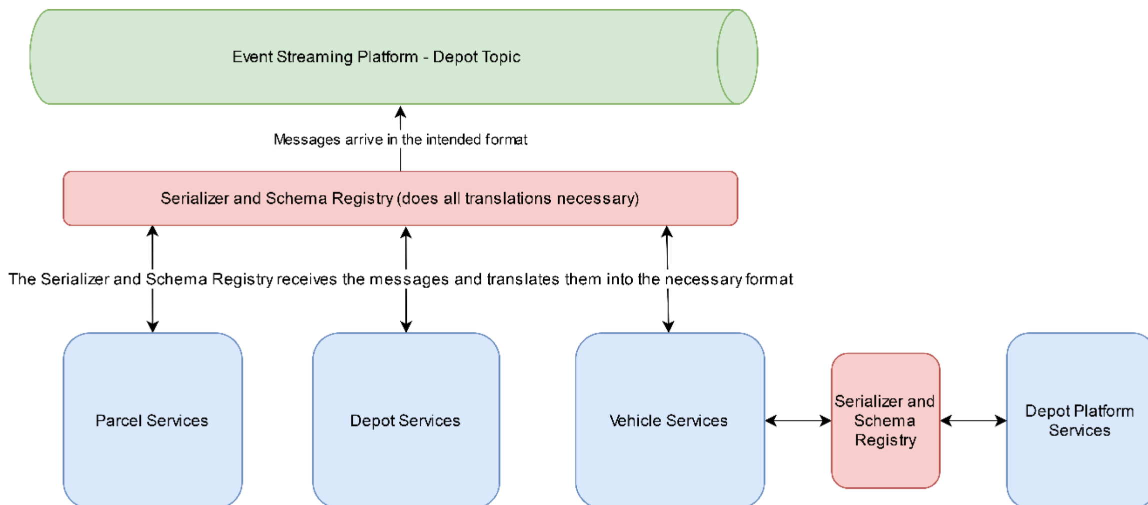

Figure 6-6. Applying a Serializer and Schema Registry in the Self-Organizing Logistics Planning System.

6.1.2. Data Management and Queries in a distributed system

A query is an information request in a software system. It is one of the basic functions any service can have, and many queries happen at any given moment in most software systems. Traditionally, if you have a monolithic application or a SOA-based application, you have a main database that all services draw information from. Querying these systems is simple, as there is only one database to get information from. However, by implementing microservices, each small service has its own database. This keeps querying information from being straightforward (Richardson, 2019). There are two patterns that help solve the data management problem: The API composition *pattern* and the Command Query Responsibility Segregation (CQRS) *pattern*.

6.1.2.1. The API composition pattern

The API composition *pattern* consists of grouping several APIs or service addresses, querying each of them, and then combining the results (Dev.to, 2020; Richardson, 2019). This pattern employs a special service called an API composer that saves a series of combined queries. When a query is to be invoked, then the API composer is invoked, and it performs the required query. We can think of the API composer as a materials procurer. Whenever a client places an order for materials, the procurer fetches all the required parts and then sends them to the client. The API composer can be a dedicated software component like an API Gateway, or an existing service that also acts as a composer. This is a simple *pattern* to apply, and it should be applied whenever it is feasible.

The API composition *pattern* makes queries in a distributed system simple and possible, but it also adds overhead to the system, as the compositor needs to retrieve data from multiple databases, thus making several requests. The resulting system requires more computational power than the original.

Furthermore, by invoking several services, the availability of the system is compromised, like in the request/response mechanisms of Section 6.1.1.1. Allow me to illustrate this with a numeric example: Say there are 4 services, Services A, B, C, and D. Service D will the compositor, and will thus get information from Services A, B and C. If services A, B and C are available 99% of the time, then the probability that Service D is successful in getting information from services A, B, and C is only $99\%^{(3)}$ =97% of the time. This occurs because the success of a query depends on the availability of all services.

This problem can be tackled by using cached information whenever a service is unavailable. A cache is a temporary storage of previous states. As it is a collection of past states, there is always the risk of loading outdated information. For some services, however, this is a fine solution. Another strategy is to let the API composer bring incomplete data: If service A and B are available, but C is unavailable, then the composer only brings back the latest data from services A and B. These two strategies can and should be implemented in unison whenever suitable.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Another problem with API composition is that of inconsistent data. Distributed systems naturally lack the consistent, timely, and isolated ACID transactions common in simpler, single database systems. For example, it might occur that the depot's Parcel Assignment Service says that the parcel was assigned to Loading Dock A, but the parcel's Information Management Service is not yet updated and says the Assignment Request is still pending. The API compositor can be developed in a way that can discover this inconsistency and resolve it, but this adds additional overhead, hurting performance and scalability.

6.1.2.2. The Command Query Responsibility Segregation (CQRS) pattern

The alternative to the API composition *pattern* is the CQRS *pattern*. Some software systems combine Relational Databases and smart text search databases. On one hand, the relational databases act like a structured record that can organize data in rows with unique IDs and columns that mark the attributes of the data items. On the other hand, text-based search engines simplify querying data from these relational databases. We can think of the relational database as the backroom of a shoe shop with many boxes neatly organized in stacks, and the text-based search engine as an experienced clerk who can find the right shoebox quickly and efficiently. The CQRS *pattern* comes from this combination of systems. As the name implies, Command Query Responsibility Segregation separates commands from queries. Thus, data is split into a command side and a query side. The command side has its own transactional database, and it stores the system's operations as they are made. For example, create transport requests, check the depot's capacity, eliminate parcels from an itinerary, etc. The query side reads these operations with an *event handler* and copies them onto an *event store* that keeps them for later queries. The query side stays updated by subscribing to the events created by the relevant services. Applying CQRS to a system makes it have two databases. For example, if we apply it to the Transfer Evaluation Service, the result would be Figure 6-7:

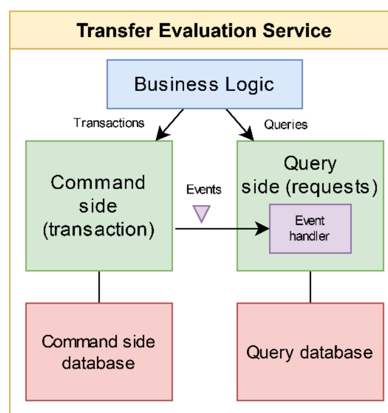

Figure 6-7. Application of the CQRS pattern to the Transfer Evaluation Service. Adapted from Richardson (2019, p. 233)

We can see the different databases for the different types of operations. The command-side database would handle the Create, Update, and Delete operations, while the Query-side database would copy all these events and handle the queries for the Transfer Evaluation Service.

With the CQRS pattern, it is possible to create a carbon copy not only of one but of many services, which is one of the many reasons why CQRS solves many of the challenges of querying a distributed system. With the CQRS pattern, we can create a service whose only function is to be queried, and this service can combine the states from many other services (Richardson, 2019). By doing this, the information of various services is in one place, so it is easier to extract information from it. Furthermore, with the CQRS *pattern*, the query side is updated asynchronously, and thus there are no one-by-one requests to the services whenever a query is made. Therefore, there is no need for extra computational power when querying these carbon copies. For example, in the Self-Organizing Logistics Planning System, the parcel's functions are scattered across several services. In Section 3.2.3, we saw that Carriers desire high visibility of the system to ease building KPIs. I solve this requirement by creating a Parcel Overview Service that records all the events from the Parcel Services.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

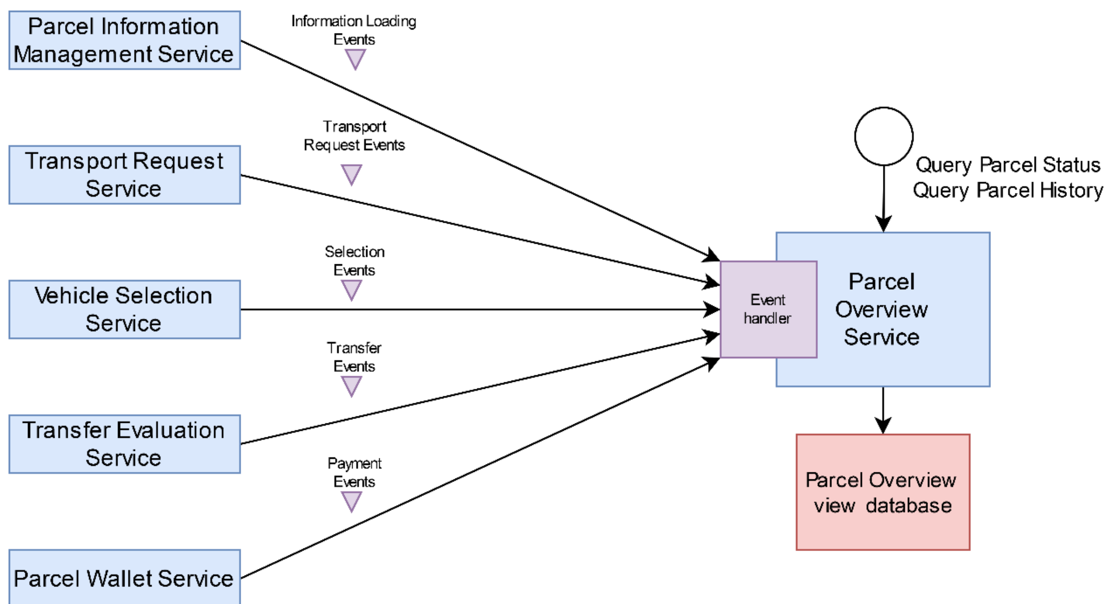

Figure 6-8. Creation of a Parcel Overview Service with a CQRS view.

If a customer wants to know about a parcel's history, then the customer would query the Parcel Overview Service and not the individual services that compose the parcel. This also hides the functioning of the parcel for the client, a hallmark of proper API handling (Bass et al., 2003; Richardson, 2019; Zhu, 2005). I implement a similar view for the vehicle digital twin.

Figure 6-8 also shows that CQRS requires a proper implementation of event sourcing since this will be the base for obtaining highly scalable and observable systems. The CQRS pattern leverages both the transactional attributes of relational databases and the querying ability of text-based search engines. Furthermore, the separation of concerns within each service makes both the command side and the query side of a service simpler and thus simpler to maintain and deploy. In conclusion, I believe CQRS should be the preferred option for the more complex queries in the system, such as querying all the parcel or vehicle statuses. For simpler queries, such as querying the depot platform and the depot digital twins, the API composition pattern would be more appropriate. We can summarize design decisions for our planning system in the following decision tree.

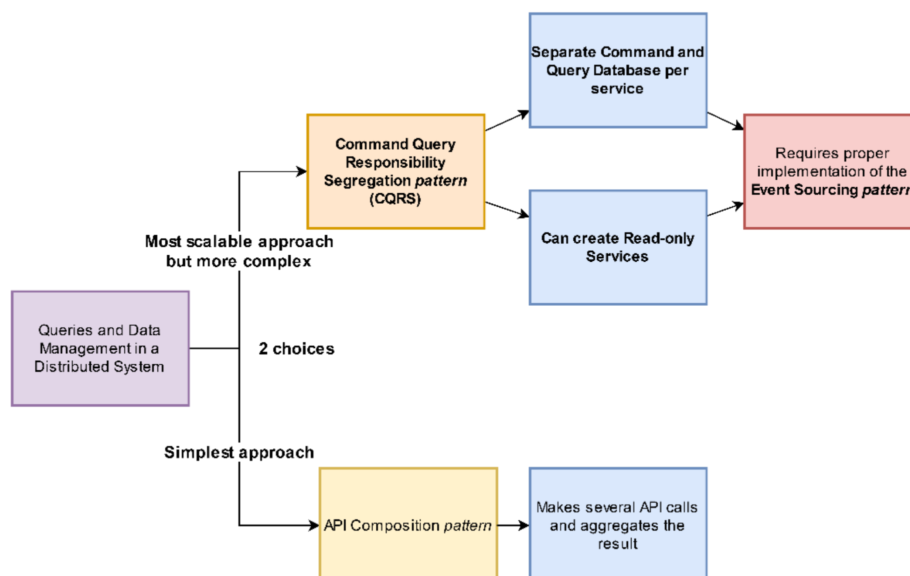

Figure 6-9. Decision tree for Querying and Data Management in a Distributed System

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

6.1.3. External APIs and Security

Querying services within the application space is different querying the system as an external client. Notably, the clients usually go through a firewall to access the service side of the application, a security system designed to monitors and filters incoming and outgoing network traffic (Boudriga, 2010). Using a firewall is a fundamental security addition for the planning system and is a given for many software systems.

The communication protocols that are used for these secure requests are often different from the communication protocols used within the application space. This can lead to unresponsiveness, especially if there are many communications between the external clients and the application space (Bass et al., 2003). Choosing how the external clients are going to access the application's information is thus an important design decision.

In context, the self-organizing logistics planning system's external clients would be the vehicle drivers with their handheld device, the carriers in their role of system operators, and the customers that want information about their parcel, that they can query through a website or smartphone app. Furthermore, the system would have to be set up so that customers receive notifications, be it via email or push notifications. Thus, there would be two mobile applications, one from the driver and one for the customers, and two web applications, one for the customers and one for the operators.

6.1.3.1. The API Composition Pattern for External Clients

Typically, web applications run inside the firewall and can thus use high-speed connections to interact with the services (Richardson, 2019). Conversely, the individual users that use browsers contact the web application from outside the firewall. These clients can communicate in two ways with the planning system: using webpage communication mechanisms with the web application, or a direct API call with the services. Other clients like the driver's mobile app would also use API calls to the services. I will consider future external clients like third party applications that could interact with the planning system's information, which could be other existing software on the carrier's side, for example. These would also communicate with API calls from outside the firewall.

By implementing the CQRS views for the parcel and vehicles, I can map the external communications of the system in the following manner:

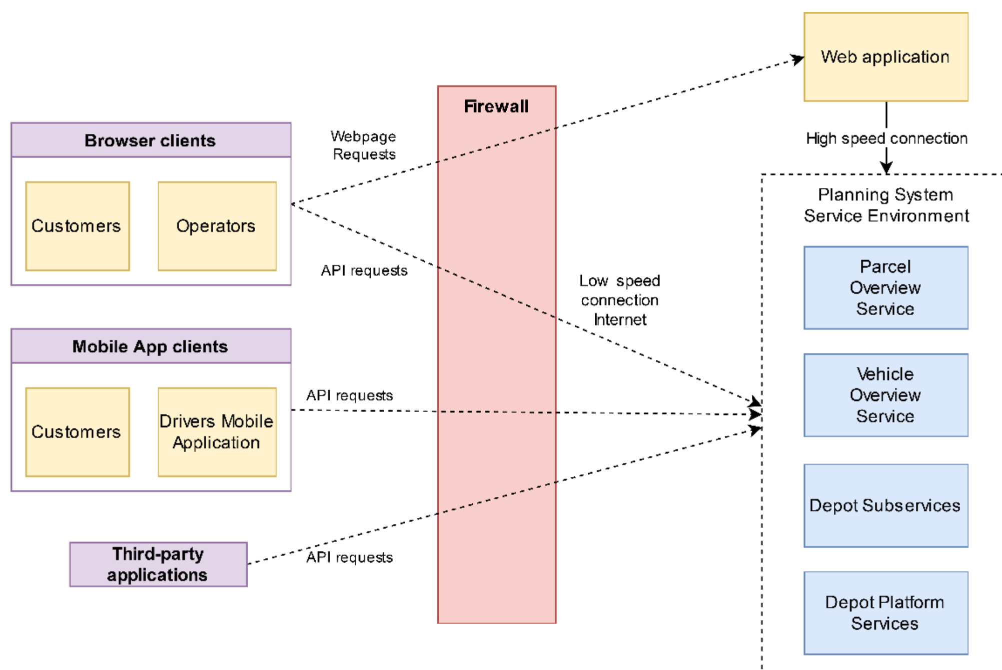

Figure 6-10. Direct API calls and Webpage requests between external users and the planning system

The first option for establishing queries and accessing the application is to apply the API composition *pattern* and have the external client be the API composer. In this scenario, the external client would query each service directly

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

and add all the results. For example, one of the mobile clients from Figure 6-10 could do a series of API requests to the vehicle, parcel, and depot services and then add the results.

6.1.3.2. The API Gateway pattern

While API composition would be the simplest approach, there are many drawbacks to this pattern in this context. First, different clients have different speeds because they are outside of the firewall. Therefore, the speed of information retrieval might be severely affected. This is a result of the common wireless connections mobile applications and third-party applications would have with the planning system. This leads to poor customer experience: if a mobile application queries each service directly, the chances that the application will be unresponsive is much higher. Furthermore, the developers of this mobile application would have to carefully write the API composition requests, which would distract them from improving the application's functionality.

The second option is to use the API Gateway *pattern*. API gateways are services that take care of API composition and other important security tasks like authentication and authorization. A separate service would be the API Gateway and would reside inside the firewall, therefore eliminating the latency issues discussed above. Figure 6-11 shows the application of the API Gateway pattern.

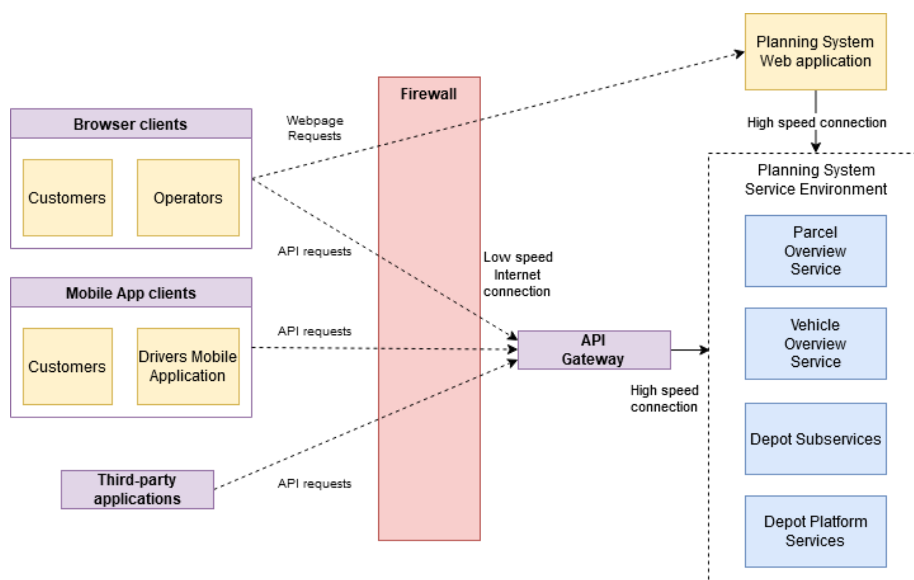

Figure 6-11. Applying the API Gateway pattern to the self-organizing logistics planning system.

These added tasks that the API gateway can perform are crucial in the context of the self-organized logistics planning system. Section 3.2.3 revealed that security is one of the indispensable requirements for carriers, and an API Gateway is a useful intermediary to verify the identity of the requesters and apply other security mechanisms. Security works differently in a distributed system than in a monolithic application. Monolithic applications leverage the use of shared databases to authenticate users, which is impossible to replicate in a microservices architecture (Richardson, 2019). Authenticating a user is proving the identity of a user. Thus, authentication is one of the cornerstones of software security: there should be a clear picture of who is accessing the system.

Common authentication systems like centralized logins can be difficult to include in a decentralized system without risking tight coupling. Therefore, I must look upon other ways of implementing authentication in the Self-Organizing Logistics Planning System.

The first and most direct option is to authenticate users on every service. This would mean that each service must correctly implement security measures at the local level. This adds a complexity layer to each service and also increases the risk of system vulnerabilities(Bass et al., 2003; Richardson, 2019; Zhu, 2005). The alternative is to apply certain security measures before the clients access the services. The API gateway is a prime place to apply these measures, and it can thus become a security corridor for the application on top of the firewall. To apply authentication at an API gateway, we can use the Access Token *pattern.*

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

6.1.3.3. The Access Token pattern

The access token pattern is an authentication strategy that can be implemented at the API Gateway. The clients that request access to the application via an API must present their credentials in their request. Then, an authentication intermediary verifies the identity of the request and creates a security token if the authentication was successful. Finally, the intermediary sends over the request along with the security token that was generated. Therefore, there is a different security token per each request.

The second type of client for the planning system would request access not through an API call but would first present their credentials with a login, like signing into an email account. Clients provide their ID and password, and the authenticator verifies the credentials and returns a security token. The client then saves this security token and includes it in all future requests. These future requests are API calls like the first scenario.

By using this pattern, authentication is centralized to the API Gateway, thus removing the need to implement security at the service level. Furthermore, centralizing authentication and other security measures also minimize the chances of having system vulnerabilities.

While Authentication is a key security measure in any system, it is insufficient on its own. The next step is to make sure only authorized clients access the planning system. This can also be done through the API Gateway.

6.1.3.4. Authorization options

The concept of authorization is to verify the validity of a client's credentials and to assign the client their due privileges and access to information (Fraser, 1997). Therefore, authorization works with a login mechanism similar to the second interaction we saw in the Access Token *pattern*. In essence, the clients present their credentials, and then the authorization client or login handler checks this data in a secure database. This secure database could be a separate authentication server or an internal database.

Modern authorization and security patterns are a field of its own, and there are many options at the implementation level. Current best practices include verifying credentials against a dedicated authorization server external to the service environment (See OAuth 2.0) and the usage of secret storage mechanisms (See The Vault Project by Hashicorp). These are offered as security frameworks applied as a package, usually as an Off-the-shelf solution. Other options include *Spring Security*, *Apache Shiro,* and *Passport.js*. The choice of authorization mechanism often depends on the language the application is coded in. However, the specific languages used to code the planning system are outside the scope of this thesis and I will thus leave these as options for the developers at the implementation level. Architecturally speaking, the space for these solutions is already created by implementing an API Gateway. I show these security patterns applied to the API Gateway in Figure 6-12:

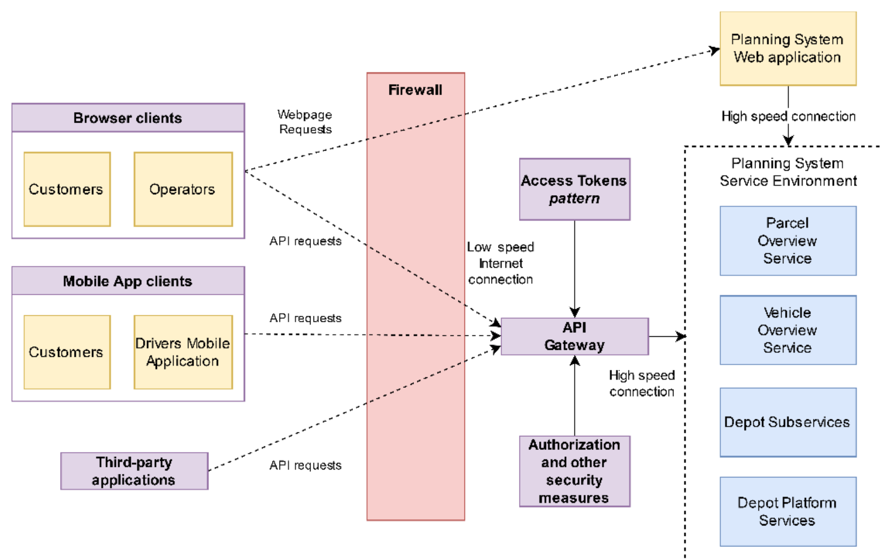

Figure 6-12. Using the API Gateway as a centralized security system.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

An important note on authorization for the Self-Organized Logistics Planning System is that the different users and their respective privileges. Thus, the mobile app of the delivery driver needs to have access to the system to send a notification in case of a failed delivery so that the algorithm can redirect the parcel to an alternative Collection and Delivery point. The mobile app should not, for example, have access to any of the other vehicle's information or parcels outside of the vehicle in question. Similarly, the carriers would be system administrators, but in a collaborative delivery scenario, they should only be able to access the data of the parcels and vehicles that they operate, not the data of any other rival carrier. I summarize each client's clearance in Table 6-1:

Table 6-1. Client Clearance

Client	Clearance
Browser Customer	Can only query about his/her parcel to the Parcel Overview service through the API Gateway/Web Application.
Mobile App Customer	Can only query about his/her parcel to the Parcel Overview service through the API Gateway.
Browser Administrator (Carrier)	Can only query about its vehicles and parcels through the API Gateway/Web Application. Can access its depots' information through the API Gateway.
Mobile App Driver (Carrier)	Can send notifications about the parcels in its vehicle, accessing through the API Gateway. Can query about the parcels held in that particular vehicle. Can access the Depots' Locations.

I summarize the External Clients patterns and options in Figure 6-13 and the Security patterns and options in Figure 6-14.

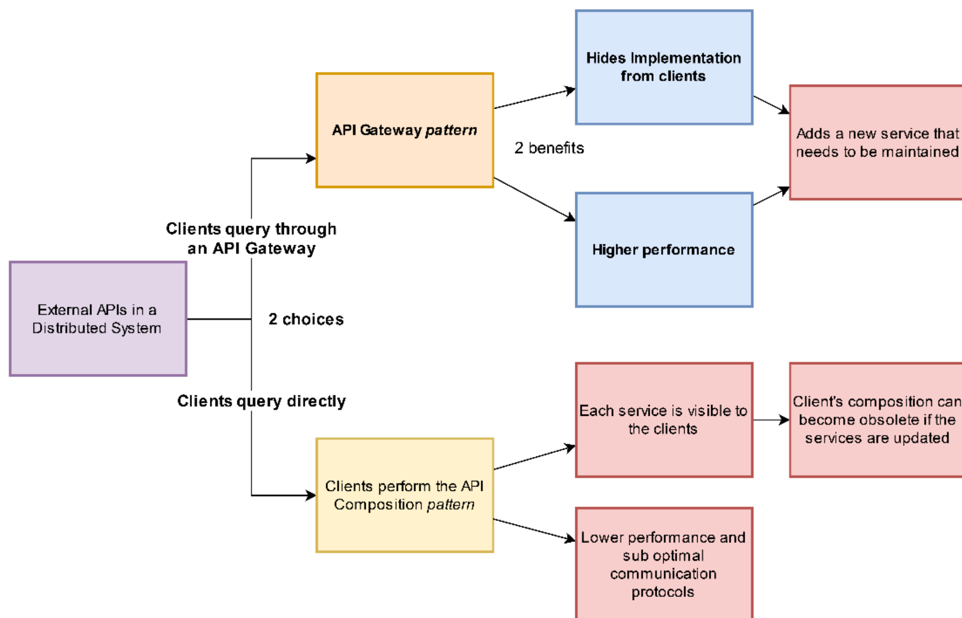

Figure 6-13. Decision Tree for External APIs in a Distributed System

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

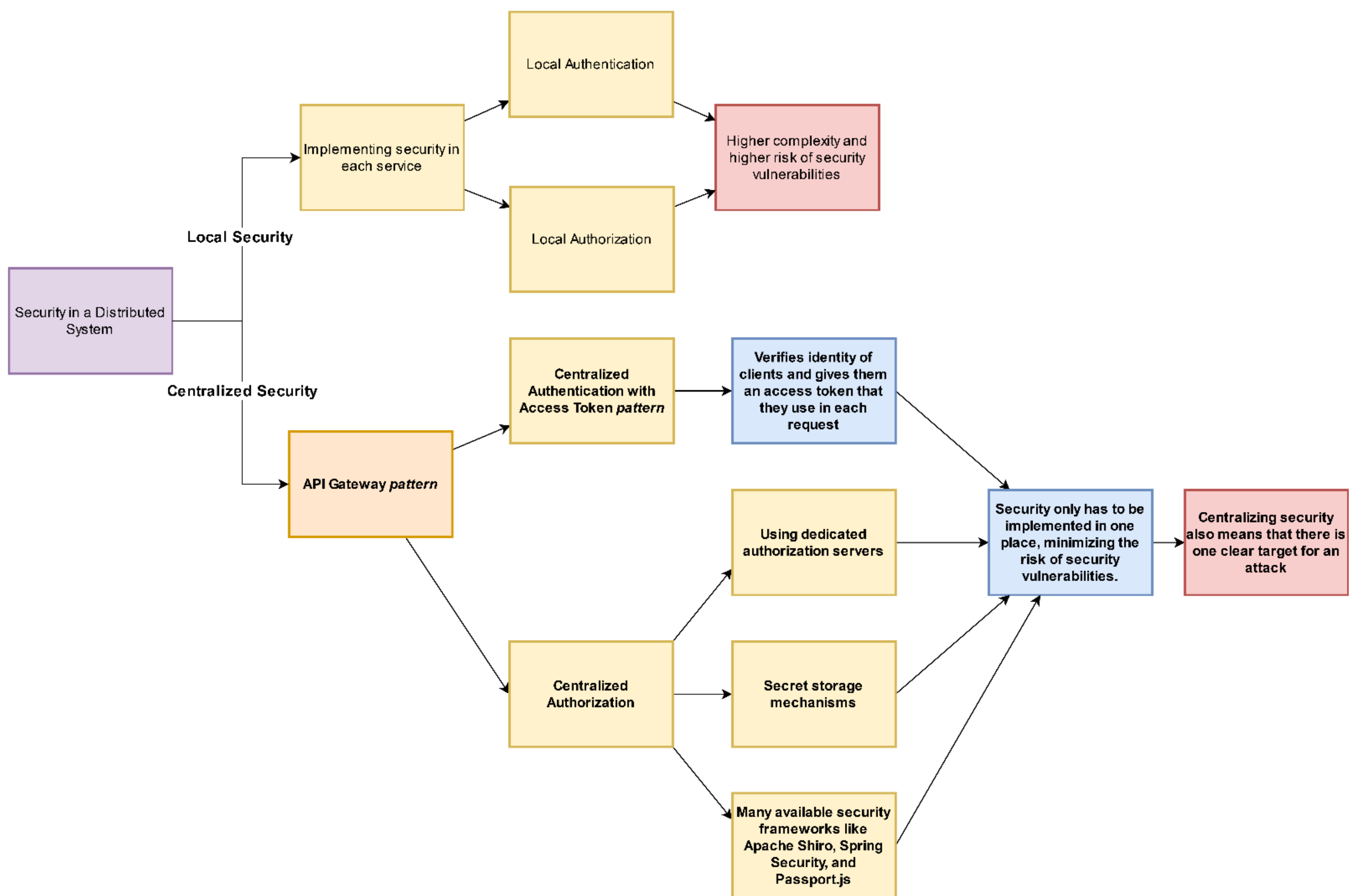

Figure 6-14. Decision Tree for Security in a Distributed System

The figures above show that both the external APIs and the security patterns depend on the API Gateway. In the former case, it allows for efficient querying while hiding implementation from the clients, and in the latter case, it allows the addition of a central security layer that simplifies development and reduces system vulnerabilities.

6.1.4. Designing for Testability and Modifiability

Developing software tests and modifying services are outside the scope of this thesis. However, the presented architecture should be considered *testable*, and *modifiable*. This implies that it should be feasible to develop tests for the software and to add functions to the software (Bass et al., 2003). I tackle this in this project primarily by offering a granular microservices architecture. In theory, the small size of the services, and clear responsibilities and operations for each service, testing and modifying each of the services should be vastly simpler than a monolithic version of the software. Furthermore, given the usage of message brokers and queues in the planning system, it is possible to add and remove functionality, thus resulting in a highly *modifiable* software. For example, carriers could create a service that processes parcel events and creates KPIs, and it could be easily implemented since the parcel has a CQRS view that allows easy access to this information.

Concerning testing, microservices behave differently from other pieces of software. This is because the business logic is often not found in one service, but a sum of services. This implies that communication protocols become the most important part of the software, not the business logic (Richardson, 2019). That is why the strategies for testing microservices are different than the strategies used to test applications in a monolithic or SOA environment. That is the reason why this thesis project offers a clear view of each service's operations and who they respond to. When presenting the software architecture, clear visual representations will show which services are connected to which. This will ease the development of tests that evaluate these communications.

An additional note is that sagas should be tested as an entity. They form the base of the business logic and they connect the services with each other. The literature suggests focusing on localized tests for each service instead of

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

complex end end-to-end tests that test the entire software at once (Richardson, 2019). Therefore, the tests for the self-organizing logistics planning system should begin by testing the individual operations from Table 5-2, and then moving on to the sagas that contain them. The size of the microservices architecture should make developing tests straightforward.

6.1.5. *Designing for Deployability*

Deployability is a key aspect for Prime Vision. While this project does not span software deployment, the designed software architecture should be *deployable*. Let us remember that deployability refers to the ease with which a software can be published onto the platform that hosts it. As defined in Section 2.3, a highly *deployable* system is a system that can perform its updates automatically, and does not burden current executions or require significant downtime between updates. Deployability is an attribute that is key during the development and implementation stage of a software product. Architecturally, certain strategies will support the deployability of the planning system. As we saw in Section 3.2.5, Prime Vision uses *continuous deployment*, and therefore high testability and deployability are important for the software developer.

There are three main strategies for deploying microservices: The language-specific package *pattern*, the Service as a Virtual Machine *pattern*, and the Service as a container *pattern*. I will discuss each of them briefly.

6.1.5.1. The language-specific package pattern

The most straightforward way of deploying a service is packaging it in a language-specific executable. An executable is a file that contains all the data of the service at hand. This means that all the details of a particular software are encapsulated into a single file. This is the way most monolithic applications are deployed. In a microservices environment, there would be an executable file per each service. After generating the executable file, developers must find a computer or platform that will run the code, thus deploying it. This computer, also called a *machine*, needs to be configured to be able to run the file.

This deployment *pattern* is simple to implement and it utilizes the resources of the *machines* efficiently (Richardson, 2019). However, several aspects of this *pattern* are incompatible with the Self-organizing logistics planning system. First, I decided to use microservices to increase the scalability and flexibility of the software. Since services have different types of responsibilities, they will likely be written in different programming languages. The selected deployment method needs to be flexible enough to handle this type of variety. While the software that executes language-specific packages is efficient, the load balancing is done manually. Load balancing refers to the distribution of computer power across machines. The fact that load balancing is manual with this *pattern* means that developers need to assign the correct number of packages to each machine to manage the efficient usage of computer resources. Other modern platforms like container frameworks have automatic load balancing. Furthermore, language-specific packages can perform these activities in theory, but each *machine* needs a specific software version for each language (Richardson, 2019). Therefore, if I use this pattern, I risk increasing the complexity of deployment significantly.

6.1.5.2. The service as a virtual machine pattern

The next option is running each service as a virtual machine (VM), used in cloud computing. A virtual machine actually is a special type of image, like an Amazon Machine Image (AMI). These images follow the same idea of packaging a service in an executable file, but they also include the software necessary to execute the service. Therefore, the part of the software that deploys the service, called the deployment pipeline, needs a Virtual Machine Image builder (Richardson, 2019). After the virtual machine image is created, it can be deployed to several machines that do not any special software because each image includes the necessary software automatically (Amazon Elastic Compute Cloud, 2020). These machines are usually in cloud platforms like Amazon Web Services (AWS). There are several benefits to cloud computing and Virtual machines specifically. The VM platform allows load balancers that distribute requests across deployed instances of the service. Each instance is a copy of a service. If there are many instances of the same service, then these are running in parallel. This is one of the most used ways of scaling a system. The deployment of a service using the Virtual Machine pattern is shown in Figure 6-15:

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

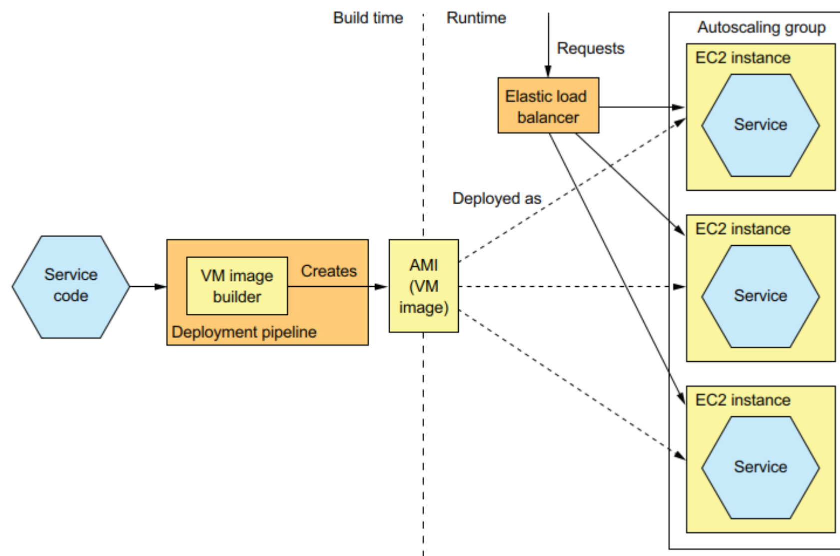

Figure 6-15. The service as a Virtual Machine pattern. Taken from Richardson's Microservices Patterns (2019, p. 391)

Figure 6-15 shows that in the Service as a Virtual Machine pattern, each service has a deployment pipeline that includes a VM image builder. This image is what is deployed to the machine on the right side. When requests come into the system, they first go through a load balancer that distributes them between the instances of the service. Virtual Machines are a great solution for deploying microservices. The cloud computing platforms like AWS are mature technologies and thus are stable and reliable. However, building and processing these VM images takes considerable amounts of time. Their relative weight also makes VM images consume a lot of bandwidth. Thus, the deployment system has some added overhead which affects performance.

The last deployment alternative I will consider is the Services as a Container *pattern*.

6.1.5.3. The service as a container pattern

Containers are the most modern deployment mechanism. Containers work like Virtual machines, as both package the service and the necessary software into an image. Containers, however, include the complete toolset necessary to run the service, even the operating system (Docker, 2020a). Each container thus becomes an isolated *machine*, where services can even have their own isolated IP addresses (Richardson, 2019). Furthermore, containers are lighter than Virtual Machine images, improving performance. The comparison between Virtual machines and containers is similar to the comparison between lightweight messaging protocols used in microservices and heavyweight communication protocols used in SOA. Thus, the lightweight nature of containers allows the system to become more scalable because the software can handle more deployments.

Both Virtual Machine Images and Docker containers would be great solutions to deploy the planning system's distributed software. The main difference between them would be performance and the providers of the cloud services that host the images. Containers have an edge with performance and speed of deployment. Being the latest technology with a growing open source community (Docker, 2020b), containers should be prioritized over Virtual Machines. Prime Vision would have to evaluate which solution would work best given their current providers. Only after testing and deploying the software is it possible to determine whether the performance is acceptable. If the performance is too low with Virtual Machine Images, then the development team should migrate to containers if possible. I summarize these design decisions in the decision tree of Figure 6-16.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

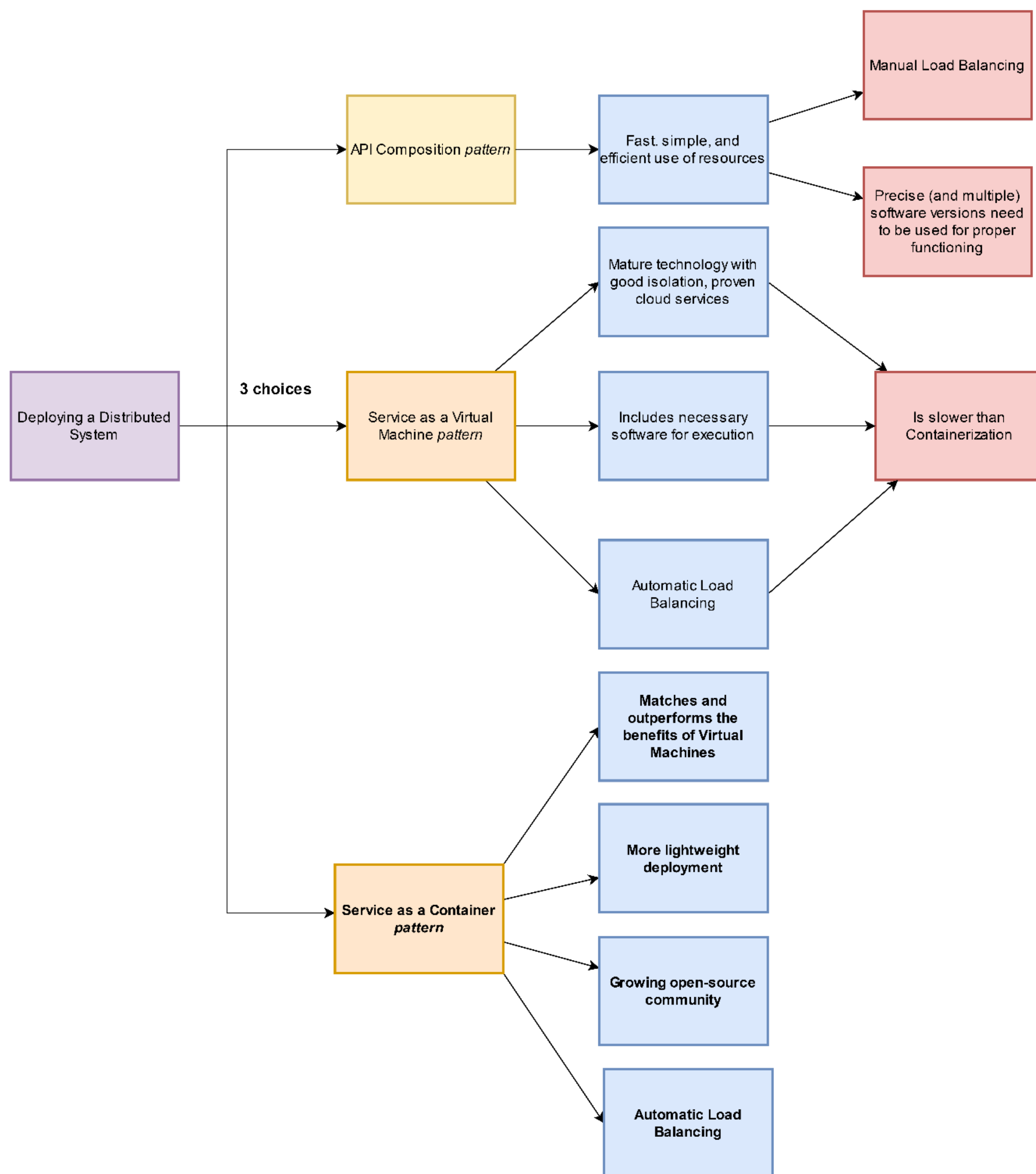

Figure 6-16. Decision Tree for Deployability patterns in a Distributed System

6.1.6. Designing for Scalability

Section 3.2.5 unveiled that one of the main concerns for Prime Vision is for the system to be highly *scalable*. Scalability refers to the capacity of a software system to increase the workload without impairing performance (Bass et al., 2003). There are several ways in which the current design aids the scalability of the self-organizing logistics planning system. First, the granular nature of the microservices proposed in this thesis project allows a clear view of the computational requirements of each element in the development phase. This will ease the employment of several testing, deploying, and scalability strategies.

Furthermore, I recommended deploying services as containers or Virtual Machine Images in Section 6.1.5. This simplifies scaling the system up because the inherent architecture of these solutions allows the easy employment of load balancers, which distribute the workload of the processors. Maintaining several instances of each service allows duplicating capacity, and this is eased by these deployment strategies as well.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

The scalability of distributed systems is an area of interest because many new applications employ these architectures. Future-proofing algorithms using Machine Learning tactics has been considered valuable for a long time, with examples in distributed systems as early as 1991 (See Brazdil et al., 1991). Today, the field is still in expansion, and several innovative solutions have been developed that exploit the concepts of Machine Learning on distributed systems, like TensorFlow and H20.ai (Abadi et al., 2016; Chen et al., 2015).

I believe these innovations can be included in the Self-organizing logistics planning system. The original algorithm was written in Python, and several Machine Learning tools can be applied for these types of algorithms, like Confluent's Apache Kafka (Confluent, 2020a). For example, in Kai Waehner's Digital Twins for a car network, a TensorFlow instance is used to train a python model, and this aggregate receives its information from an event-streaming platform like Kafka Cluster (Waehner, 2019). I believe this same concept could be applied to the Self-Organizing Logistics Planning System, as we can exploit the event-driven nature of the system to feed a Machine learning module that helps improve the Python algorithms employed in the system. I expand on this topic in Chapter 8.

Machine learning applications thus improve key algorithms. These could be applied in the key computational modules of the Self-Organizing Logistics Planning System, like the vehicle's Bid Creator and the Parcel's Vehicle Selection. These services are at the heart of the auctioning process and their performance will thus be essential in the success of the Planning System. Therefore, I recommend applying Machine Learning tactics to these two services. Naturally, it is impossible to know with certainty if these are the elements that require the most processing power until the services are designed, but the software architecture should have space to include these Machine Learning modules on these and other services.

*6.2. **Synthesizing the Application of Architectural Patterns***

The previous Section compiled the selection of the main architectural patterns. I will now illustrate the architecture of several key components of the system. Then, I will present another representation of the whole environment of the system, which would be similar to the Class Diagram I introduced in Section 0, but with the assigned services and some of the features key to the patterns we selected in this chapter. The sum of these diagrams composes the Draft Software Architecture for the Self-Organizing Logistics Planning System.

The system will exhibit two CQRS views as mentioned in Section 6.1.1.4. These would form two new services: The Parcel Overview Service, and the Vehicle Overview Service. These services are visible in Figure 6-17.

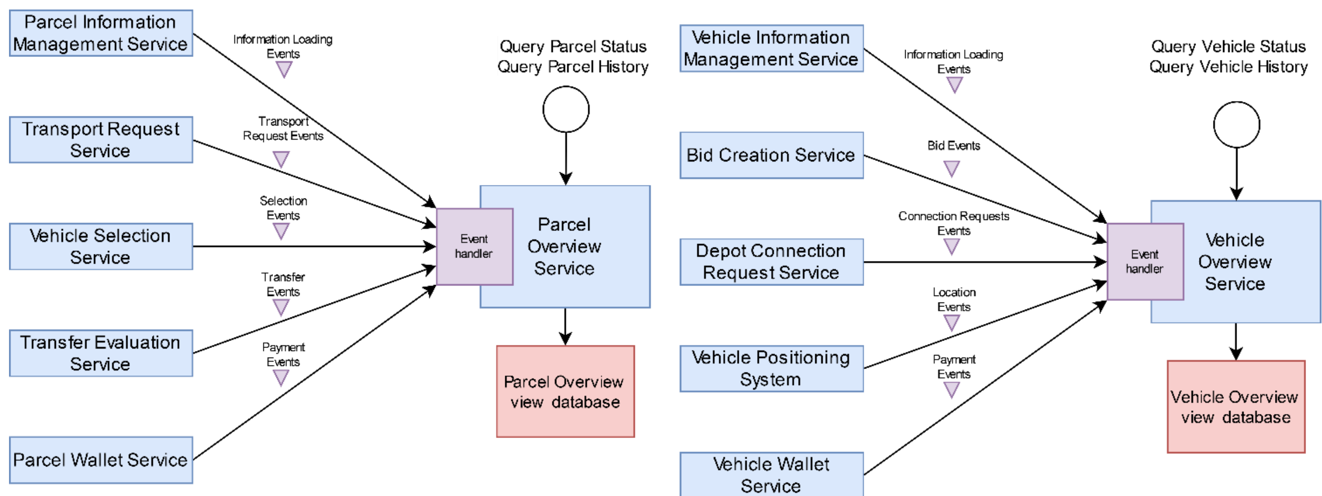

Figure 6-17. CQRS views for the Parcel and Vehicle in the Self-Organizing Logistics Planning System

These services' only function is to be subscribed to the individual services that comprise each digital twin and copy these in a separate database that is easy to query. Because of this functionality, key system functions like querying the delivery status of the parcel or the payment status of a particular bid can be done easily, reliably, and without requiring extra processing power or request/response mechanisms for queries.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

These CQRS views are key to communication with external clients. These CQRS views are the ones that would be contacted by the API Gateway, along with the services that comprise the depot digital twins and the depot platform, as mentioned in Section 6.1.3. Furthermore, the API Gateway is a prime location for implementing several security measures like authentication and authorization, as applied in Figure 6-12. Furthermore, implementing an API gateway reduces the number of requests going through the firewall and thus increases performance.

I will leave out both the CQRS views and the external API connections from the updated representation of the system. This is to avoid cluttering the view from the main features of the system. We can see the updated system in Figure 6-18.

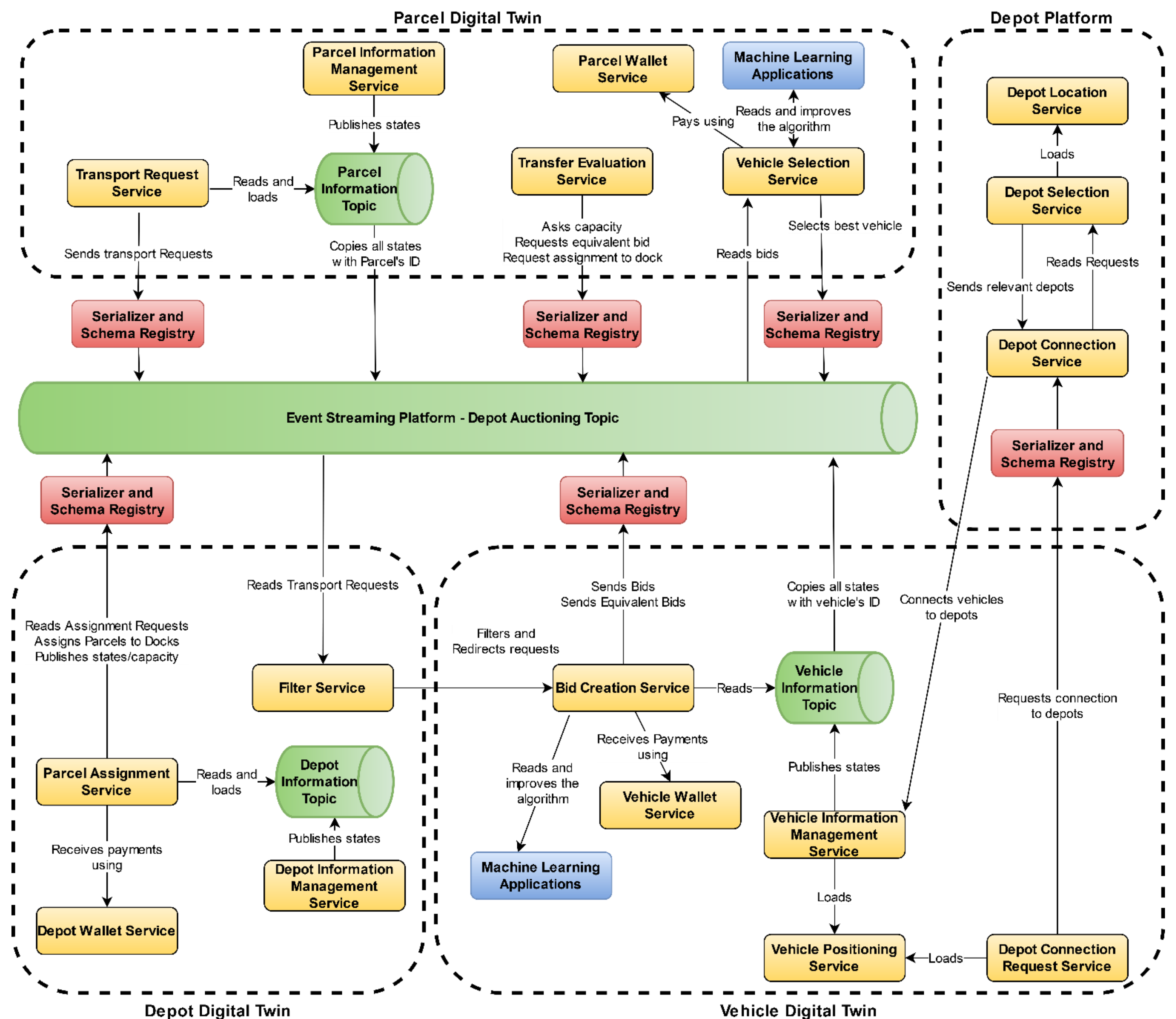

Figure 6-18. Conceptual Design of the Self-Organizing Logistics Planning System's Software Architecture

This is the conceptual design of the Self-Organizing Logistics Planning System that results from the decomposition of the system through Business Capabilities, and the application of architectural patterns for resolving the issues that arise from the distributed nature of the system. This software architecture shows us the most important dependencies of the system, such as:

- The Transport Request Service gets its data from the Parcel Information Topic, which is updated by the Parcel Information Management Service. Then the Transport Request Service creates the request, the

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

request is translated by the Serializer and the Schema Registry. After the request is translated, it is published on the Depot Auctioning Topic, where vehicles that are subscribed can react to them.

- The Transport Requests are read by the Filter Service, which redirects the Transport Request to the relevant vehicles. The vehicle Bid Creation Service then creates a bid, using information from the Vehicle Information Topic that is updated by the Vehicle Information Management Service. The Vehicle Information Management Service in turn reads the vehicle's location, necessary for the creation of the bid.
- When the parcel is in a vehicle and that vehicle is connected to the Depot Auctioning Topic, then the parcel can evaluate transfers. For that, the Transfer Evaluation Service asks the depot for its capacity. If the answer is yes, then it creates a transport request that is published on the Auctioning Topic, where other vehicles read it. If there is a vehicle that wins the parcel's bid, then the same Transfer Evaluation Service requests assignment to an (off)loading dock, which the depot can read.
- The Depot Assignment Service reads the Assignment Requests published by the Transfer Evaluation Service and assigns the parcel using the information from the Depot Information Topic, which is updated by the Depot Information Management Service. The depot can also receive payments using the Depot Wallet Service.
- The Vehicle's Depot Connection Request Service communicates with the Depot Platform through a Serializer/Schema Registry. The depot platform's Vehicle Connection Service gathers its information from the Depot Selection Service who in turn needs the Depot Location Service.
- Both the Bid Creation Service and the Vehicle Selection Service have a connection to potential Machine Learning Applications.
- The architecture is highly modular. Each service can be tuned, and some services can even be removed. For example, the Filter Service helps find vehicles that would be interested in creating a bid for a transport request. This saves computational time by eliminating vehicles that deliver far from the parcel's destination. However, this function is in the algorithm to improve the performance of the system, and this software architecture allows this service to be implemented or not. In other words, this service could be removed if performance is sufficient and further delivery options want to be explored in the implementation phase.

The Conceptual Design of the system finishes Phase 2 of Dym and Little's Systems Engineering Design Methodology (2014).

6.3. ***Summary***

In this chapter, I discussed the main obstacles that a distributed system faces. Then, I discussed and compared architectural *patterns* that help solve these obstacles, summarizing the design decisions with Decision Trees at each topic. The topics of interest were communication, transaction logic, querying the system, security, testability, modifiability, deployability, and scalability. The result was an event-based system that uses the local transactions of each service as updates that trigger the chain of operations. Thus, the system uses event sourcing and sagas for transactions. Querying was solved using a combination of two patterns, the most notable being the Command Query Responsibility Segregation *pattern*. This pattern was applied to the parcels and vehicles, easing querying for the system without adding processing requirements.

Then, I tackled Security and querying by external clients by applying an API Gateway where security can be centralized. Testability and modifiability were only tackled with the granularity of the system, and I discussed some patterns to ease deployment on the implementation phase. I recommended the use of containers for deploying services, a state-of-the-art deployment strategy that also helps scalability and isolation. Furthermore, I discussed the possibility of adding Machine Learning modules for enhancing the algorithms on key services, such as the vehicle's Bid Creation Service and the parcel's Vehicle Selection Service.

Finally, I synthesized the design into the system's Draft Software Architecture in Figure 6-18 and explained its main features. The development of this Software Architecture ends the conceptual Phase of Dym and Little's

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Systems Engineering Design Methodology (2014). In the following Chapter, I discuss the result of the expert panels that validate this design, comprising Dym and Little's Preliminary Design Phase.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

7. Design Verification

Chapters 5 and 6 constituted Phase 2 of Dym and Little's Systems Engineering Design Methodology. This chapter concerns the verification of the Software Architecture developed in the previous two chapters, thus belonging to Phase 3: Preliminary Design. The main research question that will be tackled in this Chapter is:

8. *Would the conceptual design comply with the customer requirements?*

The main methodology to perform this verification is a series of Expert Panels, that were held bi-weekly from May 2020 to July 2020. The internal validity of this verification is supported by the fact that a system's Software Architecture allows us to infer the Quality Attributes of said system (Bass et al., 2003; Richardson, 2019; Zhu, 2005). In the expert panels, the draft software architecture was discussed thoroughly, and some changes were suggested.

The Research Flow Diagram of Figure 7-1 summarizes the main methodologies and research question of this Chapter.

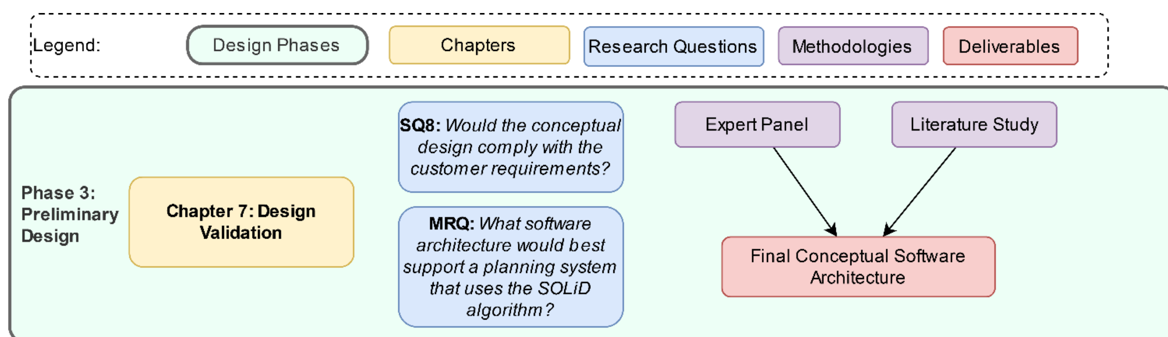

Figure 7-1. Phase 3, Preliminary Design, adapted from Systems Engineering Design methodology by Dym et al. (2014)

7.1. The Expert Panels

Work with developers began in May 2020 with bi-weekly discussions about planning systems and the SOLiD algorithm. These discussions were mostly held online and used supporting software to share screens, develop sketches, and record the meetings. Other meetings were held in person, using blackboards and brainstorming sessions. The developers that comprised this Expert Panel were selected with my supervisor, Bernd van Dijk, who is the Director of the Innovations Department at Prime Vision at the time of writing. The selection criterion was based on the expertise and availability of the developers. This was a careful selection of experts from a pool of about 30 software specialists. From the initial definition of this expert panel, the intention was that these workers would eventually be the developers of the logistics planning system in the implementation phase.

The expert panel was comprised of two software developers from the Innovation department that had been focusing on robotics in the last two years from the time of writing, a Software Architect who worked in the sorting system currently used for first and middle-mile delivery, and one software developer from the Internet of Things team. All of these developers have been working in Prime Vision for more than 3 years. In conclusion, the expert panel comprises experts in computer science, robotics, parcel logistics, and distributed systems.

Part of these interactions with developers led to the Questionnaire that helped refine the Customer Requirements for a logistics planning system, discussed in Chapter 3. Given the time constraints of the rest of the workers at Prime Vision, the questionnaire was only administered to these 4 developers. With this questionnaire, I discovered that only the developer from the IoT team had worked with microservices in a large-scale project, two had only tested them in small personal projects, and the last developer had only watched a webinar about microservices but had no direct development experience with them. I also noticed that even when the developers said they had experience with microservices, the terminology they and I used were different. For example, they were not familiar with concepts like patterns, *sagas*, and Command Query Responsibility Segregation. Therefore, Chapter 6 was

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

instructive for them as well. While they were not as familiar with the terminology, after an explanation of their functioning, they agreed with the pattern's way of solving most problems for the Logistics Planning System. I hypothesize that since software development is an ever-changing discipline, and information comes from many different sources, such as conferences, forums, and open-source communities, developers often have a different vocabulary depending on where they gather their knowledge from.

I hereby discuss the feedback gathered from the discussions with the expert panel.

7.1.1. Transactions in a Distributed System

At first glance, the decomposition of the system by its operations and competencies was validated as a good decomposition method. The experts believed that this separation of the software by business responsibilities led to a satisfactory decomposition of the system, as a functionality-first approach was in their opinion the best way to ensure services become isolated. However, the developers advised me to be careful with the compliance with the isolating one service from the other. In the experience of the developer from the IoT department, when two services have many communications with each other, the typical response is to group them in one service. This has implications at the software level, but also at the organizational level because the two teams that originally developed the two services typically merge for the composite system. In his experience, this meant slower, less agile teams with objectives that were less clear than the original smaller teams. This slows down production and development. Therefore, the developers advised me to be thorough with the separation of services based on the operations of the system and to check which services had several back-and-forth communications.

One of the first concerns was the lack of traditional ACID transactions. As the developers have not worked with microservices extensively, the idea of having an eventually consistent system was unorthodox. ACID transactions are the transactions that usually happen inside monolithic services, and are characterized by being isolated and consistent on all levels, can be recovered in case of a crash, and happen concurrently, so that no two actors are working on the same database at the same time, for example.

In the proposed software architecture, transactions indeed lack some of the qualities of ACID transactions. This occurs because many of the interactions and the data for external clients are *eventually consistent*, a typical characteristic of a system that uses asynchronous messaging. This means that it takes some time before all the databases read the latest events from the topics and have updated information. However, some key features of ACID these transactions were guaranteed. For example, states and messages are recoverable by the usage of topics that buffered and saved information. Furthermore, since topics buffer messages for orderly consumption, consistency, and timeliness of transactions is also supported. The experts validated the emphasis on replacing request/response systems by request/asynchronous response systems with order queues, as this would better serve the high availability requirements. The high availability customer requirements were thus hereby verified. However, the team had no experience with sagas, so the careful development and testing of sagas will be a crucial matter in the implementation phase. This will be necessary to comply with the functional requirement of having orchestrated transactions and will not be able to be tested until the implementation phase, an important limitation to this project.

7.1.2. Specific changes to Services

Transactions of course span the whole system, but there was one area of interest in which the development team believed it was better to join two separate services. These were the Depot Information Management System and the Parcel Assignment Service. The developers argued that having a separate service for updating the depot and assigning parcels to loading docks could be troublesome. The reason is that if the assignment and the depot update happen asynchronously, there is a risk of the depot update not happening before the parcel assignment service accepts and assigns another parcel. This is the risk given by the lack of Isolation in distributed transactions. By joining the Depot Information Service and the Parcel Assignment Service into one service, this problem can be avoided. This is an example were excessive granularity can be harmful to the system, and thus, the decomposition was revisited and there were only two services for the depot: The Depot Information Services, who responds to Capacity checks, assigns parcels to loading docks, and updates the depot's capacities, and the Parcel Wallet Service, who handles billing and payments.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Developers also mentioned that the architecture should show certain communication details. For example, the Parcel's Vehicle Selection Service and the Vehicle's Bid Creation Service communicate their data in batches to the Machine Learning modules, and this should be visible on the architecture.

7.1.3. Addition of other functions and services

The developers also took notice that the Mobile App for the vehicle driver ought to be developed, and therefore, there should be a mapping entity connected to this entity on the External APIs view. While the architecture of such a mobile application is out of the scope of this thesis project, I added a mapping service to the external APIs view that the Driver's Mobile Application sends API requests to.

Monetization was another point of interest for Prime Vision's management and development team. I brought this topic to the Commercial Director at Prime Vision (See Appendix D), and it was determined that a service-oriented business model was desirable for Prime Vision. Given this preference, I suggested a usage-based billing system that took advantage of the local transactions and thus bills in terms of parcels delivered, auctions won, or other usage metrics. Both the development team and the commercial director agreed that this would be the best monetization strategy for the Self-Organizing Logistics Planning System (Interviewee C, personal communication, July 31, 2020).

Architecturally, this billing system can be modeled as an extra service that gets information from the CQRS views of the parcel and vehicle and sums it, or it could be localized per depot, feeding off of the Depot's Auctioning Topic. I chose the former, as it requires fewer connections to the inner workings of the algorithm. This Billing Service would be inside the firewall like the Planning System's Web Application. This service is visible in the final External APIs view at the end of this chapter.

Lastly, some unrelated projects wanted to be added to the Self-Organizing Logistics Planning System. For example, Prime Vision is working on an augmented reality tool that helps get a parcel's dimensions with a camera. While this is not yet developed, the software architecture shown in Chapter 6 leaves room for these added services. Architecturally, this could be connected to the Parcel Information Management Service, and it could publish the parcel's dimensions to the Parcel Information Topic. The system would work the same, as the Transport Request Service would still load information from this very same topic. These and other functions are easily applicable to the suggested architecture, as other services can use the system's events and messages to add more functions. For future services, the development team must decide whether to keep it inside or outside of the firewall and should document this accordingly.

7.1.4. Information Access and Filtering

Another concern was the transparency of data to the carriers, which will ultimately be the users of the system. In other words, it was necessary to review the access carriers had, which can be applied with the authorization mechanisms described in Section 6.1.3. Using the API Gateway not only as a security system but also as a filtering mechanism, certain data could be hidden from users that are not meant to access it. There are two reasons for these remarks. The first is the intellectual property of the company: Since the algorithm's functions are the property of the company, the implementation details should be hidden from the Carriers. Thankfully, this is solved by the CQRS views defined in Section 6.2. The second motivator to adding filtering to the service is the protection of Carrier data to all other Carriers participating in the system. In a collaborative delivery scenario, the data from rival carrier's bids and parcels should not be retrievable by other carriers.

The architecture per se was not changed by this discussion, but the data restrictions will be tightened in the implementation phase.

7.2. ***Non-Architectural Requirements***

As defined in Section 2.5, certain functional requirements are not architectural, and thus cannot be attained by architecture alone. This is the case for some requirements gathered in Chapter 3, like minimization of costs, increasing capacity utilization, robustness against traffic jams, and accurate prediction of Estimated Time of Arrival for parcels. Thymo Vlot's algorithm covers the rest of the functional requirements listed, such as minimization of costs, increasing capacity utilization, and robustness against traffic jams. However, Vlot's algorithm is inherently

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

incompatible with an accurate prediction of Estimated Time of Arrival for a parcel, as the self-organized system that changes and re-evaluates routes many times per day. Therefore, this particular functional requirement cannot be fulfilled by the system as is. This was also mentioned by Vlot as an important limitation of the system (Vlot, 2019, p. 96). These types of requirements depend solely on the algorithms implemented and are thus outside of the scope of this project. In these cases, the software architecture should only avoid obstructing the solution to these problems, which is established by software architecture that was verified by the developers.

7.3. *Defining the final Software Architecture*

Outside of the points risen in Section 7.1, the team believed that the system was well designed and that it would cope with the strong requirements in scalability and availability, thus verifying the compliance with customer requirements.

Applying the changes that resolve the development team's concerns leads to the Final Software Architecture of the system. I will show three main views: The CQRS views for the parcels and the vehicles, which remain unchanged from Chapter 6, the External APIs view, and the main system. For clarity, I exclude the CQRS views from the main system's view in Figure 7-5.

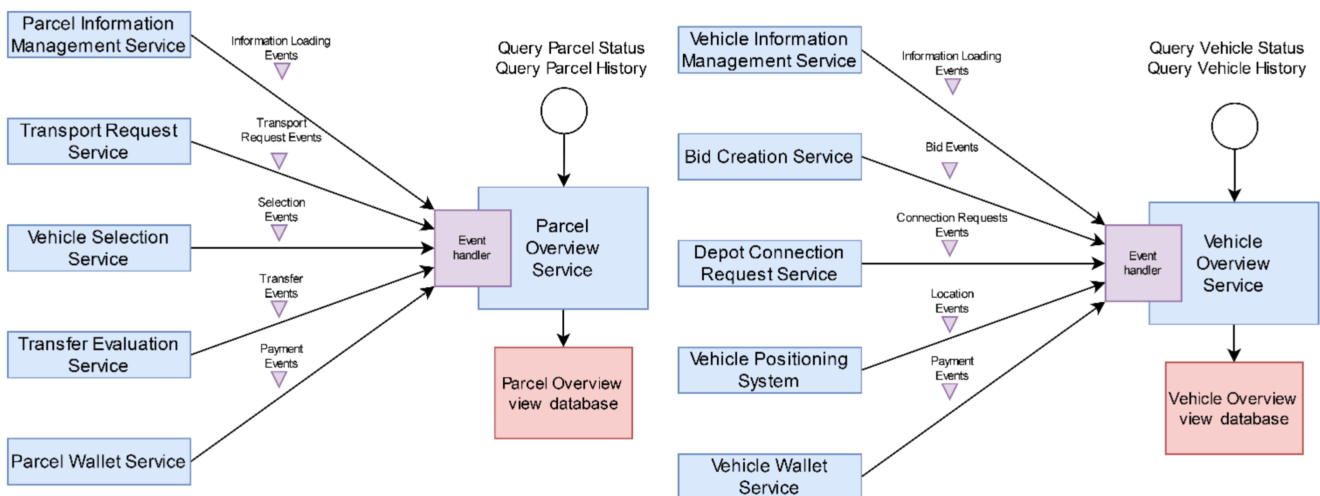

Figure 7-2. CQRS views for the Parcel and Vehicle in the Self-Organizing Logistics Planning System

These CQRS views will help the distributed system handle queries efficiently, and it can also simplify the communication with external clients that do not need to see the implementation of the algorithm. These CQRS views are holistic to the parcels and vehicles. Further CQRS views that are subscribed to fewer services can be developed, but this should be done in the implementation phase of the project. Inside of each of these components that connect to the Parcel and Vehicle Overview Services are two databases, as shown by the example of Figure 7-3:

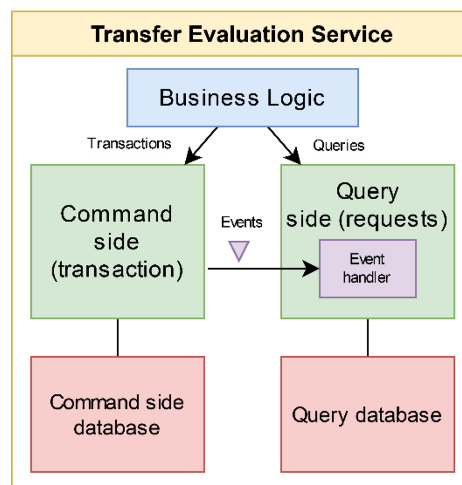

Figure 7-3. CQRS view of the Transfer Evaluation Service.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

The command side database is where the transactions occur, and the query side database is where the transactions are copied. The Parcel and Vehicle Overview Services of Figure 7-2 are query-only services that would be subscribed to these query databases.

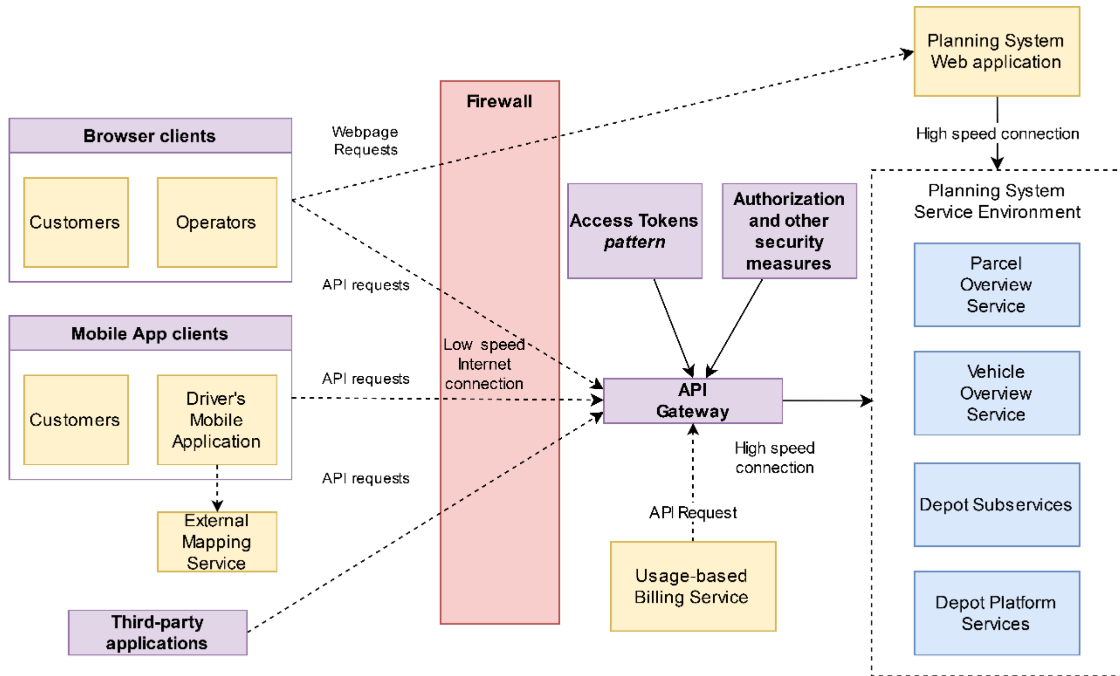

Figure 7-4. Final External APIs view for the Self-Organizing Logistics Planning System

As we see in Figure 7-4, the Billing Service was added as an extra service that can send API requests to the API gateway, which can authorize the service, and retrieve the necessary data for it to carry out its functions. Furthermore, the Mapping Service mentioned in Section 7.1.3 is visible on the external client-side of the application. Under this architecture, the Driver's Mobile Application would request information to the API gateway and then send a request to the Mapping Service, which would send map routes to the Driver's Mobile Application. This particular mapping service could be put inside the firewall; however, I will leave that decision to the development team in the implementation phase. The algorithm and software architecture of these added services are outside of the scope of this thesis project.

The design also showcases a clear entry point for security mechanisms like Access token and other authorization methods in the API Gateway, and the CQRS views of key services simplify queries in the system.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

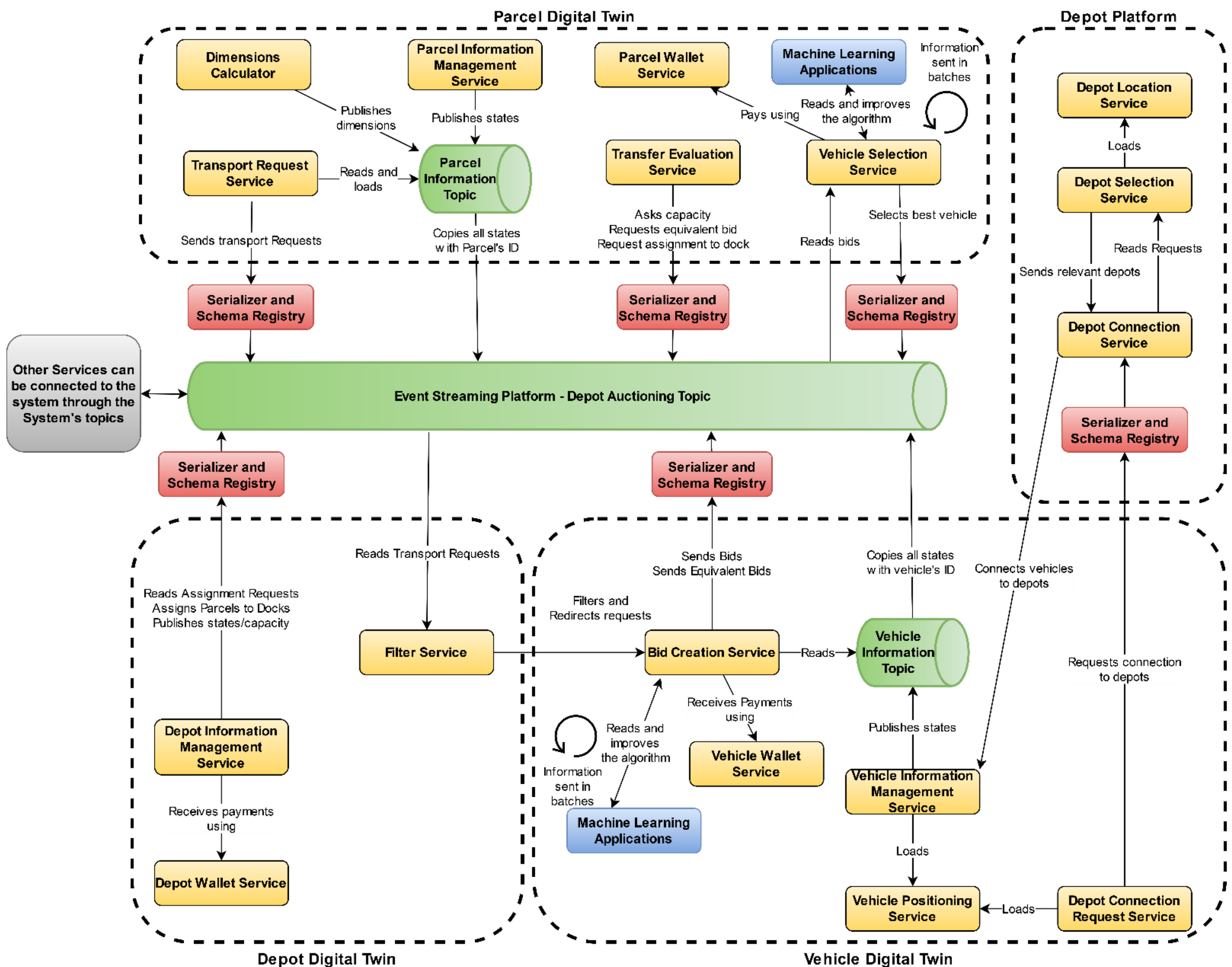

Figure 7-5. Final Conceptual View for the Self-Organizing Logistics Planning System

The final conceptual view for the Self-Organizing Logistics Planning System showcases high modularity and scalability. Extra services can be added and utilize the messages broadcasted in the topics in the system, so further projects can therefore be combined into the planning system. Furthermore, certain services are optional for the system. For example, the Filter Service could be deactivated to test auctioning with more vehicles, and the Machine Learning Applications are optional and could be implemented later into the lifecycle of the project. This is because Machine Learning offers scalability through optimization, but it does not concern crucial functionality. As requested by the developers, the final design of the software architecture also adds an indicator for batch information for these applications.

The modularity of this architecture eases the development of the system. Each of the services in this architecture could be developed by a small team of developers. The architecture clearly shows the relationships between the services, and thus there is a clear view of how tests can be developed. Hypothetically, each of the operations can be tested separately, a feat only achievable by a granular separation in microservices.

These views form a clear image of how the system would function and it helps developers infer the qualities of the system. This is the main mechanism through which this Software Architecture was validated by the experts.

This architecture helps stakeholders communicate about the system's characteristics, and it narrows the development options to the main design decisions visible in the architecture. One of the greatest benefits of this architecture is that it can be easily adapted for different projects. Being a highly distributed system with a clear software architecture, the mechanisms implemented in this project can be applied in many other scenarios, not

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

only that of parcels and vehicles. This gives this design enormous financial and strategic potential. Furthermore, Chapters 5 and 6 clearly explain the design process and the rejected alternatives, so the final design has solid conceptual integrity, and the research becomes more replicable, increasing the internal validity of this project.

7.4. *Summary*

In this chapter, I verified the design through Expert Panels. The research question that I sought to answer was whether the designed software architecture would comply with the customer requirements set in Chapters 3 and 4. The expert panel raised several concerns that were subsequently addressed in the Final Software Architecture. Having addressed these points, we can conclude that the design complies with the customer requirements set in Chapters 3 and 4. Therefore, we have answered research sub-question 8 and the main research question of this thesis project. However, compliance with customer requirements can only be validated through the expert panel since it is impossible to test the system without having developed the software. Thankfully, software architecture helps infer the characteristics of the system, a crucial point in verifying the project. However, the lack of testable software still is an important limitation of this thesis project.

The Final Software Architecture for the Self-Organizing Logistics Planning System includes four views: The first two concern the application of the Command Query Responsibility *pattern*. The first describes the query-only services that gather the data of the parcels and vehicles. The second view is an example of the internal components of one of the services that would provide data for query-only services. The third view concerns the External APIs, which shows how external clients communicate with the system. This view also shows how security is implemented in the system. The final view is the conceptual view of the system, showing most of the services in the system. This view achieves several objectives. First, it shows the main functioning of the system in a clear form that is understandable to most stakeholders. Secondly, it helps the incumbent company set up goals and timelines for the development of the system. This system also shows the functioning of a highly distributed system, which could become the basis for other software products that would also benefit from a distributed architecture. This gives this project enormous financial and strategic potential.

The Final Software Architecture presented in this Chapter shows the most fundamental design decisions of the project. These decisions restrict the system, which in turn simplifies the development phase by reducing the design and system complexity. Furthermore, this project also offers a clear exploration of how and why these design decisions were made, giving a clear idea of the architectural patterns that were rejected. This grants the system solid conceptual integrity. Also, the rejected alternatives can be explored by other researchers.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

8. Technology Options

The main research question of this project was answered by defining the Final Software Architecture for the Self-Organizing Logistics Planning System. However, on the request by the incumbent company Prime Vision, an extra research sub-question was raised:

9. *What technologies should be considered to develop the components of the defined software architecture?*

This chapter delves into the potential technologies that could be implemented for this system. The main methods to answer these questions are expert panels, as used in Chapter 7, and a literature study, which also includes the study of existing products. This Chapter includes abstract computer science concepts and complex vocabulary that is inherent to the software solutions that I discuss. Therefore, the target audience for this Chapter is the software specialists that are interested in the potential technologies that would be used in the Logistics Planning System. This Chapter's methods and research question are shown in the Research Flow Diagram of Figure 8-1:

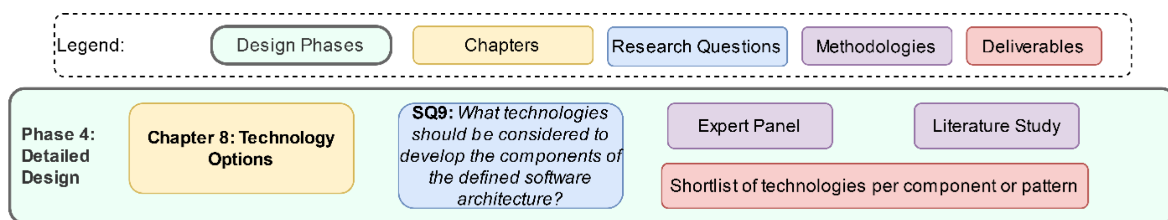

Figure 8-1 Phase 4, Detailed Design, adapted from Systems Engineering Design methodology by Dym et al. (2014)

The bi-weekly expert panels held from May 2020 to July 2020 also included discussions of the potential technologies the system would use. In fact, it was often difficult to speak of the system without mentioning technologies, as the developers often think and design with these in mind. I combined the discussions with the developers with research on state-of-the-art products that also showcase distributed architectures to create a shortlist of technologies for most components.

8.1. Communication options

Communication is at the heart of any distributed system. Chapter 4 gave insight into the orchestrated nature of the algorithm, and thus it is necessary to look for a highly scalable messaging platform that allows this type of ordered consumption of messages.

8.1.1. Event Streaming Platforms

Event Streaming Platforms are the basis for asynchronous messaging, and is the tool that manages messages, stores them in topics, and allows for consumers and services to publish and be subscribed to these topics. As it is crucial in the functioning of the Self-Organizing Logistics Planning System, a highly scalable, lightweight streaming platform should be chosen.

- **RabbitMQ:** An open-source option that supports multiple messaging protocols in its asynchronous messaging platform, message queuing. Furthermore, RabbitMQ allows easy integration of several security measures like authentication and authorization (RabbitMQ, 2020). As an open-source project, many tools and plugins have been developed to add features to RabbitMQ. These tools would save valuable time in the implementation phase and could ease the development of several services. RabbitMQ also has many deploying options that make it simpler to bring into production.

  There are several drawbacks to RabbitMQ, however, as it is designed for asynchronous messaging but not so much on the idea of having persistent storage of messages. This means that once a message is consumed by a client, this message is deleted from the topic is was on, which affects several functions of the Self-Organizing Logistics Planning System, and also hurts the availability in the sense of

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

recovering data from a crash. Persistent storage of messages is key for a system that fully relies on *event sourcing*.

- **Apache Kafka:** Perhaps the most scalable message broker available today, Apache Kafka is another open-source, free lightweight event streaming platform. Developed by Confluent, Apache Kafka is perhaps the most standout option for message brokers. Like RabbitMQ, Apache Kafka supports many languages and has a growing open-source community that develops plugins, example codes, and added tools that work symbiotically with Kafka, like Kai Waehner's Digital Twin example (Waehner, 2019).

  Apache Kafka allows for persistent data storage, which eases event sourcing tremendously. This means that the message broker can work as a secondary database where events are logged on. This is extremely helpful in fault location, debugging, and recovery of systems (Richardson, 2019). There is one function in Apache Kafka that is especially relevant for the Self-Organizing Logistics Planning System: retrospective event streaming. This feature allows for newly-connected services to access messages published before they were subscribed to the message broker. In the Self-Organizing Logistics Planning System, this could be directly applied to vehicles that are recently connected to a depot and can thus read the transport requests that were published earlier. This means that parcels would not have to repeat their request, lowering the overhead of the whole system.

  Many distributed systems utilize Apache Kafka to publish, store, and process information. Furthermore, Apache Kafka offers many integration tools to connect to other useful pieces of software. For example, Kafka Connect and the Confluent Schema Registry are versatile Serializer and connection software that can act as an intermediary between Kafka Topics, Databases, search indexes like Elastic Search, and batch systems like Hadoop (Confluent, 2020b). In the Software Architecture of Figure 7-5, this module would be the Serializer/Schema Registry, shown as a red box.

  Given the flexibility, scalability, and modifiability achievable with Apache Kafka, I recommend it to integrate communications in the Self-Organizing Logistics Planning System. Furthermore, given the available library of distributed system projects that have similarities to the Logistics Planning System, I believe the development team could cut development time tremendously.

Developers at Prime Vision were familiar with both RabbitMQ and Apache Kafka. Furthermore, they verified that Apache Kafka is indeed the most scalable option at the time of writing, with one developer mentioning that Apache Kafka can send a million messages at almost the same speed that it takes to send a thousand.

8.1.2. Remote Procedure Invocation (Request/Response mechanisms)

As I mentioned in Chapter 6, communication in the Self-Organizing Logistics Planning System is mainly done through asynchronous messaging. However, there are always several direct calls between services, and these use other communication protocols. I will shortlist some of the most common remote procedure invocation protocols.

- **REST:** One of the most popular communication mechanisms, REST APIs have been a web service standard for many years. The biggest example of a REST API is the HTTP call done for websites. REST APIs work in the request/response style that was described in Figure 6-2. REST APIs are secure, firewall-friendly, and familiar to most developers. Given these characteristics, it could well be utilized for external API calls to the planning system.

- **gRPC:** gRPC is a more modern approach to building APIs that are a bit more flexible than REST APIs, as it has been developed to ease cross-language communication between services. With gRPC, it is more straightforward to design compact and efficient APIs with many operations (Richardson, 2019). gRPC allows for both request/response and request/async response communications. However, gRPC APIs are less known and require other supporting tools like Protocol Buffer. The specific functioning of these methods is outside the scope of this thesis, but the usage of these technologies should depend on the team's expertise have with them. gRPC requires development teams to have an API-first approach, which has been defined already on this thesis by the development of

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

the Operations table in Appendix H. Therefore if the development team decides to use this communication protocol, part of the work is already defined.

Developers at Prime Vision have developed many applications with REST and gRPC protocols and confirmed their use and applicability to the system at hand.

8.2. ***Database options***

Microservices utilize a database per service, and their modularity and heterogeneity make it possible to leverage the benefits of different technologies simultaneously. Certain databases are better for some tasks than others. Therefore, if a type of service benefits from using a specific type of database, that type of database should be favored in the implementation phase. A Database's main activities are to store information, organize it, and have the data be ready for access when queried. Databases use a specific language for querying called Query Languages, and these classify databases. I will consider databases that use two types of languages: Structured Query Languages (SQL) and Non-Structured Query Languages (NoSQL).

8.2.1. Databases that use Structured Query Languages (SQL)

SQL databases handle structured data and they are thus good for datafiles that contain several classes of information. In context, these information classes could be the parcels' ID, location, and delivery preferences, for example. Therefore, SQL databases should be considered for this sort of information. Some of the most prominent databases that use this query language at the time of writing are:

- **MySQL:** A staple open-source relational database that is familiar to most developers and is free to use. It is developed by Oracle and it is commonly used in web applications. MySQL is also part of the family of the LAMP web application software stack, which is an acronym for Linux, Apache, MySQL, Perl/PHP/Python (Oracle, 2020). Being part of this family of applications means it should be easy to integrate with the technologies of the family. Since the SOLiD algorithm was written in Python, this should be a good option for developing the web application for the Self-Organizing Logistics Planning System.

- **PostgreSQL:** Another free and open-source relational database, PostgreSQL was developed to focus on scalability and modifiability (PostgreSQL, 2020). PostgreSQL supports ACID transactions and automatically updatable views, so it could theoretically work well for the Depot Information Management Service.

8.2.2. Databases that use Non-Structured Query Languages (NoSQL)

- **MongoDB:** MongoDB is a very flexible document database. It supports transactions that are almost ACID, and it has additional features like replicability (MongoDB, Inc, 2020). CQRS works by replicating data, so MongoDB can be a highly valuable database type for these views. MongoDB supports text-based file formats like JSON and it is very scalable. It also uses *sharding* to scale horizontally, so it has a built-in load balancing.

- **DynamoDB:** Amazon's NoSQL database is another powerful database tool that is designed to provide high levels of availability and durability (Amazon Web Services, Inc, 2020). Like MongoDB, it is also a document storage, and it supports JSON-based queries. DynamoDB is an Off-the-shelf product and is therefore not free or open-source. Like MongoDB, DynamoDB would also allow successful implementation of CQRS views.

- **Redis:** Redis is another open-source option that features in-memory data storage, which can be used to cache information, send messages, and store data (Redis, 2020). Redis also has built-in replication and is also designed to achieve high availability. Redis is also a good option for looking up JSON objects, especially for primary key searches, like searching for objects by their ID. This would apply to parcels and vehicles.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

- **Elastic-Search:** Elastic-Search is a special type of database specialized in text-based queries. It is an open-source option with high scalability and performance, used by companies like Slack, Microsoft, and Netflix (Elasticsearch B.V., 2020). It could very well be implemented to look up transport requests. Elastic-Search is typically implemented on the query-side of a CQRS view, and developers should strongly consider this option in the implementation phase.

Developers at Prime Vision are familiar with all of these Databases and have developed applications with most of them. In the Expert Panels held with the developers, Elastic-Search was considered one of the top options to handle the large amounts of data coming from an Event Streaming Platform like Kafka, thus verifying my recommendation. It is important to mention that this is not the sole recommendation, certain services work better with specific types of databases. The selection of each database for each service is thus left for the developers at the implementation phase.

8.3. *Machine Learning Options*

As mentioned in Chapter 6, Machine Learning could provide the Self-Organizing Logistics Planning System considerable scalability through the optimization of its Python Algorithms. In this project's context, there is a need for a way of integrating Machine Learning modules onto the event-driven architecture of the Self-Organizing Logistics Planning System. Some of the technologies that would help achieve that are:

- **H20.ai:** An open-source machine learning framework that allows the application of several other frameworks. H20.ai is a tool that a development team can implement to train a Python, R, or Scala algorithm. The SOLiD algorithm is written in Python and thus this would be a suitable solution.
- **TensorFlow:** TensorFlow is an open-source machine learning platform that helps build machine learning models. Developed by Google, it is the brain behind the teaching of algorithms in a Machine Learning application (Malhotra, 2020). H20.ai and TensorFlow can work symbiotically. As explained by Malhotra (2020), TensorFlow is a framework that facilitates the creation of machine learning models, and H20.ai is a platform that allows this training process to run efficiently, improving scalability.
- **Apache Hadoop:** Developed by Apache, this open-source software allows the deployment of several Machine Learning applications (The Apache Software Foundation, 2020a). An application of Apache Hadoop could be that of a platform where some algorithms are applied.
- **Apache Avro:** Apache Avro is a Serializer software that is highly compatible with Apache Kafka and Kafka Connect. It can act as an intermediary between a Kafka Topic and a Schema Registry and Kafka Hive.
- **Kafka Hive:** Apache Hive is an SQL database solution that is highly compatible with Apache Kafka. The reasoning for using Apache Hive is that it can be integrated into the streaming platform of Apache Kafka, allowing the processing of large quantities of information with great performance (Apache Hive, 2020). Apache Hive could potentially be used to house instances of Tensor Flow, H20.ai, and Apache Hadoop.

These technologies have been tested and applied symbiotically by members of the Open Source community of Apache Kafka (See Confluent, 2020). The development team at Prime Vision should see these examples and evaluate the technologies thereafter. Figure 8-2 shows Confluent's application of Machine Learning Technologies:

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

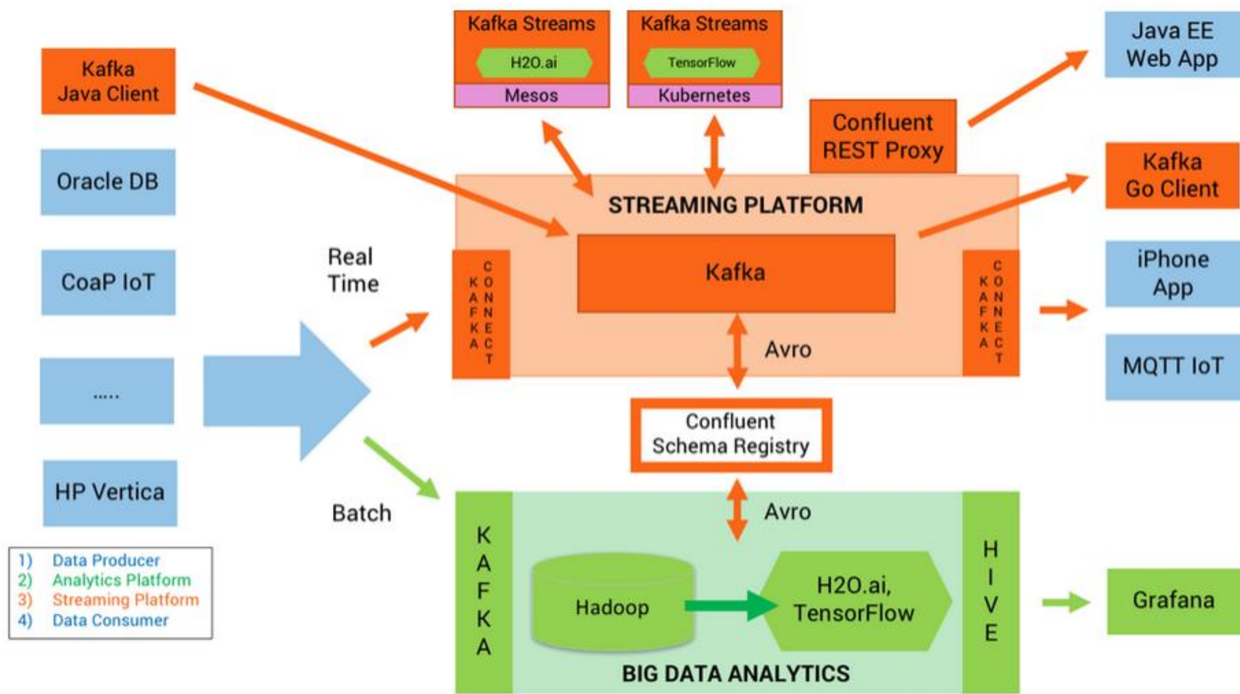

Figure 8-2. Example Application of Machine Learning Tools. Retrieved from Confluent (2020a)

Figure 8-2 shows that a Kafka Streaming Platform, at the center of the image, can be connected using Apache Avro to a Schema Registry. In this example, they use the Confluent Schema Registry. With Apache Avro and the Confluent Schema Registry, the data from the Streaming Platform is translated and in the correct format to enter the Machine Learning cluster, which is encompassed by Kafka Hive. Kafka Hive is the platform where instances of Hadoop, TensorFlow, and H20.ai run.

The Hadoop Cluster is where the Python algorithm would be executed, and the result of this algorithm would be sent in batches to the H20.ai cluster, where there is a machine learning model made with TensorFlow. The model would be executed and the H20.ai platform ensures that the model training is done efficiently.

Developers at Prime Vision have experience utilizing these Machine Learning applications and confirmed them as the current state-of-the-art for production-ready services. However, the development team does not have a lot of experience with the implementation of these technologies alongside an event streaming platform. Therefore, it is reasonable to expect that the project will have a learning curve on the implementation stage.

8.4. *Deployment Options*

As discussed in Chapter 6, deployability is a key Quality Attribute for any software system. Having a system that is easily deployable give a company many benefits, such as reducing time to market and streamlining the production process of a software (Richardson, 2019). In Chapter 6, I recommended containerization as the most sophisticated alternative for deploying services. There are two tools necessary to deploy containers, a Container Platform, and a Container Orchestration Framework.

- **Docker:** Docker is the most widespread and utilized Container Platform, and it is the current industry standard at the time of writing (Riggins, 2019). All the benefits of containerization listed in Chapter 6 are achievable through the Docker Container technology.
- **Docker Compose:** Docker's deploying mechanism allows the execution of several containers. With this tool, it is possible to launch several isolated environments (containers) on a single host (Docker, 2020c). While Docker Compose allows for simple testing during the development phase of a project, it is limited to a single *machine* or *host*, so it is not the best tool for deploying services in the production stage (Richardson, 2019).

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

- **Kubernetes**: Perhaps the most sophisticated Container Orchestration Platform, Kubernetes allows the management of resources of many Container instances. As a Container Orchestration Platform, Kubernetes's functions are to manage resources, schedule deployments, and manage services (Richardson, 2019). This means that with Kubernetes you can decide how many instances run of each service, track, and control how many resources they are consuming, perform load balancing, and update services. Google and Microsoft offer their version of Kubernetes, while Amazon offers an equivalent alternative in Amazon Elastic Container Service (Amazon ECS). The development team should evaluate which alternative suits the Self-Organizing Logistics Planning System better. These platforms are not free to use, and thus vendor selection is an important decision.

Developers at Prime Vision are familiar with Docker Containers and utilize Kubernetes for the deployment of many of their services, and they verified the choice of this technology as a top alternative for deployment.

8.5. API Gateway and Security Options

Security is an important customer requirement. In Chapter 6, I described how the implementation of an API Gateway helps centralize security in a microservices application. With regards to implementing an API Gateway, there are many solutions, some of the most notable are:

- **Netflix Zuul:** Developed by Netflix, this open-source solution performs the most important API Gateway functions, like API composing (also referred to as routing) and authenticating (*Netflix/Zuul*, 2013/2020). However, due to an implementation detail with routing, it is incapable of getting transactions from one service and publish them elsewhere (Richardson, 2019). This makes it difficult to communicate with CQRS views, so this technology would not allow the proper implementation of the necessary functions in the Self-Organized Logistics Planning System.
- **Spring Cloud Gateway:** Another open-source solution, the Spring Cloud Gateway is built on top of many other frameworks useful for API gateways (*Spring Cloud Gateway*, n.d.). This Gateway allows us to filter the clients that enter the application, route requests, and take care of authentication and authorization. Since it is developed in the Spring Ecosystem, it can easily integrate the Spring Security Framework discussed in Chapter 6. According to Richardson (2019), this API Gateway is compatible with CQRS views. Therefore, the development team should consider this solution to save time during the implementation phase.
- **Apache Shiro:** Another Open-source option by Apache, Apache Shiro is ideal for Java applications, which could be important for external clients and web services, that can be coded in Java. This solution support authentication, authorization, cryptography, and session management, thus being a comprehensive security solution for Java environments (The Apache Software Foundation, 2020b).

Developers at Prime Vision mostly develop their own API Gateways, but they know of these options. Therefore, this stays an open option left for the implementation.

8.6. Visualization and Monitoring Options

One of the main concerns for Carriers is to have visibility of the system. In other words, access to key system information. Observability platforms allow the system administrator to have an efficient, helicopter view of a software system. The most notable tool for this function at the time of writing is Grafana:

- **Grafana:** Also utilized in the example of Figure 8-2, Grafana is a visualization tool that can be deployed in any cloud platform. It is friendly with many types of databases, thus making it simple to receive information from them (Grafana Labs, 2020). Grafana uses the source data from a streaming platform and transforms it into intuitive graphs. This feature allows for easy monitoring of the system, and the system includes alerts for particular values that might be considered critical. Furthermore, this tool eases the creation and visualization of KPIs, thus helping satisfy the Carrier's customer requirements. Grafana is also compatible with many technologies, having production-ready plugins with Elastic-

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Search, PostgreSQL, Azure Monitor, GitHub, Kubernetes, JSON, and many others (Grafana Labs, 2020).

The expert panel has previously worked with Grafana and rates it as a very useful tool, thus confirming my recommendation.

The evaluation of these technologies answers Research Sub-question 9 and finishes Phase 4 of Dym and Little's Engineering Design Methodology (2014).

8.7. Summary

In this Chapter, I sought to answer Research Sub-question 9, related to the technologies that could be used for the Self-Organizing Logistics Planning System. I reviewed the most notable technologies implemented for distributed systems at the time of writing, and recommended several solutions for communication, databases, machine learning, deploying mechanisms, API Gateways, and Visualization tools. I checked the recommendations I made with the development team at Prime Vision and they were familiar with most technologies and confirmed my recommendations. This was important for the internal validity of this section for two reasons: First, this review is not comprehensive because there are many options for most technologies. Secondly, I have no experience with these technologies. Therefore, before the developer's verification, I could only infer the value of these technologies from reviews in the literature and the open-source community of many of these technologies. Thus, the developer's verification greatly helps solidify these particular technology recommendations.

The main recommendations I left for developers were Apache Kafka for the Event Streaming Platform, Docker Containers combined with Kubernetes for deployment, the Spring Cloud Framework for the API Gateway combined with the Spring Security Framework, and Grafana for visualization. I also recommended certain technologies for the application of Machine Learning, as many applications are needed for complete implementation, and left a practitioner's example that shows how they can work in tandem.

The recommendations in this chapter are subject to the revision of developers at the implementation stage. However, since I gave examples of functioning distributed systems that also show their source code and how they implemented these technologies together, this Chapter leaves developers with a valuable review of options at the development stage.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

9. Conclusions

9.1. *Revisiting Research Questions*

This thesis project sought to find a suitable software architecture for a Logistics Planning System that leveraged Self Organization. The main justification to pursue this project is that the current last-mile delivery scenario results in costly operations and heavy carbon footprints. The Self-Organizing Logistics Planning System envisioned in this project addresses these challenges. This project focused on the resolution of the following Research Question:

What software architecture could support a Logistics Planning System that uses Self-Organization?

To answer this question, I formulated nine different research sub-questions that guided the research process. I will summarize the answer to each in this section.

1. *What is software architecture and how can it be designed?*

Software architecture is a discipline that seeks to represent software systems. These abstract representations show the key features of a software system and the key relationships between its components. However, its scope is not detailed, but abstract. Therefore, the implementation details from each component are hidden from a system's Software Architecture. Given these characteristics, a system's Software Architecture is a prime guide for developers at the implementation phase that also leaves them enough freedom to choose particular implementation methods.

The main method used to design software architecture is to use and combine Architectural Patterns, which are documented solutions to commonplace problems. Architectural patterns are documented in a context, problem, solution that allows them to be compared to one another systematically.

2. *What are the current operations and capabilities of Carriers in last-mile delivery?*

The main operations in the last mile are the following: First, parcels are loaded onto conveyor belts where their barcodes are scanned. Second, a centralized planning system assigns the parcel to a vehicle that has a pre-established fixed route. These routes are based on postcode aggregation, and carriers create several delivery zones that fix routes every 1 to 4 months. Thirdly, parcels are loaded on their assigned vehicle, and a handheld application calculates the optimal route for the driver to use on that particular day, usually minimizing distance traveled. Finally, the drivers carry out the delivery. If a customer cannot receive their parcel, the driver will attempt to deliver the parcel to a neighbor or return the package to a nearby Collection and Delivery Point.

Carrier's technical capabilities at the operational level include scanning parcels at three moments, arrival at a sorting depot, loading onto a vehicle, and loading off of a vehicle. Furthermore, the handheld devices used by drivers can perform these scans, calculate routes, and are connected to a Global Positioning System. Lastly, the current centralized planning system is connected to the depot's conveyor belts, so the transport process from the depot to their loading docks is streamlined. This is also possible because of the usage of fixed routes, since this simplifies depot layout, giving each route an assigned spot in a depot.

3. *What are the customer requirements for a logistics planning system?*

Shippers and customers have similar requirements in their role of clients to the Carriers. They mainly want more options for delivery, track and trace options, customer notifications, and low costs. Carriers, on the other hand, want to fulfill these requirements, and thus focus on lowering costs to stay competitive, and want sophisticated logistics planning systems that enable this track and trace functionality, allows them to recollect data for building KPIs, and give accurate delivery windows for customers. With regards to system qualities, Carriers rated availability, security, performance, modularity, and modifiability as the most important characteristics for the system. Prime Vision, representing the Carrier's software providers, wants to suit the customers and Carrier's requirements and considered the scalability, modularity, modifiability, deployability, and testability of the system the most important characteristics to do so. Furthermore, Prime Vision wants to achieve a strategic broker position between carriers, and therefore the Logistics Planning System should be able to house several Carriers under one

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

platform. Finally, Prime Vision desires the system to have a subscription or usage-based billing system to ensure a sustainable business model.

4. *What are the functional requirements of a self-organizing logistics planning system?*

To answer this question, I analyzed the agents that comprise Thymo Vlot's algorithm, which were parcels, vehicles, and a central platform. I then suggested a system with a different set of actors: parcels, vehicles, and three new agents, a depot digital twin, a depot platform, and a vehicle filter. These three additional actors prevent the system from having a single actor that knows all the sensitive data of the carriers that use the system.

I summarize the functional requirements for each of the actors in the following table:

Capacity/Entity	*Parcels*	*Vehicles*	*Depot Platform*	*Depot Digital Twin*	*Vehicle Filter*
Compute Algorithms	*X*	*X*	*X*	*X*	*X*
Store Information	*X*	*X*	*X*	*X*	
Communicate	*X*	*X*	*X*	*X*	
Track its Position	*X*	*X*			
Make Payments	*X*				
Receive Payments		*X*		*X*	
Log Events	*X*	*X*		*X*	
Check its capacity		*X*		*X*	
Connect Entities			*X*		
Know the location of all depots		*X*	*X*		
Know the Center of Delivery of vehicles					*X*
Orchestrated activities between entities	*X*	*X*	*X*	*X*	*X*

5. *What is the most appropriate architectural style for a self-organizing logistics planning system?*

Given the stark requirements in scalability, modifiability, modularity, and availability, I chose the microservices architectural *style* for the Self-Organizing Logistics Planning System. This also fits Prime Vision's development practices with small agile teams and continuous deployment and testing.

6. *How to decompose a self-organizing logistics planning system?*

To decompose the system, I first developed a conceptual view of the system with the main actors. Then, I separated each actor into subclasses by their activities. This led to a class diagram specified in Figure 5-6. This showed the main relationships of the system and did a preliminary decomposition of the system based on responsibilities. I finalized the decomposition of the system by applying the Decomposition by Business Capability *pattern*. This resulted in a separation of concerns based on business concepts, like creating transport requests, making payments, and storing information. After applying this *pattern*, the system was separated into 17 services.

The Parcel Digital Twin became an aggregate that comprised 5 services:

1. Parcel Information Management Service: stores and publishes the parcel's information.
2. Transport Request Service: loads the parcel's information and creates a transport request.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

3. Vehicle Selection Service: Compares the vehicles' bids and chooses according to parcel preferences.
4. Transfer Evaluation Service: Requests equivalent bids and evaluates transferring to a second vehicle.
5. Parcel Wallet Service: Handles the payment to vehicles and depots.

The vehicle digital twin also became an aggregate that comprised of 5 services:

6. Vehicle Information Management Service: stores and publishes the vehicle's information.
7. Bid Creation Service: reads transport requests from parcels and creates bids.
8. Vehicle Positioning Service: informs the vehicle and relevant services about the vehicle's position.
9. Vehicle Wallet Service: Handles billing and reception of payments.
10. Depot Connection Request Service: Requests to be connected to nearby depots.

The Depot Digital Twin had its functions separated across 4 services:

11. Depot Information Management Service: Updates the depot's capacity on its loading docks.
12. Parcel Assignment Service: Loads the depot's capacity and assigns requesting parcels to a loading dock.
13. Depot Wallet Service: Handles billing and reception of payments.
14. Filter Service: Filters transport requests to relevant vehicles based on the center of delivery.

The Depot Platform, which connects vehicles to nearby depots, comprised 3 services:

15. Depot Location Service: Stores the location of a region's depots.
16. Depot Selection Service: Receives Depot Connection Requests and filters the relevant depots for each requesting vehicle.
17. Depot Connection Service: Connects requesting vehicles to relevant depots.

7. Which architectural patterns would best suit a self-organizing logistics planning system?

After decomposing the system, I select architectural patterns to solve the inherent drawbacks of a distributed system. This is done by selecting architectural patterns. Drawing these from the literature and examples on online repositories, I selected the following patterns:

For communication, I selected a combination of the Asynchronous Messaging pattern and the Synchronous Remote Procedure Invocation pattern. The asynchronous messaging pattern is comprised of two more patterns: The Request/Asynchronous Request pattern and the Publish/Subscribe pattern. By emphasizing asynchronous interactions between services, the overall system achieves superior availability.

To execute the business logic and ensure the orderly execution of transactions, I applied the *choreography-based saga pattern*. This pattern uses local events from services to carry out business logic. Therefore, this pattern depends on the services publishing their relevant events. The main tool for this is the application of the *event-sourcing pattern*, which specifies that services will publish their interactions in communication channels that other relevant services can access.

To solve the complexity of querying a distributed system, I combined the application of the API Composition *pattern* and the Command Query Responsibility Segregation *pattern*. The first pattern acts like a materials procurer that receives an order with a list of materials and then fetches them one by one, delivering the aggregate result. This is the simplest *pattern* to apply in this context. The Command Query Responsibility Segregation *pattern* is more complex and requires that the relevant services have 2 separate databases: One for transactions and operations, and one database for queries. This pattern makes it easier to query those services with a query-only database, and it also allows the development of query-only services, which is the main reason why it is valuable in distributed systems. By making a query-only service that is subscribed to all of the parcel's sub-services, for example, we can have a Parcel Overview Service, that can summarize the lifecycle of a parcel, including all interactions. I developed two such query-only services for parcels and vehicles.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

External communications also have several security and performance challenges, that arise from clients having to go through a firewall and the fact that they utilize slow, wireless internet connections. To solve these challenges and minimize traffic at the firewall, I implemented the API Gateway *pattern*, that centralizes querying from external clients to a single service. This becomes a prime position to apply security measures like the Access Token Pattern and other authorization solutions, which is the main way in which I tackled security for the Self-Organizing Logistics Planning System.

Finally, I discussed three deployability patterns, concluding in the selection of the Service as a Container *pattern*. This solution implies using a modern technology of containerization that packages a service along with all the software that it requires to be executed. This streamlines the deployment of services, making load balancing automatic, and achieving superior performance to other options.

After selecting these architectural patterns, I drew two figures that comprise the Draft Software Architecture for the Self-Organizing Logistics Planning System. These are Figure 6-17 and Figure 6-18.

8. *Would the conceptual design comply with the customer requirements?*

The verification of the Software Architecture was done through expert panels held at Prime Vision with software developers and software architects. This discussion led to several concerns. The first was the lack of ACID transactions. These are transactions typical to traditional databases that have consistent data. In a distributed system, it is difficult to achieve them. The way to circumvent lacking these ACID transactions is to properly implement sagas. This discussion did not realize any changes in the architecture but did highlight that it will be necessary to carefully test sagas in the implementation phase.

Afterward, the Expert Panel recommended several changes for certain services. The Depot Information Management Service and the Parcel Assignment Service were merged into one service. This guaranteed ACID transactions in that particular interaction, which is important for the consistency and proper functioning of the system. Therefore, the final depot digital twins only have 3 services: The Depot Information Management System, the Vehicle Filter, and the Depot Wallet Service.

Then, the developers recommended the addition of three services: the usage-based billing service, a parcel dimensions measuring service from another project, and a mapping service that would be utilized for the driver's handheld application. I added these services in their respective views and redrew the Software Architecture, now in its final form. Finally, developers recommended the addition of filters and a careful specification of each of the client's clearance. These filters and security measures were added to the tasks of the API Gateway in the External APIs view.

I developed four views for the final software architecture. The first concerns the CQRS views for the parcels and vehicles, found in Figure 7-2. The second is an example CQRS view of a single service in Figure 7-3. The third is the external APIs view of the system in Figure 7-4, and the last view is the final Conceptual view of the Software Architecture of Figure 7-5. These four figures comprise the Final Software Architecture for a Self-Organizing Logistics Planning System, answering the main research question of this project.

However, several functional requirements could not be tackled architecturally. This was not a result of bad architecture design, but of the nature of the requirement. As I mentioned in Chapter 2, some problems can be addressed with software architecture, and others by algorithms. This thesis does not span the latter, and thus cannot solve them. Therefore, these are functions of the algorithm and are impossible to address with the architecture alone. For example, sending an accurate Estimated Time of Arrival for a parcel to a customer was considered *very important* to the Carriers, but it does not depend on the Software Architecture of the system, but its algorithms. Thymo Vlot mentioned this particular point as a limitation of his algorithm, as a reliable calculation of Estimated Time of Arrival is inherently incompatible with a self-organizing logistics system that changes its solution several times during the day. Thus, this remains an unsatisfied functional requirement. However, if a solution to this problem is found, the Software Architecture I offer in this project should be able to support said solution, as long as it does not require a complete overhaul of the system.

However, Prime Vision's request led to a final research sub-question, relevant for the implementation phase of the project, which was the following:

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

9. What technologies should be considered to develop the components of the defined software architecture?

To answer this research question, I looked at examples of distributed systems in the literature and online repositories. The final recommendation was a system based on Apache Kafka as an event streaming platform to which all the other functions can be connected. Other technologies include the usage of REST and gRPC APIs for external clients' requests, several Database Systems, such as DynamoDB, MongoDB, and PostgreSQL. Specific databases are better for specific applications. Therefore, the final selection of each database type for each service is left up to the developers in the implementation phase. I included an example application of machine learning in a distributed system that relied on Apache Avro to serialize information from the Event streaming platform to Apache Hive, which receives the information in batches. Then, an instance of Apache Hadoop stores this data, which is then consumed by an H20.ai cluster that uses machine learning algorithms developed with TensorFlow.

I finally recommended using certain off-the-shelf and open source solutions for API Gateways and Security, like Spring Cloud Gateway, which can be integrated into a Spring Security Framework. Apache Shiro would be another notable option specific to Java applications if the development team makes any for the external clients.

9.2. ***Relationship with Management of Technology & Supply Chain Management***

This project required a multi-disciplinary approach to be resolved. I first had to delve into the parcel delivery environment, drawing information from the ample sources in the logistics literature and interviews with industry experts. This gave me insight into the type of managerial and practical issues companies have in the delivery field and directed me to find solutions for them in my project. In this project, I designed software architecture for a new technological solution, and it was necessary to think of this solution as a corporate asset that could be leveraged. This was very interesting because there were several interested stakeholders in the project: On the one hand, the Carriers were the most direct customers, as they would be the ones using the Logistics Planning System, but on the other, the customers of e-retailers and the e-retailers themselves would also interact with the system and have a stake in the process. This meant that the design had to accommodate to the wishes and desires of many different stakeholders, making certain design decisions complex. The positioning of the incumbent company Prime Vision was also taken into consideration in the design of this software architecture, seeking to give this company the privileged broker position it seeks in the last-mile delivery market. Therefore, this project focuses on satisfying a wide array of customer requirements while significantly aiding the strategic and competitive positioning of Prime Vision. This is a hallmark of a Management of Technology project.

Furthermore, my project concerned a highly innovative technology, self-organizing logistics systems, and solved some of the obstacles to its implementation by applying state-of-the-art tools from other disciplines, like computer science and software architecture. Therefore, I leveraged several high-end technologies from different disciplines to produce a software architecture that can help solve corporate and societal issues, all while advancing topics of scientific interest in the logistics field. This is also a hallmark of a well-executed Management of Technology project.

This project also leaves Prime Vision with a workable framework to implement this Logistics Planning System. The defined software architecture has many small components, which eases the separation of responsibilities during the implementation phase. In essence, given the architecture that I proposed in this project, it should be simpler for Prime Vision to develop a roadmap for implementation and separate the development team into small, 2-3 person development teams for each of the components. This is always an important challenge in the implementation phase of a project and is partly addressed by this thesis.

The system was designed in a way that it would be a desirable logistics planning system for a single carrier, and scalable enough to house several carriers and enable collaborative delivery. This gives Prime Vision a clear strategy to focus on catering this Logistics Planning Systems to carriers in the short and medium term and encompassing various carriers under the same platform in the long term. Having designed the system with Carriers as the main focus customer, the designed software architecture emphasizes the survival of the Logistics Planning System in the earliest adoption phases, the most critical phase for project success in new product development (see Ortt (2010)).

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Furthermore, the design gives a clear idea of the type of software systems that need to be developed, with classics like API gateways and modern components like the Application of Machine learning tools. Thus, Prime Vision now can clearly evaluate which components it would be able to develop in-house, which components should be developed in collaboration with other organizations like TNO, and which components should be outsourced completely. This is also an important question for a Manager of Technology, and this project helps to address it.

I also addressed the monetization and commercialization of this potential technology. Prime Vision wishes to have more services in its portfolio and increase its current wallet with its customers. This project helps them enter a new segment of the parcel logistics market, the last mile of delivery. According to their preferences, I suggested a usage-based billing system that would grant Prime Vision with a sustainable business model and would exploit its position as a broker between Carriers.

Another feat of this project is that the suggested software architecture results in a system that does not require physical computers for each parcel. This occurs because I designed the Parcel Information Management Service to make the parcel's location equal to its vehicle's location. This simple solution eliminates an enormous potential financial investment in IoT devices for each parcel, which would slow down the development of the project and increase its financial risk.

9.3. Scientific Value of the Project

The academic value of this project is twofold: On one hand, the Logistics Planning System I proposed in this thesis allows for the implementation of a Self-Organized Logistics algorithm, such as Thymo Vlot's algorithm. This implies that if the system is developed, it would be able to feasibly cope with the computational complexity of collaborative delivery, an area of major scientific interest. Furthermore, the definition of the digital agents for vehicles does not discriminate with vehicle type, it could very well be a bike or a person with a smartphone. This means that this logistics planning system, as envisioned by Vlot, would also allow the implementation of multi-modal transport, another key topic in the logistics literature. On the other hand, this project also advanced the scientific literature that unites logistics and software architecture. Distributed systems have been studied on their own, but the literature that applies this knowledge to the logistics field is scarce. This project, therefore, becomes an example of how to tackle complex modern city logistics issues with the application of state-of-the-art software design strategies.

9.4. Societal Value of the Project

This project is a step forward to implementing a less congesting logistics planning system for parcel delivery. If the Logistics Planning System I proposed is developed, it would be possible to harness the benefits of Thymo Vlot's SOLiD algorithm. In the best-case scenario, the logistics planning system would allow horizontal collaboration between carriers, reducing duplication of routes, and overall emissions in cities. While Carriers would naturally resist participating in such a collaborative scheme, the logistics planning system could also be used as a leveraging tool with governmental authorities whose priority is to safe keep the environment.

Other benefits specified by Vlot would also be attainable, such as added flexibility and options, both for the end customer and the delivery drivers, as the system adds green delivery and other options for customers and flexible centers of delivery for drivers.

9.5. Practical & Managerial Implications of the Project

The software architecture I propose in this project also has significant commercial value. A key feature is that the proposed software architecture is of a distributed system that uses state-of-the-art microservices patterns. The basic premises behind this software architecture are system agnostic, and the same concepts and services defined in this thesis project can be used by Prime Vision to develop other distributed systems. The main change would be the services and algorithms used, but the main communication style, security patterns, and execution of sagas should give Prime Vision tools with enormous commercial value, as it could become the basis for a line of products based on distributed systems. This value has already started to realize: Within the project's working group, comprised of myself, my supervisor, and the experts who helped me verify the approach and technologies I suggested, several discussions have already been held to address other business challenges with the proposed

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Software Architecture. Prime Vision intends to extend this distributed system to a warehouse software product that applies self-organization to loading docks, containers, and conveyor belts. The principles and the architecture would be the same, it would just be a matter of adapting the algorithms.

This project also has an important pedagogic value. First, the inclusion of experts in the design process formed a working group within the Prime Vision organization, which now holds conference calls with the rest of the participants of the overarching SOLiD project. This project has thus made its mark in the Benelux logistics field by pushing the SOLiD project forward. Second, the development of the software architecture of this project trained several members of Prime Vision in how a potential Self-Organizing Logistics Planning System would work. Third, the system's software architecture enabled me to communicate with all members of Prime Vision how this system would work, independent of their background. Given the complexity that this project handles, I consider this to be a great achievement. In conclusion, this project made a significant step towards making self-organization in logistics a reality.

If the Self-Organizing Logistics Planning System were implemented, the prices in delivery would fall. As in any competitive environment, where competition is the method through which a company obtains customers, the clearer the competitive channels, the larger the benefits for the customers. And there is no more direct channel of competition than an auctioning system for each parcel. This is the main reason why Carriers would resist the implementation of this system, as it would directly, efficiently, and objectively select the best vehicle to carry out a particular delivery, thus the main beneficiary would be the customer. Therefore, Carriers would not be able to leverage certain intangible assets like their corporate image, as it would only depend on their price and their emissions. Consequently, the number of auctions their vehicles win would depend on their actual cost and emissions rather than their corporate sustainable image. This system would thus enhance transparency and affordability for customers and e-Retailers at the expense of long term profit margins for Carriers.

This is why Prime Vision should first develop a competitive routing system as a standalone product for each Carrier, and then seek to leverage the user base to promote collaborative delivery. This would have much higher chances of success than suggesting a collaborative delivery platform from the start. When applied in isolation, the Carriers are the main beneficiaries of the auctioning system. When seeking to implement collaborative delivery, third parties and governmental entities could prove to be valuable facilitators for the project. However, this remains uncertain.

9.6. ***Project Limitations & Recommendations for Future Research***

This project comprised the design of the software architecture of a system, but not the development of said system. Therefore, there are important limitations with regards to testing the design, as it has not been developed yet. Furthermore, certain software components would add value to the company but were not part of this project, such as the software architecture of the driver's mobile app, the carrier's administrator web application, and the mapping service used by the driver's mobile app. Furthermore, in this project, I suggest the inclusion of depot digital twins and a depot platform, but I do not develop the code of these agents. I do define the logic and responsibilities these agents have, but Prime Vision will still have to develop the code for these entities on its own.

Another important limitation of this project is that the review of available architectural patterns was not comprehensive. Because of time constraints, I limited my scope to architectural patterns considered to certain sources, which directly affects the final output of this project. However, it is important to mention that there is no such thing as a comprehensive consideration of architectural patterns, as they are, by definition, found in practice and not cataloged in one place.

The Questionnaire for developers used in Chapter 3 to prioritize certain quality attributes over others was only answered by the members of the Expert Panel for a total of 4 answers, which is not a sample large enough to be considered significant. Therefore, the project does not have proven external validity. However, this project is highly contextual, and having only developers that were acquainted with the SOLiD algorithm increases the internal validity of the results. In hindsight, it would have been more appropriate to use a Forced Choice ranking scale that made developers and carriers rank requirements explicitly in comparison to each other since this would have facilitated the selection of architectural patterns and prioritization of requirements in the design.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

In this thesis, I utilized Dym, Little, and Orwin's framework for the design methodology. However, after completing the report, I realized that perhaps a better methodology would have come from software development literature. Dym, Little, and Orwin's framework is better suited to the development of other types of systems and physical goods. The concepts from their framework that I was able to extract and successfully apply were the definition of customer requirements, constraints, design objectives, and preliminary design. The rest of the phases were not too compatible with software architecture design, as seen in the preliminary and detailed design phases of Chapter 7 and 8. This also discards Johannesson and Perjons's methodology (2014). If I started the project again, I would look for software design frameworks that were better suited for these design challenges.

Further work must be carried out to separate Vlot's code in the suggested services. Since the original algorithm and its simulations were comprised of a series of Python scripts, Prime Vision now has the task to dive into this algorithm and fetch the business logic parts that belong to each of the services that I defined in the software architecture. Additionally, further work is needed to refine the algorithm with the limitations Thymo Vlot mentioned in his thesis report, such as adding realism by adding different dimensions to the parcels and incorporating delivery time windows for customers. These challenges are still present since I did not alter the original algorithm.

As mentioned in Section 9.1, this project did not address issues that are not architectural, such as ensuring that parcels have a reliable Estimated Time of Arrival. These questions belong to the realm of the logistics planning system's algorithm and are thus outside of the scope of this thesis. However, it is still an important function for carriers and should be addressed in further research. Perhaps it would be possible to make reasonable estimates using the data gathered in the machine learning modules for the parcel and vehicle digital twins, making this an interesting research question for the implementation phase of the project.

This project offers *a solution* for the software architecture needed for the Self-Organizing Logistics system, not the *only* solution possible. Software Architecture has infinite solutions and this thesis only offers one possibility. Further research should compare the usage of a SOA-based architecture or a different decomposition in services for the system against the solution offered by this project. If the systems are developed, then more accurate metrics and hard data could sustain a decision of which framework is best.

There is further research to be carried out with the auctioning system used in the algorithm. Currently, the auctions are sequential, and vehicles are locked into each auction: The vehicles must wait for an auction to finish before they can bid for other parcels. This could create significant bottlenecks if a particular parcel is unresponsive. There are some solutions from game theory that have more complex, robust types of auctions, such as simultaneous item auctioning (See Rosenthal and Wang (1996)) and combinatorial auctioning (See Nisan (2007, p. 267)). There should be further research to evaluate the integration of these robust types of auctions in the self-organizing logistics algorithm.

Another point of interest is the case of lost parcels in a collaborative scenario. There is a challenge in making the system desirable for carriers, and an important question arises: Who is responsible for a lost parcel when several actors collaborated to deliver it? The research by Daudi et al. (2016) reinforces that trust is a necessary factor in horizontal collaborations and that impartial solutions agreed upon by all members of an alliance are usually stronger in the long term. The odds become better when there is an intermediary between companies that ensures an impartial application of these agreements. In context, Prime Vision has great positioning to assume this role of arbiter between carriers. Concerning a solution for shared responsibility, I suggest the incorporation of quantitative finance and insurance algorithms into the Logistics Planning System.

Some solutions address these shared responsibility issues in the risk management and insurance literature, referred to as "Moral Hazard" and "Deposit Insurance" (See Dau et al. (2020) and Wu et al. (2020)). An insurance-based solution would be interesting for the Self-Organizing Logistics Planning System. There should be further research to see if it would be possible to add an insurance value to each parcel, that could be added to the parcel's overcost. In such a system, all the carriers that handle a parcel automatically add an insurance premium to the delivery cost of the parcel, thus accounting for the potential loss of the parcel. If such a system were agreed to by the Carriers using the system, and Prime Vision or another party acts as an arbiter between, the collaboration conditions would

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

be close to ideal according to literature research on collaboration (Basso et al., 2019; Cruijssen et al., 2007; Daudi et al., 2016; Martin et al., 2018).

Testing software was outside of the scope of this thesis and will provide Prime Vision with important development challenges, as testing microservices and sagas is a process that is not intuitive. Given that the development team has not worked with microservices extensively before, this becomes an important obstacle for the project in the implementation phase.

Future research also includes the development and testing of this microservice environment. Prime Vision should compare some technology solutions to others and verify compliance with customer requirements. Fortunately, this thesis report includes an extensive list of priorities for all involved stakeholders, which will simplify the prioritization of technologies and solutions in the implementation phase.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., … Zheng, X. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *ArXiv:1603.04467 [Cs]*. http://arxiv.org/abs/1603.04467

Allen, J., Piecyk, M., Piotrowska, M., McLeod, F., Cherrett, T., Ghali, K., Nguyen, T., Bektas, T., Bates, O., Friday, A., Wise, S., & Austwick, M. (2018). Understanding the impact of e-commerce on last-mile light goods vehicle activity in urban areas: The case of London. *Transportation Research Part D: Transport and Environment*, *61*, 325–338. https://doi.org/10.1016/j.trd.2017.07.020

Allen, J., Bektaş, T., Cherrett, T., Friday, A., McLeod, F., Piecyk, M., Piotrowska, M., & Austwick, M. Z. (2017). Enabling a Freight Traffic Controller for Collaborative Multidrop Urban Logistics: Practical and Theoretical Challenges. *Transportation Research Record: Journal of the Transportation Research Board*, *2609*(1), 77–84. https://doi.org/10.3141/2609-09

Amazon Elastic Compute Cloud. (2020). *Amazon Machine Images (AMI).* https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html

Amazon Web Services, Inc. (2020). *Amazon DynamoDB*. Amazon Web Services, Inc. https://aws.amazon.com/dynamodb/

Amsterdam. (n.d.). *Volg het beleid: Schone lucht* [Webpagina]. Amsterdam.nl; Gemeente Amsterdam. Retrieved July 31, 2020, from https://www.amsterdam.nl/bestuur-organisatie/volg-beleid/duurzaamheid/schone-lucht/

Apache Hive. (2020). *Apache Hive*. https://hive.apache.org/

Bartholdi, J. J., Eisenstein, D. D., & Lim, Y. F. (2010). Self-organizing logistics systems. *Annual Reviews in Control*, *34*(1), 111–117. https://doi.org/10.1016/j.arcontrol.2010.02.006

Bass, L., Clements, P., & Kazman, R. (2003). *Software Architecture in Practice*. Addison-Wesley Professional.

Basso, F., D'Amours, S., Rönnqvist, M., & Weintraub, A. (2019). A survey on obstacles and difficulties of practical implementation of horizontal collaboration in logistics. *International Transactions in Operational Research*, *26*(3), 775–793. https://doi.org/10.1111/itor.12577

Boudriga, N. (2010). *Security of mobile communications*. Boca Raton : CRC Press. http://archive.org/details/securitymobileco00boud

Brazdil, P., Gams, M., Sian, S., Torgo, L., & van de Velde, W. (1991). Learning in distributed systems and multi-agent environments. In Y. Kodratoff (Ed.), *Machine Learning—EWSL-91* (Vol. 482, pp. 412–423). Springer Berlin Heidelberg. https://doi.org/10.1007/BFb0017034

C++ Foundation. (2015). *Serialization and Unserialization, C++ FAQ*. https://web.archive.org/web/20150405013606/http://isocpp.org/wiki/faq/serialization

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., & Zhang, Z. (2015). MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *ArXiv:1512.01274 [Cs]*. http://arxiv.org/abs/1512.01274

Cleophas, C., Cottrill, C., Ehmke, J. F., & Tierney, K. (2019). Collaborative urban transportation: Recent advances in theory and practice. *European Journal of Operational Research*, *273*(3), 801–816. https://doi.org/10.1016/j.ejor.2018.04.037

Cloudera Docs. (2019). *Schema Registry Overview*. https://docs.cloudera.com/csp/2.0.1/schema-registry-overview/topics/csp-schema_registry_overview.html

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Confluent. (2020a). *Build and Deploy Scalable Machine Learning in Production with Kafka*. Confluent. https://www.confluent.io/blog/build-deploy-scalable-machine-learning-production-apache-kafka/

Confluent. (2020b). *Kafka Connect—Confluent Platform 6.0.0*. https://docs.confluent.io/current/connect/index.html

Costa, R., Jardim-Goncalves, R., Figueiras, P., Forcolin, M., Jermol, M., & Stevens, R. (2016). Smart Cargo for Multimodal Freight Transport: When "Cloud" becomes "Fog." *IFAC-PapersOnLine*, *49*(12), 121–126. https://doi.org/10.1016/j.ifacol.2016.07.561

Cruijssen, F., Cools, M., & Dullaert, W. (2007). Horizontal cooperation in logistics: Opportunities and impediments. *Transportation Research Part E: Logistics and Transportation Review*, *43*(2), 129–142. https://doi.org/10.1016/j.tre.2005.09.007

Daudi, M., Hauge, J. B., & Thoben, K.-D. (2016). Behavioral factors influencing partner trust in logistics collaboration: A review. *Logistics Research*, *9*(1), 19. https://doi.org/10.1007/s12159-016-0146-7

Deckert, C., & Görs, N. (2019). Transport carbon footprint in the german courier, express and parcel industry (CEP industry). *NachhaltigkeitsManagementForum | Sustainability Management Forum*, *27*(1), 23–30. https://doi.org/10.1007/s00550-018-0471-1

Dev.to. (2020, July). *Exposing microservices with an API gateway and the API composition pattern*. DEV Community. https://dev.to/benzo/exposing-microservices-with-an-api-gateway-and-the-api-composition-pattern-54i4

Digital.ai. (2020). *What is DevOps? The Ultimate Guide to DevOps*. https://digital.ai/resources/devops-101/what-is-devops

Docker. (2020a). *What is a Container? App Containerization*. https://www.docker.com/resources/what-container

Docker. (2020b, July 30). Docker Index: Dramatic Growth in Docker Usage Affirms the Continued Rising Power of Developers. *Docker Blog*. https://www.docker.com/blog/docker-index-dramatic-growth-in-docker-usage-affirms-the-continued-rising-power-of-developers/

Docker. (2020c, October 1). *Overview of Docker Compose*. Docker Documentation. https://docs.docker.com/compose/

Dou, W., Tang, W., Wu, X., Qi, L., Xu, X., Zhang, X., & Hu, C. (2020). An insurance theory based optimal cyber-insurance contract against moral hazard. *Information Sciences*, *527*, 576–589. Scopus. https://doi.org/10.1016/j.ins.2018.12.051

Ducret, R. (2014). Parcel deliveries and urban logistics: Changes and challenges in the courier express and parcel sector in Europe — The French case. *Research in Transportation Business & Management*, *11*, 15–22. https://doi.org/10.1016/j.rtbm.2014.06.009

Dym, C. L., Little, P., & Orwin, E. J. (2014). *Engineering design: A project-based introduction* (4th edition). Wiley.

Elasticsearch B.V. (2020). *Open Source Search: The Creators of Elasticsearch, ELK Stack & Kibana | Elastic*. https://www.elastic.co/

Emarsys. (2020). *Global commerce insights*. Commerce Insights. https://insights.emarsys.com/

Ergun, O., Kuyzu, G., & Savelsbergh, M. (2007). Reducing Truckload Transportation Costs Through Collaboration. *Transportation Science*, *41*(2), 206–221. https://doi.org/10.1287/trsc.1060.0169

Evans, E. (2004). *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional.

Faugere, L., & Montreuil, B. (2016). *Hyperconnected City Logistics: Smart Lockers Terminals & Last Mile Delivery Networks*. 15.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Florio, A. M., Feillet, D., & Hartl, R. F. (2018). The delivery problem: Optimizing hit rates in e-commerce deliveries. *Transportation Research Part B: Methodological*, *117*, 455–472. https://doi.org/10.1016/j.trb.2018.09.011

Fraser, B. Y. (1997). *Site Security Handbook*. https://tools.ietf.org/html/rfc2196

Gevaers, R., Van de Voorde, E., & Vanelslander, T. (2011). Characteristics and Typology of Last-mile Logistics from an Innovation Perspective in an Urban Context. In C. Macharis & S. Melo, *City Distribution and Urban Freight Transport* (p. 14398). Edward Elgar Publishing. https://doi.org/10.4337/9780857932754.00009

Grafana Labs. (2020). *Grafana Features*. Grafana Labs. https://grafana.com/grafana/

Haerder, T., & Reuter, A. (1983). Principles of transaction-oriented database recovery. *ACM Computing Surveys*, *15*(4), 287–317. https://doi.org/10.1145/289.291

Harrington, T. S., Singh Srai, J., Kumar, M., & Wohlrab, J. (2016). Identifying design criteria for urban system 'last-mile' solutions – a multi-stakeholder perspective. *Production Planning & Control*, *27*(6), 456–476. https://doi.org/10.1080/09537287.2016.1147099

Hashicorp. (2020). *The Vault Project*. Vault by HashiCorp. https://www.vaultproject.io/

Hülsmann, M., & Windt, K. (Eds.). (2007). *Understanding Autonomous Cooperation and Control in Logistics: The Impact of Autonomy on Management, Information, Communication and Material Flow*. Springer-Verlag. https://doi.org/10.1007/978-3-540-47450-0

Interviewee A. (2020, July 15). *Planning System Requirements* [Personal communication].

Interviewee B. (2020, July 22). *Planning System Requirements* [Personal communication].

Interviewee C. (2020, July 31). *Planning System Requirements* [Personal communication].

Interviewee D. (2020, July 22). *Planning System Requirements* [Personal communication].

Interviewee E. (2020, July 30). *Planning System Requirements* [Personal communication].

Johannesson, P., & Perjons, E. (2014). *An Introduction to Design Science*. Springer International Publishing. https://doi.org/10.1007/978-3-319-10632-8

Krajewska, M. A., & Kopfer, H. (2009). Transportation planning in freight forwarding companies: Tabu search algorithm for the integrated operational transportation planning problem. *European Journal of Operational Research*, *197*(2), 741–751. https://doi.org/10.1016/j.ejor.2008.06.042

Lasi, H., Fettke, P., Kemper, H.-G., Feld, T., & Hoffmann, M. (2014). Industry 4.0. *Business & Information Systems Engineering*, *6*(4), 239–242. https://doi.org/10.1007/s12599-014-0334-4

Litman, T., & Burwell, D. (2006). Issues in sustainable transportation. *International Journal of Global Environmental Issues*, *6*(4), 331. https://doi.org/10.1504/IJGENVI.2006.010889

Malhotra, S. (2020, April 22). *TensorFlow Vs H2O: The Best Enterprise-grade Machine Learning Tool*. AI Oodles. https://artificialintelligence.oodles.io/blogs/tensorflow-vs-h2o/

Martin, N., Verdonck, L., Caris, A., & Depaire, B. (2018). Horizontal collaboration in logistics: Decision framework and typology. *Operations Management Research*, *11*(1–2), 32–50. https://doi.org/10.1007/s12063-018-0131-1

Martin, R. C. (1995). *Designing Object-Oriented C++ Applications*. Prentice-Hall.

McFarlane, D., Giannikas, V., & Lu, W. (2016). Intelligent logistics: Involving the customer. *Computers in Industry*, *81*, 105–115. https://doi.org/10.1016/j.compind.2015.10.002

McFarlane, D., Giannikas, V., Wong, A. C. Y., & Harrison, M. (2013). Product intelligence in industrial control: Theory and practice. *Annual Reviews in Control*, *37*(1), 69–88. https://doi.org/10.1016/j.arcontrol.2013.03.003

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

MongoDB, Inc. (2020). *MongoDB: The most popular database for modern apps*. MongoDB. https://www.mongodb.com

Muñoz-Villamizar, A., Montoya-Torres, J. R., & Vega-Mejía, C. A. (2015). Non-Collaborative versus Collaborative Last-Mile Delivery in Urban Systems with Stochastic Demands. *Procedia CIRP*, *30*, 263–268. https://doi.org/10.1016/j.procir.2015.02.147

Netflix/zuul. (2020). [Java]. Netflix, Inc. https://github.com/Netflix/zuul (Original work published 2013)

Nisan, N. (Ed.). (2007). *Algorithmic game theory*. Cambridge University Press.

OAuth 2.0. (2020). Password Grant. *OAuth 2.0 Simplified*. https://www.oauth.com/oauth2-servers/access-tokens/password-grant/

Okholm, H., Thelle, M., & Möller, A. (2013). *E-commerce and delivery*.

Oracle. (2020). *MySQL*. https://www.mysql.com/

Özener, O. Ö., & Ergun, Ö. (2008). Allocating Costs in a Collaborative Transportation Procurement Network. *Transportation Science*, *42*(2), 146–165. https://doi.org/10.1287/trsc.1070.0219

Pan, S., Trentesaux, D., & Sallez, Y. (2016, October). Specifying Self-organising Logistics System: Openness, intelligence, and decentralised control. *SOHOMA'16 Workshop on Service Orientation in Holonic and Multi-Agent Manufacturing*. https://hal.archives-ouvertes.fr/hal-01389875

Passport.js. (2020). *Passport.js*. Passport.Js. http://www.passportjs.org/

Perry, D. E., & Wolf, A. L. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, *17*(4), 40–52. https://doi.org/10.1145/141874.141884

PostgreSQL. (2020). *PostgreSQL: The world's most advanced open source database*. https://www.postgresql.org/

RabbitMQ. (2020). *Messaging that just works—RabbitMQ*. https://www.rabbitmq.com/

Redis. (2020). *Redis*. https://redis.io/

Richardson, C. (2019). *Microservices Patterns: With Examples in Java*. Manning Publications.

Riel, A. J. (1996). *Object-Oriented Design Heuristics*. Addison-Wesley Professional.

Riggins, J. (2019, July 4). The Rapid Rate of Container Adoption. *The New Stack*. https://thenewstack.io/the-rapid-rate-of-container-adoption/

Roland Ortt, J. (2010). Understanding the Pre-diffusion Phases. In J. Tidd, *Series on Technology Management* (Vol. 15, pp. 47–80). IMPERIAL COLLEGE PRESS. https://doi.org/10.1142/9781848163553_0002

Rosenthal, R. W., & Wang, R. (1996). Simultaneous Auctions with Synergies and Common Values. *Games and Economic Behavior*, *17*(1), 32–55. https://doi.org/10.1006/game.1996.0093

Russo, F., & Comi, A. (2011). Measures for Sustainable Freight Transportation at Urban Scale: Expected Goals and Tested Results in Europe. *Journal of Urban Planning and Development*, *137*(2), 142–152. https://doi.org/10.1061/(ASCE)UP.1943-5444.0000052

Shahin, M., Ali Babar, M., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, *5*, 3909–3943. https://doi.org/10.1109/ACCESS.2017.2685629

Shaw, M., & Garlan, D. (1996). *Software Architecture. Perspetives on an Emerging Discipline*. Prentice-Hall.

Spring Cloud Gateway. (n.d.). Retrieved October 5, 2020, from https://cloud.spring.io/spring-cloud-gateway/reference/html/

Spring Security. (n.d.). Retrieved September 29, 2020, from https://spring.io/projects/spring-security

Taniguchi, E., & Thompson, R. G. (2002). Modeling City Logistics. *Transportation Research Record*, *1790*(1), 45–51. https://doi.org/10.3141/1790-06

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

TechTarget. (2019). *What is SOAP (Simple Object Access Protocol)?* SearchAppArchitecture. https://searchapparchitecture.techtarget.com/definition/SOAP-Simple-Object-Access-Protocol

The Apache Software Foundation. (2020a). *Apache Hadoop*. https://hadoop.apache.org/

The Apache Software Foundation. (2020b). *Apache Shiro*. https://shiro.apache.org/

van Ommeren, C. R., Fransen, R. W., Pingen, G. L. J., van Leeuwen, C. J., Paardekooper, J. P., & van Meijeren, J. C. (2020). *Exploring Possibilities of Letting Vehicles Plan and Organise Transportation Themselves*. 21.

Vlot, T. S. (2019). *Power to the parcel: A method for self-organisation in last-mile parcel delivery*. TU Delft.

Waehner, K. (2019, November 8). IoT Live Demo—100.000 Connected Cars with Kubernetes, Kafka, MQTT, TensorFlow. *Kai Waehner*. https://www.kai-waehner.de/blog/2019/11/08/live-demo-iot-100-000-connected-cars-kubernetes-kafka-mqtt-tensorflow/

Waehner, K. (2020, February 6). The Rise Of Event Streaming – Why Apache Kafka Changes Everything. *Kai Waehner*. https://www.kai-waehner.de/blog/2020/02/06/rise-of-event-streaming-why-apache-kafka-changes-everything/

Witt, B. I., Baker, F. T., & Merritt, E. W. (1993). *Software Architecture and Design: Principles, Models, and Methods*. John Wiley & Sons, Inc.

Wu, Y.-C., Chen, T.-F., & Lin, S.-K. (2020). Risk management of deposit insurance corporations with risk-based premiums and credit default swaps. *Quantitative Finance*, *20*(7), 1085–1100. Scopus. https://doi.org/10.1080/14697688.2020.1726437

XenonStack. (2020, January 29). Service-Oriented Architecture vs. Microservices | The Comparison. *XenonStack*. https://www.xenonstack.com/insights/service-oriented-architecture-vs-microservices/

Xu, L. D., Xu, E. L., & Li, L. (2018). Industry 4.0: State of the art and future trends. *International Journal of Production Research*, *56*(8), 2941–2962. https://doi.org/10.1080/00207543.2018.1444806

Zhu, H. (2005). *Software design methodology*. Elsevier Butterworth-Heinemann.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Appendix A. Questionnaire for Developers

I administrated a questionnaire for 4 developers at Prime Vision the 7th of July of 2020. The questionnaire had several characteristics for a logistics planning system, and also included a final question on their experience with microservices. These developers eventually became the expert panel who helped verify the software architecture in Chapter 7. The Questionnaire had a ranking system that went from 1 (not important) to 4 (very important).

Characteristic	Developer 1	Developer 2	Developer 3	Developer 4
Compatibility with current operational philosophy (client side)	2	2	2	1
Compatibility with current operational philosophy (client side)	2	2	2	2
Capacity to integrate multiple carriers under 1 platform	3	4	4	3
Ability to ease multi-modal transport operations	3	3	3	4
Availability (up-time of the service)	4	4	4	4
Modularity	4	4	4	4
Modifiability	4	4	4	4
Portability (from one software/hardware platform to another)	3	3	4	4
Performance and Efficiency	4	4	4	4
Security	4	4	4	4
Use of Standard Languages/Elements	2	2	3	4
Testability	3	4	4	4
Conceptual integrity	3	3	3	3

Question	Developer 1	Developer 2	Developer 3	Developer 4
How familiar are you with Microservices?	I worked with them in a large-scale project	I tried them in a small personal project	I tried them in a small personal project	I watched some online lectures/webinars

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Appendix B. Interviewee A. Manager Interview at Prime Vision 1

A semi-structured interview was carried out with a product manager at Prime Vision on July 15th, 2020. The interview was carried out with the support of a questionnaire that listed requirements and attached a ranking system to them. This ranking system went from 1 (not important) to 4 (very important). This product manager has experience with current distribution systems offered by Prime Vision to its Logistics customers. The following is the full transcript of said interview.

Q: *What is Prime Vision's current scope with its customers in last-mile delivery operations?*

A: Currently, Prime Vision's products offer coding inside the sorting centers, creating barcodes with dimensions, direction, etc. We offer the SBS system, that chooses which sorting centers the parcels should go to (macro level) and which vehicles to use (micro level). We also offer consultancy in order to optimize warehouse utilization (mezzo level).

Prime Vision's current offerings are about warehouse distribution and planning, but not exactly last-mile delivery, as the route creation is often done by another solution at the depot, offered by another software company to our customers, called MEDEA. Some of our customers are migrating to a platform called OOM. Our customers, however, are open to receiving offers for *systems of this sort.*

Q: *How would you consider the openness to new practices of Prime Vision's customers?*

A: Usually top management is willing try new systems out, but at the operational level, it is often very difficult to change things for warehouse workers and drivers, since they are used to that way of working. The key challenge for us is to find a starting point where we could facilitate a mindset shift for a new product, while not altering operations too much, at least in the beginning.

Q: *If Prime Vision goes into the Planning System market, what characteristics would be important for Prime Vision?*

- Compatibility with current hardware/software (3) and current operational philosophy (3).

It is important to work with the systems that the service providers have in hand already, especially the hardware, so that the entry barrier to the technology is not too high. When it comes to operational philosophy, changing operations is always considered a hassle, at least to drivers, depot workers, and other delivery operators. At the same time, you do not want an entirely compatible system because that might compromise the concept that is being brought forward. So, we need to facilitate the change of mindset of our customers. To do this, we could use particular cases where we can implement the concept of SOLiD so that operators see the product's potential, such as evening delivery, where demand is not too high and time constraints are not as stiff.

- Capacity to have multiple carriers under the same platform (3)

Having the capacity to integrate multiple carriers under one system is important, but it is not a priority on the early stages of development. Given the contacts we have in the sector, we could probably sell a planning system to several Carriers. If the platform is agnostic, then we have a product with great business potential.

- Capacity to ease multi-modal transport operations (4)

Some of our customers have been looking to integrate other modes of transport into their delivery system, so we need to offer a tool that allows them to have that, and maybe even swap between those modes of transport.

- Availability (4)

Availability is always important for any modern software product. Again, we should not focus it for the proof of concept, but for the final software product.

- Modifiability (4) and Modularity (3)

If we have a tool that has a main identity and way of doing things, it would be interesting that this tool is modifiable enough for our customers. Some customers have different needs, and we need to be able to change the product

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

for each of them, or at least make their systems compatible with our product. This leads us to think of creating Software Development Kits (SDKs) that connect their system with ours.

Currently, drivers get a list of parcels that they need to deliver on that day, if there was a way of transferring parcels from one driver to another, that could ease peak time operations.

From a developing point of view, modifiability is the goal, and modularity is then the means to achieve it. Furthermore, our clients are having trouble calculating the routes in the mornings, because the route calculation cannot start until the last package is sorted, which causes great delays and is often taking 2 hours of operations every day. Tackling these challenges would add great value to our customers, which focuses the performance of a hypothetical planning system.

Automatic deployment is also very important. We strive to develop using CI/CD.

- Portability (3)

We need to host many different services and carriers in a platform that hosts the SOLiD algorithm, so a cloud solution that can include these would probably be a good idea and solve most of the portability issues that could present themselves.

- Performance and Efficiency (4)

We need instant decision making, that would be the game changer for our customers. The current system needs to wait about 2 hours in the depot every morning to finish all the routes for all vehicles. If we could reduce this processing time, we could make a big impact for our customers.

- Security (3)

Current standard practices for software development take care of most security concerns.

- Use of standard languages and elements (2)

Standard programming languages are important for developers to communicate with each other, but they should not be a design focus, as they tend to understand one another either way.

- Testability (4)

We strive to develop with the state-of-the-art Continuous Integration/Continuous Development, which takes testability as a pillar to development. Testing should be able to be automated and should take into consideration relationships with other pieces of software.

- Conceptual integrity (3)

When it comes to developing a project, the longer the development cycle, the harder it is to maintain focus. A product champion that could maintain a clear picture of the project would help development tremendously.

Q: *What is meant by CI/CD?*

A: This is a current (best) practice for development called Continuous Integration/Continuous Deployment. This practice seeks to make software applications auto deployable, and development to be continuous, with many small stepwise advances while developing software.

To achieve this, we use solutions like Docker Containers and Kubernetes, which, apart from enabling the previous characteristics, also help insulation of software packages.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Q: *What would you consider constraints for the project, at a company level and otherwise?*

A: Well, the project right now is in its infancy stage, with most reunions focused on the R&D side of things. I think there is a need for a product owner or a product champion, because we need someone that knows the idea behind the project very well and is able to sit down with our customers and start making business. In my experience, for the project to go to the next phase, we need someone able to create a roadmap and push for its implementation, logging progress and setting goals.

Another important constraint would be resources related, as our staff is usually very busy with many developments. Therefore, finding time to develop completely new platforms would be very challenging. As things stand, I do not think Prime Vision can spare dedicating a full-time team to a project like SOLiD. This also tells us that we need to see what parts of the software our partners can help us with. Collaboration thus takes the front seat for a big new project like this.

Q: *How would you handle monetization for a product like SOLiD?*

A: Prime Vision usually works closely with its customers, and developments are often taken as a join investment, and the price fee for the new products depends on an hourly-based billing. If there would be a standard product, the price depends on the number of hours it took to be developed.

For a new planning system that uses SOLiD, we perhaps should evaluate these two options that we usually utilize, but also, if there is a way of measuring the savings that the system is giving the customer, we could also charge extra based on those savings.

Q: *What do you think of an automated application that charges based on usage, taking in consideration the number of successful requests form the parcels and the transactions made?*

A: I believe that a system like that would work very well, as you see sometimes it is better to come up with a creative way to bill our clients, so that the business proposition is more appealing. A pay-as-you-go system would also have a lower entrance threshold for our customers, which would give the project a higher chance to succeed.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Appendix C. Interviewee B. Manager Interview at Prime Vision 2

A semi-structured interview was carried out with a project manager at Prime Vision on July 22nd, 2020. The interview was carried out with the support of a questionnaire that listed requirements and attached a ranking system to them. This ranking system went from 1 (not important) to 4 (very important). This product manager focuses on the strategic products and innovations for the medium- and long-term plans for Prime Vision. The following is the full transcript of said interview.

- Q: *What does Prime Vision want to achieve with the SOLiD project?*

If I were to dream, then the real objective of the project is that we would create an algorithm and a platform that enables parcels to make their own decisions. The end goal would be a platform that can connect many parcels, and this does not have to be necessarily focused to a specific algorithm that we need to develop thoroughly. Therefore, we could even think of developing a platform that connects the digital twins of the parcels to several algorithm suppliers. The main objective for Prime Vision would be that we set a standard platform for parcels to connect to, and like an *App Store,* they could connect to whichever algorithm and transport provider they like. We need to focus on this connectivity element more than the algorithm, at least at first.

-Q: *Would this mean that the system ought to be agnostic?*

Yes, indeed it will be important to us that multiple providers can connect to the system. The bigger the scale the bigger the benefits, so we need to aim towards that.

- Q: *If Prime Vision goes into the Planning System market, what characteristics would be important for Prime Vision?*

- Compatibility with current hardware/software (2) and current operational philosophy (4).

Compatibility with current hardware/software systems is not so important because we can expect to change many things, but we do need a bridge between current operations and the future innovation. Hardware investments in assets like roller cages and sorting machines are very important for any company, so we should not look to have a system that requires a complete overhaul. Any innovation starts in the "now", so we need to find some sweet spots where we can use the current systems and leverage them into our advantage for the next generation, but in terms of equipment and also working philosophy and depot operations.

- Capacity to have multiple carriers under the same platform (4)

The main objective is to have an agnostic system that houses many carriers.

- Capacity to ease multi-modal transport operations (4)

While this may not be crucial on the short term, we want to make a system that is able to cope with the future generation of transport. The ability to swap from several transportation modes also increases the potential scalability of the system, especially if we manage to someday include the rest of the delivery chain.

- Availability (4)

The current standard of quality for software expects 24/7 service and uptime.

- Modifiability (4) and Modularity (4)

I believe we should strive to develop a system that adopts to many customer requirements in the development phase, and that we do not have to revisit them for substantial changes after deployment. In essence, it is important to be able to change to changing customer requirements, but the original software should be able to already satisfy most of them. For the development of the software, modularity is crucial, because it allows for a proper separation of work between teams, and eases deployment, testing, and the efficiency of the teams.

I believe that the system, being focused on the parcel, helps modifiability greatly. The top-down approach creates very complex decision trees for all parcels, but the bottom-up approach keeps things simple, which is a huge advantage. The major drawback, however, is the allocation of the computing power and the real-time network. We need to focus then on this software architecture.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

- Portability (3)

We should not be fixed to a particular cloud provider, because it puts some risk onto the venture, so we need to strive to avoid vendor lock-in. However, this is sometimes unavoidable. From a financial point of view, it is always good to have a plan B and a plan C.

- Performance and Efficiency (4)

That is indeed crucial and is the particular point where we can "win the game". The main challenge is to have a real-time decision for each parcel. This is what would give us an edge against other competitors. Current practices seem to have some delays when it comes to starting operations because they must wait until the last parcel to calculate routes. If we can implement this system and have many fast calculations, the potential is tremendous.

- Security (3)

We should always comply with the current security standard for software development.

- Use of standard languages and elements (3)

To ease development and communication between developers, it is always good to use standard language and elements.

- Testability (4)

In line with current quality standards for software development (CI/CD), we always design with testability and deployability in mind.

- Conceptual integrity (4)

We want to be clear on what the project wants to achieve.

- Q: *What would you consider constraints for the project, at a company level and otherwise?*

The first thing is that we are not a huge company that can dispense with resources at will and develop everything from the ground up. Therefore, it is very important that we set up the software architecture that we believe will achieve the wanted characteristics and then begins a phase where we should look to collaborate with our customers to develop them. In this case modularity comes back again as a way of achieving that joint development. This also helps having Buy-in from customers and further understanding their requirements for the project.

We could also consider a consortium for development, and maybe we should keep our options open.

-Q: *How would you consider the openness from Prime Vision's customers to implement new technologies in the last mile?*

The postal market is not too bad when it comes to innovation. At some portions of the operational process, they are willing to be front runners in innovation, but usually there are few funds for it. In conclusion it is also up to us at Prime Vision to come up with a good story and a good reasoning behind our product so that we can bring it to the market.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Appendix D. Interviewee C. Commercial Director at Prime Vision

A semi-structured interview was carried out with the commercial director at Prime Vision on July 31st, 2020. This commercial director has been part of the company since its inception in 2007. The following is the full transcript of said interview.

Q: *Could you give me an overview of Prime Vision's market offerings?*

A: We started out as an Optical Character Recognition software company, and we are still innovating in this area. We have developed many solutions for optimizing barcode reading, handwritten addresses, damaged codes, and labels in many languages. We originally worked with the companies that create optical machines and developed the software for them, but some years ago we started working directly with their customers, which is why we currently work directly with Logistics Service Providers.

Ever since we swapped towards Logistics Service Providers, we accompany our customers in all areas of development. For example, postal services are declining, but they will not disappear for at least ten years. Therefore, you have operations with large, fixed costs that must be minimized, and this is an area in which we can make a difference. In the parcel business, the opposite is true. Volumes are increasing exponentially, so Carriers must upscale and make efficient use of all their assets. We help our customers process over 70.000 parcels per hour with our recognition software.

We offer a sorting decision system, which connects to the conveyor belts in the depots. Another product is video coding, which is a back-up processing system for the initial visual recognition software. It analyzes the video footage and find a match on a separate database. One of our latest products is asset tracking, which helps visualizing where current assets are in real-time, be it vehicles, roll cages, staff, parcels, and more. Our customers had many challenges in terms of operations, especially for the need to scale up and down. Thus, automation of certain key processes became valuable. This led to our expansion into the hardware market, and we currently innovate with robots and mechatronics.

Q: *What is the business model that you normally use at Prime Vision?*

A: Originally, we used to make big projects for these optical solutions companies, and we still have many projects, but in the last few years we have tried to move our business model towards services. Nowadays, 35% of our income comes in the form of services, and we want to continue offering subscription-based products that also ensure long-term cash flow for the company. Because we offer products to many companies across the globe, we can also leverage the installed base we have, and offer the maintenance and update of our software products as a 24/7 service.

We also innovate along our customers, because this allows us to test Minimum Viable Products and make Proofs of Concepts, important elements for breakthrough technologies. Including our customers in the development process also pools the risk of a new venture and increases the chances that said customer will acquire and use a product. It also helps our long-term relationship with these customers because they have had a personal interaction with our staff, and it has been positive, they trust us and feel more open towards working with us. I would consider this to be a valuable intangible asset of Prime Vision.

Q: *Is Prime Vision used to developing agnostic systems?*

A: Yes, indeed. In fact, we were able to transition from the optical machinery manufacturer's market to the postal and Carrier market because we were always designing agnostic software that worked on most machines. We have kept that as a staple of quality for our products and we strive to continue working in this manner, as it gives us more flexibility and widens our market potential.

Q: *Would Prime Vision be open to dive into the last-mile delivery market?*

A: I believe that we should definitely investigate expanding our current offerings. We have many contacts in the postal and parcel delivery market, and we should capitalize on our relationships to come up with new solutions. Right now, our customers utilize other products to calculate routing in the last mile and there are many operational challenges in that area. Our sorting system decides which parcels to sort from the large hubs to the smaller depots,

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

but if we could capture the last leg of delivery in our product range, that would improve our positioning in the market considerably.

Q: *If we were to develop a planning system for the last-mile of delivery, should we tailor-make it for particular Carriers or should we create it agnostically?*

A: I think it needs to be agnostic, our business model focuses agnostic systems so that we can duplicate offerings and reach as many customers as possible. Now, it is possible to cater products to customers, but we need to design with modularity in mind.

Q: *When designing a planning system, it is possible to focus the Carrier company that delivers the parcel or the end customer who receives it, who do you think should be focused in the case of the SOLiD project?*

A: Our clients can only be successful if they solve their customer's problems. This is the major factor that separates giants like Amazon from other retailers. Therefore, it is crucial to understand what the end costumers wants. This does not mean we can forget about our immediate customers, but we need to understand the power of customers in the market. For example, PostNL had to change its operations to cater specifically to Bol.com. They have created a special grouping of parcels at the warehouses of Bol.com that has a cutoff time of 2AM. This way they pick up their parcels until the end of the day and guarantee special sorting the next day. This was only possible because Bol.com had both the volume and market power to justify an operational change in PostNL.

We should cater to our customers (the Carriers), so that we can sell our products, but the long-term success can only come by focusing the customer experience and the customer needs. Therefore, we need to balance these two out.

Q: *What would be the ideal outcome for Prime Vision from the SOLiD project?*

A: A good scenario would be if we could offer a product that is valuable to our delivery operators. However, the best scenario would be if we could provide a platform that focuses the end customer and offers all delivery operators as options for the end customer. Because the game is won through the best customer experience. If we offer a product that eases depot operations for PostNL or any other provider but does not deliver the product in the specified time window, the customer will have a bad experience and will not use the system again. So, the fundamental focus should be on the end customer. Naturally, we cannot get there at the beginning, but we should begin development with this final vision in mind.

Q: *How would you handle monetization for a product like SOLiD?*

I think we should focus on offering a service, and steer away from hefty projects if possible. If we have some sort of pay as you go system, this would line up with our current financial objectives nicely, and it would lower the entrance barrier to new customers.

Q: *What do you think of an automated application that charges based on usage, taking in consideration the number of successful requests form the parcels and the transactions made?*

I think that is exactly the way we should handle monetization for this project. We should be aware of initial costs, however, as we should investigate joint development and risk sharing at the early stages of development, but then stabilize into a full-service platform in the long term.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Appendix E. Interviewee D. Project Manager Interview from Dutch Carrier 1

A semi-structured interview was carried out with a senior project manager from a major Dutch logistics company on July 22nd, 2020. The interview was carried out with the support of a questionnaire that listed characteristics for a logistics planning system and attached a ranking system to them. This ranking system went from 1 (not important) to 4 (very important). A final rank of 5 (indispensable) was utilized to describe *constraints*, or characteristics that would be absolutely necessary for a planning system. The following is the full transcript of said interview.

Q: *Has your company delved into "Dynamic routing"? Could you describe current projects?*

A: Yes, let us begin with standard routing. Our normal operations have a first wave of sorting in the evening and a second wave of sorting in the morning of the next day. After the second wave, parcels go directly to the vehicle that will take them to their destination. Our drivers know how to deliver these parcels; in our experience, it is best to leave the inner-city routes up to them, and not any planning software. The reason for this is that the volumes that we handle are around 1 million parcels per day, and the Netherlands has about 7 million households. Therefore, the standard distance from one stop to the next is about 200 meters.

However, for our other product, evening delivery, we have a much lower volume of parcels. This allows us to use a different system that calculates routes daily after the last wave of receiving parcels, which happens on the evening of the previous day. Thus, the parcels are known from the day before. The system then calculates a more refined route for all vehicles daily, however I would not call it optimal. Thus, the volume of parcels changes the approach that we can take, at least on the short term.

Q: *Does your company often develop their own planning systems, or does it normally outsource these?*

A: Some years ago, we outsourced almost all our routing systems. In the latest years, however, a colleague of mine found some open source tools like Street Map and the Google R tools that we were able to tweak for our own needs, and the end result was the granular planning system that we currently utilize for evening delivery. This system is quite automated, it updates every involved party automatically from the data of the Track & Trace system, and all routes are presented to every depot, all the while the software was almost free, developed in-house.

Q: *If your company were to acquire a new Logistics Planning System, what characteristics would be desirable in it?*

- Compatibility with current hardware/software (2) and current operational philosophy (4).

We sometimes do a complete overhaul of our software systems, so we are open to changing those elements of our delivery chain. Hardware is also flexible, because we have done medium term replacements and we are aware that both hardware and software eventually become obsolete.

Operational Philosophy is more complicated because you cannot change it in an instant. There should always be a gradual change between two ways of working, and that should be done in parallel, both the old and new ways simultaneously, so that the new way is verified to work, and that the workers assimilate the change more easily.

- Ability to enable multi-modal transport (2).

We do not think that a single planning system would encompass the whole delivery chain, so we are mostly swapping from truck to truck. It would be interesting however on the last mile where vehicle options are broader.

- The system should be available 24/7 (5).

Indeed, any modern system that we acquire today is expected to be up and running constantly.

- The system should be reliable (be able to cope with accidents, vehicle breakdown, traffic jams) (3).

Yes, not all things can be thought of ahead of time and there will always be a degree of uncertainty. A system that can cope with these changes becomes more valuable.

- The system should minimize cost of delivery (4)

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Any modern system should minimize costs. While some systems minimize distance, in the logistics field, this is often quite suboptimal for operators. This occurs because waiting to a full truckload (FTL) would be more economic than reducing the distance between various vehicles. Any planning system therefore needs to take these subtleties into account.

- The system should improve capacity utilization (reduce empty-running) (4)
- The system should ease depot operations (4)

This is very important, because depot operations can be quite spiky. In other words, we need to avoid having 100.000 parcels in a particular hour and then 100 for the next six. The alternative, which is to sort the parcels gradually, is what we call *equal flow*. This smoothens operations considerably.

- The system should be secure against digital threats (5-indispensable)

Security must be guaranteed in any modern software system. We usually require our providers to show that they are compliant with modern security standards.

- The system should accurately predict the estimated time of arrival (ETA) (3)

For last-mile delivery, we consider a 2-hour window to be optimal. Our current system allows us to make a prediction of the time window with a granularity of 6 hours **at the time of purchase** in certain retailers like bol.com. This is possible because we have stable routes and a lot of historical data, something that is unique to the Netherlands market. In other markets like Germany, knowing that delivery will be done within 2 days is enough. From a customer experience point of view, this expected time of delivery is a valuable characteristic for a logistics planning system to have.

- The system should be able to track KPIs like distance travelled, number of successful stops, etc. (3)

It is always great to have a way of keeping track of important data, so that we can do statistical analysis later. However, I would say that recording all data could potentially infringe on the privacy of the driver, as it would be visible when they go for a lunch break, for example.

- The data generated by the system should be transparent to my company (5-indispensable).

Well, this is a given, of course it should be transparent to our company.

- The system should have an environmentally friendly option for customers (2).

If our customers have the option to select green delivery, and the system prioritizes this over other options at the risk of higher costs, I think that would add value to our customers. However, I would not consider it indispensable.

Q: *Are there any other characteristics that you would consider desirable in a logistics planning system?*

A: I would point towards the idea of *equal flow*. Managing peaks is extremely costly and having a system that takes that into account would be very worthwhile. For example, it is much cheaper to hire 200 10-hour shifts than it is to hire 1000 2-hour shifts. Allowing gradual dispensation of parcels and avoiding rush hours minimizes costs considerably.

Another point would be aiding realism on the micro level, if the system can for example be accurate in its usage of three-dimensional space with the parcels' sizes, that would have a great impact on day to day operations.

Q: *Would a system that allows for intra-day pick-up help achieve this* equal flow*?*

A: That would definitely help, but on the way that our company is organized right now, it would be challenging to organize that. If you increase the range of times at which vehicles can do pick up and drop off, this will help equal flow.

Q: *Would your company be open to participate in a collaborative system that includes many carriers under one auctioning platform?*

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

A: I think it would not be something that my company would like to participate in. I would only see it in a sub-contractor scenario, were our subcontractors bid against each other, because we already use a similar system, and that competition benefits our company. In the carrier vs. carrier scenario, the benefits are reaped by the customer and strategically, it would probably be a bad decision to willingly enter such a system.

We have some large retailer customers that already work with several carriers, but we have one to one contract with these customers that stipulate a particular price per parcel according to volumes and other factors.

An open system like that would be unpredictable for us, since we would not be able to guarantee a steady flow of deliveries, which goes against the same idea of *equal flow*.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Appendix F. Interviewee E. Project Manager Interview from Dutch Carrier 2

A semi-structured interview was carried out with a senior project manager at a major Dutch logistics company on July 30th, 2020. The interview was carried out with the support of a questionnaire that listed characteristics for a logistics planning system and attached a ranking system to them. This ranking system went from 1 (not important) to 4 (very important). A final rank of 5 (indispensable) was utilized to describe *constraints*, or characteristics that would be absolutely necessary for a planning system. The following is the full transcript of said interview.

Q: *Could you describe the last-mile delivery process at your company?*

A: Everything begins in a yearly meeting that defines several delivery zones, grouping areas per postal code. From these delivery zones, we define static tours that will be used for delivery later. We seek to balance these tours for driving time, number of parcels to be delivered, picked up, etc. We upload these routes in a sorting table, which is the input for our sorting system. This sorting occurs every day, from the evening until 6:30 in the morning the next day. When all parcels are sorted, the customers receive a notification announcing that the parcel will be delivered the next day, with no time stamp.

There is a designated area in the depot for each tour, where parcels are transported by conveyor belts. Then vehicles arrive the morning and the loading procedure begins. The drivers order the parcels in the order that they will deliver them. Parcels are scanned one last time when loaded into the vehicle, and this is the input for the route-calculation software that is done by a mobile device. This tool is different from the tool that divides the parcels into the predetermined tours. When this tool finishes the calculation, messages are sent to the customers, giving them a 1-hour window of expected time of arrival.

Q: *How long does it take for deliveries to begin in a normal day of operations?*

A: From the moment vehicle drivers arrive at the depot and the moment they exit the depot; the loading and route calculation process lasts about 90 minutes. They arrive to their respective loading areas, designated by the aforementioned tours, and load parcels in order of delivery.

Q: *Are there any challenges with the current system?*

A: Indeed. First of all, anything that lessens the time to begin delivery every day is an important factor for efficiency. Another point is that many of our customers are return customers that we send parcels to periodically. This gives us an opportunity to create routes based on historical data to work around these important clients. If we had a tool that automated this route generation, we could reasonably predict the estimated time of arrival (ETA) at the time of purchase a day in advance, which would be great for our customers.

Q: *How visible are your assets (parcels and vehicles) throughout a normal day of operations?*

A: The parcels are observable through the log of the places where they have been scanned. Every time they are scanned, an event is created, and from that log we can create what we call the *Parcel's Lifecycle*. This comes from the gathering of a lot of data, which we utilize to do analytics later, always looking to optimize this *parcel lifecycle*. In general terms, we do not have visibility of the parcels in the points between scans, but we can always look up the vehicle that holds it, although this system is separated from the parcel tracking system.

Q: *Does your company often develop their own planning systems, or does it normally outsource these?*

A: Most of the tools that we utilize are outsourced. We usually adapt them to our needs, but in general the products come from a provider. Our expertise is more focused on the orchestration of the process and the connecting of these tools and system. We can cover the first and second line of support for these systems, but when thinking of developing new features, we turn to our providers.

Q: *How would you consider the openness from your company to test new planning systems for delivery?*

A: I would say that we are very open to alternatives, and this is mostly because we acknowledge that we are in a growth market. This is more prevalent in the Netherlands, where our company is growing faster than the market average. We are building 1 to 2 new depots every year, and we are trying to innovate with every new depot that we

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

inaugurate. My belief is that you cannot innovate if you are not open to discontinue your current systems, so we are always open to new options and pilots of new technologies.

Q: *If your company were to acquire a new Logistics Planning System, what characteristics would be desirable in it?*

A: While we are open to new options, we certainly favor systems that take advantage of our current infrastructure. Therefore, we value modular systems very highly, or the most approximate thing to a "plug and play" product. Another key issue is that we do not like to buy many small products from different providers. Thus, we favor providers that can offer a wider range of products. A good example would be a provider that could help us throughout the complete last mile delivery, instead of a supplier specialized in barcode scanners.

Another important characteristic when considering a new last-mile delivery planning solution is the issuing company. We have worked with startups in the past, and we still do it because it helps innovation tremendously. However, we have seen that business continuity is risked when dealing with incubators and other younger firms. Therefore, our company favors more stable partners that have proven technologies. We must strive to strike a balance between innovation and stability. If a particular technology is very promising, however, we would be open to working with a startup.

(at this point the questionnaire was brought to discuss further characteristics)

- Compatibility with current hardware/software (3) and current operational philosophy (2)

  I believe that we cannot focus compatibility with our current operational philosophy without jeopardizing our capacity to innovate. We understand that change must be done gradually, but we do not need to focus this on a new product.

- Ability to enable multi-modal transport (3).

  Flexibility with modes of transport would add value to last-mile operations.

- The system should be available 24/7 (5).

This is a regular requirement for any system that we acquire.

- The system should be reliable (be able to cope with accidents, vehicle breakdown, traffic jams) (4).
- The system should minimize cost of delivery (3)

For any planning system, we look to optimize costs over distance.

- The system should improve capacity utilization (reduce empty-running) (3)

If the focus is on the last mile, this particular factor is not indispensable. It would be a nice addition, however. We currently combine drop-off with new pick-ups, but clients are often ready for pick-ups only in the evenings. So, this is a current challenge for our system where there is room for improvement.

- The system should ease depot operations (1)

In our current framework, depot operations and last-mile delivery are different areas of expertise: the system that handles sorting towards depots and the system that sorts parcels to vehicles are different. Therefore, it is not important for a last-mile solution to ease depot operations.

- The system should be secure against digital threats (4)

This is a normal requirement for any modern product. We expect providers to be acquainted with current security standards or some equivalent.

- The system should accurately predict the estimated time of arrival (ETA) (5)

This is a point that directly affects our customer experience, so it is indispensable that it is handled in the best possible manner. We currently handle 1-hour intervals, but we want to reduce it to 30 minutes.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

- The system should produce data to generate and analyze KPIs such as distance travelled, number of successful stops, etc. (3)

We want to achieve full traceability, so data generation is important. At the moment, we only have visibility at the scan points. If this could somehow be improved, it would be welcome.

- The data generated by the system should be transparent to my company (4).

We do not want to have any black boxes.

- The system should be environmentally friendly (4).

This would add customer value, but it is not a must have.

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Appendix G. Preliminary List of Operations

After defining the steps the system follows in Chapter 4, I defined the preliminary operations for each of the classes of Figure 5-6. This is a preliminary list of operations because the services had not been defined yet, a step done later in Section 5.2.2. The following table includes the assignment of operations to each of the classes of Figure 5-6. Therefore, some of the classes in this list are eliminated after defining the services of the system.

Actor	Story	Command	Pre-Conditions	Result	Description	Collaborators
Vehicle Computer	Create Depot Connection Request	Create Depot Connection Request	Vehicle has capacity for more parcels	Depot Connection Request (VehicleID, position)	The vehicle asks the Depot Platform to connect it to nearby depots	Vehicle Information: Load Vehicle Information, Vehicle Positioning System: Load Vehicle Position
Depot Platform Computer	Connect Vehicle to Depot	Connect Vehicle to Depot	Depot Connection Request (VehicleID, position)	Vehicle is connected to Depot Vehicle Twin	By connecting a vehicle to a depot, the vehicle can see the parcels' transport requests	Depot List - Load Depot Locations
Depot List	Load Depot Locations	Load Depot Locations	Depot Platform-Connect Vehicle to Depot	Depots' locations are loaded	The Depot platform requires the position of depots to find relevant depots for requesting vehicles	-
Parcel Information	Store Parcel Information	Store Parcel Information	-	Parcel information is can be loaded by the Parcel Computer	Initial step for the parcel, its information is stored so that the Parcel computer can Request Transport	-
Parcel Computer	Create Transport Request	Create Transport Request	Parcel status: unassigned	Transport Request, state: "pending"	Normal transport request	Parcel Information - Load Parcel Information
Parcel Computer	Create Transport Request	Create Transport Request	Parcel status: assigned	Transport Request, status: "pending"	Transport request when parcel already is in a vehicle	Parcel Information - Load Parcel Information Vehicle Positioning System - Load Vehicle Position
Depot Digital Twin	Display Transport Request	Display Transport Requests	Unassigned Transport Request status: pending, or Assigned Transport Request status: pending	Transport Request is visible to connected vehicles	The depot displays the transport requests of the parcels within it	Depot Platform-Connect Vehicle to Depot
Vehicle Computer	Evaluate Transport Request	Evaluate Parcel	Unassigned Transport Request status: pending, or Assigned Transport Request status: pending	Parcel is feasible (yes/no)	First check a vehicle does when reading a transport request by a parcel	Depot Digital Twin: Display Transport Requests
Vehicle Computer	Create Bid	Create Bid	Parcel is feasible (yes)	Potential Itinerary - Bid (VehicleID, overcost, emissions, ETA)	Route creation for a vehicle and the creation of a Bid from the best route	Vehicle Information: Load Vehicle Information, Vehicle Positioning System: Load Vehicle Position, Depot

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

						Digital Twin: Display Transport Bids
Vehicle Information	Load Vehicle Information	Load Vehicle Information	Vehicle Computer: Evaluate Parcel, Create Bid	Vehicle Computer can now perform Create Bid	The Vehicle's computer needs to know the vehicle's details to create bids	-
Vehicle Positioning System	Load Vehicle Position	Load Vehicle Position	Vehicle Computer: Evaluate Parcel, Create Bid, Parcel Computer: Assigned Transport Request status: pending	Vehicle's latest position is loaded	Several actors need the vehicle's position to perform their actions	-
Depot Digital Twin	Display Bids	Display Bids	Bid (VehicleID, overcost, emissions, ETA)	Bids are visible to requesting parcel	The depot displays the bids of the vehicles connected to it	Vehicle Computer: Create Bid, Depot Platform: Connect Vehicle to Depot
Parcel Computer	Select best bid	Select Best Bid	One or more Bids (VehicleID, overcost, emissions)	One Bid is accepted, all other bids are Rejected, change Parcel status to "assigned", change Transport Request status to "finalized"	The parcel selects one bid and rejects all others. The Depot Digital Twin helps the parcel communicate	Depot Digital Twin: Display Bids, Display parcel selection/rejection
Parcel Computer	Request Assignment to Loading Dock	Request Assignment to Loading Dock	Parcel status is "assigned" or "en route" Assigned vehicle is connected to the Depot Parcel is connected to the Depot	Depot Digital Twin receives the request and begins Check Capacity Operation	The parcel requests to be assigned to a loading dock so that it can be picked up or dropped off by the incumbent vehicle	If parcel is "assigned", none If parcel is "en route", then: Vehicle Positioning System: Load Vehicle Position (at a depot)
Depot Computer	Check Capacity of docking stations	Check Capacity	Parcel status "assigned"	Verifies which loading docks have capacity for the parcel whose status changed from "unassigned" to "assigned"	When a parcel accepts a bid, the depot must check which loading dock can accept the parcel	Depot Information: Load Capacity of Loading Docks, Parcel Information - Load Parcel information (dimensions)
Depot Information	Load Capacity of Loading Docks	Load Capacity of Loading Docks	Depot Computer- Check Capacity	Loading Docks' latest capacity values are loaded		-
Depot Computer	Assign Parcel to loading dock	Assign parcel to loading dock	Depot Computer- Check Capacity, Parcel status: "assigned"	Assigns the parcel to a loading dock, vehicles are informed, update Loading Dock's capacity value	After checking the capacity of its loading docks, the depot can assign the parcel to one of them and let vehicles know.	

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Depot Computer	Initiate transport to loading dock	Order Transport to Loading Dock	Depot Computer- Assign Parcel to loading dock	Publish transport order	The Depot Digital Twin orders the transport of the parcel to its assigned loading dock	-
Parcel Computer	Request Drop Off	Request Drop Off	Assigned Transport Request status: Accepted	Request Drop Off	Drop off request when the parcel changes vehicles	Vehicle Computer – Create Bid
Parcel Wallet	Pay bid	Pay bid	Vehicle Location= Parcel destination (Parcel status is "delivered"), or Parcel Computer - Request Drop Off	Overcost is paid to vehicle, parcel is dropped off.	Parcels pay the vehicle at drop-off, be it at a depot where the parcel will transfer, or at the parcel's destination	Vehicle Computer - Create Bid
Vehicle Wallet	Vehicle Receives payment	Receive Payment	Parcel Wallet: Pay bid	Change Parcel status to "delivered" (when delivered), or "assigned" (when assigned to a different vehicle)		
Parcel Log	Save parcel interaction	Log interaction	Any change in status, from the parcel and its requests, any message sent to the Depot Digital Twin	All events are stored in the log	All significant events in the parcel's lifecycle are retrievable	
Vehicle Log	Save vehicle interaction	Log interaction	Any change in status, from the vehicle and its bids, any message sent to the Depot Digital Twin and Depot Platform	All events are stored in the log	All significant events in the vehicle's lifecycle are retrievable	

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Appendix H. List of Operations

To define APIs, we must assign system operations to the defined services. This is the formal finalization of the first decomposition in services of the Self-Organizing Logistics Planning System. The following table includes the assignment of operations to done immediately after the first separation in microservices. Some of the services in this table were eliminated after the application of architectural patterns. Several services have different triggers and require different collaborators. These are marked with scenarios (a), (b), (c).

Actor	*Command*	*Pre-Conditions*	*Result*	*Description*	*Collaborators*
Parcel Information Management System	Store Parcel Information	-	Parcel info can be loaded	Saves data (Parcel ID, preferences, earliest pick-up time, etc) so that it can be loaded by the parcel computer to create a transport request	-
Parcel Information Management System	Store Parcel Interaction	Any interaction with a Parcel ID	Parcel interaction is stored and retrievable	Event log for the parcel	All other parcel interactions with the same Parcel ID
Parcel Information Management System	Load Vehicle's Itinerary	External party requests Parcel itinerary	Parcel location and itinerary is retrievable	The parcel's location and itinerary can be loaded from the vehicle's itinerary	Bid Creator: - Load bid's Itinerary
Transport Request Service	Load Parcel Information	Parcel Information Management System: - Store Parcel Information	Transport Request Service can now "Create Transport Request"	The Transport Request Service loads the parcel's information to then create a Transport Request	Parcel Information Management System: - Store Parcel Information
Transport Request Service	Create Transport Request	(a) Parcel status is "unassigned", (b) Parcel status is "assigned",	(a) Creates transport request type: unassigned (b)(c) Creates Transport Request type: assigned	Creates Transport Request	(a)Transport Request Service: - Load Parcel Information Depot Information Management Service: - Broadcast Requests (b) Transport Request Service: - Load Parcel Information Depot Information Management Service: - Broadcast Requests Bid Creator: - Create Equivalent Bid
Transport Request Service	Create Assignment to Dock Request	(a) - Parcel status changes from "unassigned" to "assigned" (b) Transfer Evaluation Service: - Parcel Requests Drop-off to original vehicle	Depot Digital Twin is informed and can begin "Assign Parcel to Loading Dock"	When the parcel is assigned or when it found a better transport, it must be loaded/unloaded. This is done at a loading dock. Thus, the parcel requests to be assigned to one.	(a) Parcel Information Management System: - Store Parcel Information (b) Depot Connection Service: - Connect Vehicle to Depot (the parcel can communicate to the depot thanks to the vehicle's connection to it)

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Vehicle Selection Service	Select Best Bid	Depot Digital Twin: - Broadcast Bids (a) Bid Creation Service: - Create Bid (b) Bid Creation Service: - Create Bid, - Create Equivalent Bid	Best vehicle is selected, a message is sent to this vehicle, all other bids are rejected	The parcel selects the best vehicle according to its preferences	-
Transfer Evaluation Service	Request Capacity to Depot	Depot Connection Request Service: - Connect Parcels to depots	Capacity Request is sent to the Depot Information Management Service	When the vehicle goes through a depot, the parcels within can ask the depot if it has capacity for the parcel. The parcel can then evaluate a transfer	-
Transfer Evaluation Service	Request Equivalent Bid	Depot Information Management Service: - Load Depot Capacity: yes	Parcel sends a request for an equivalent bid to the Vehicle's Bid Creator	The parcel needs an equivalent bid from its current vehicle to compare new vehicle's bids to it	Bid Creator: - Create Equivalent Bid
Transfer Evaluation Service	Request Drop off to original vehicle	- Parcel status is "en route", Vehicle Selection Service: - Select best bid	Drop Off request is sent to the vehicle	The parcel asks for drop off when it finds a better vehicle at a depot/transfer point. This also triggers Transport Request Service: Create Assignment to Dock	-
Parcel Wallet Service	Pay bid	- Parcel status changes from "en route" to "delivered"	Vehicle Wallet Service receives payment for transport	The parcel pays the vehicle for transport, finishing the delivery.	Vehicle Wallet Service: - Receive Payment from parcel
Parcel Wallet Service	Pay surcharge	Vehicle Wallet Service: - Create surcharge, - Parcel status changes from "en route" to "delivered"	Vehicle Wallet Service receives payment for extra handling and transport	When the customer cannot receive the parcel in the allotted time window, there is a surcharge for the parcel	Vehicle Wallet Service: - Receive Payment from parcel
Parcel Wallet Service	Pay Depot	Depot Wallet Service: - Create Quote for the Parcel	Depot Wallet Service receives payment for parcel's handling at the depot	After every stay at a depot, the parcel may have to pay the depot for handling	Depot Wallet Service: - Receive payment from parcel
Vehicle Information Management Service	Store vehicle's information	-	Vehicle information can be loaded	Saves vehicle data (Vehicle ID, Delivery Window, Center of Delivery, etc) so that it can be loaded by Bid Creator to create a Bid	-
Vehicle Information Management Service	Load Depots List	Depot Location Service: - Create Depots List	Vehicle knows the locations of depots	The vehicle must know the depots' locations so that parcels can be notified	Depot Location Service: - Create Depots List

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Vehicle Information Management Service	Store Vehicle's interaction	Any interaction with the same Vehicle ID	Parcel interaction is stored and retrievable	Event log for the vehicle	All other parcel interactions with the same Vehicle ID
Bid Creation Service	Load Vehicle Information	-	Bid Creator Service can now Create Bid	-	Vehicle Information Management Service: - Store Vehicle Information
Bid Creation Service	Create Bid	Transport Request Service: - Create Transport Request, Filter Service: - Forward Requests	Bid is published on the depot and is readable by the Vehicle Selection Service	Vehicle creates an offer for the parcel	Bid Creation Service: - Load Vehicle Information, Vehicle Positioning System: - Load vehicle position, Depot Information Management Service: - Broadcast Requests, - Broadcast Bids
Bid Creation Service	Create Equivalent Bid	Transfer Evaluation Service: - Request Equivalent Bid	Bid is published on the depot and is readable by the Vehicle Selection Service	Vehicle creates an adjusted offer for the parcel that is evaluating a transfer at a depot	Vehicle Information Management Service: - Load Depots List,
Bid Creation Service	Store bid's itinerary	Bid Creation Service: - Create Bid, status: pending	The itinerary of each bid is stored and can be applied	The itinerary linked to each bid is stored to be implemented if the bid is accepted	-
Bid Creation Service	Load bid's itinerary	Bid Creation Service: - Store bid's itinerary, - Any Bid status from the Bid Creation Service changes to "accepted"	The vehicle's itinerary is updated, and so is the itinerary of all parcels on-board	The vehicle applies the itinerary that accompanied the bid for the parcel. It then starts to go pick up the parcel.	Bid Creation Service: - Store bid's itinerary, Depot Information Management System: - Broadcast Bids, - Broadcast Requests
Bid Creation Service	Temporarily remove parcel from itinerary	Depot Connection Request Service: - Connect Parcels to depots	Once the parcel is removed, the parcel can initiate an assigned transport request		
Vehicle Positioning System	Load vehicle position	-	The vehicle's position is gathered from a GPS device	The location of the vehicle is known, which is useful for several operations	-
Vehicle Wallet Service	Create surcharge	Handheld device of driver informs of failed delivery	The parcel is added a surcharge for extra handling	If a customer cannot receive a parcel on the estimated delivery window, the vehicle adds a surcharge on its quote	Parcel Waller Service: - Pay surcharge
Vehicle Wallet Service	Receive Payment from parcels	Parcel Wallet Service: - Pay bid	The vehicle receives the bid's value	The vehicle receives payment for the transport of the parcel	-

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Filter Service	Forward Requests	Transport Request System: - Create Transport Request	The filter contacts relevant vehicles with the request	The filter computes a list of relevant vehicles and forwards transport requests to them	-
Depot Connection Request Service	Request Connection to depots	(a) Vehicle status is "inactive" (b) Bid creator: - Apply itinerary - Capacity utilization is below a threshold, (c) Vehicle is passing through a known depot in its itinerary	The vehicle requests connection to depots. The Depot Selection Service can begin	By requesting connection to nearby depots, the vehicle can receive more transport requests and send bids to parcels.	(a) Vehicle Information Management System: - Store Vehicle's information (b) Bid creator: - Apply itinerary, (c) Bid creator: - Apply itinerary, Vehicle Information Management System: - Load Depots List, Vehicle Positioning System: - Load vehicle position
Depot Connection Request Service	Connect Parcels to depots	Depot Connection Service: - Connect vehicle to relevant depots	The vehicle connects the parcels to the depots it is connected to	By connecting parcels to depots, the parcels can evaluate transfers with other vehicles at those depots	-
Depot Information Management Service	Read and update capacity	Parcel Assignment Service: - Assign Parcel to Dock	The depot updates the capacity of its loading docks	By updating its capacity, the depot knows where to assign parcels to. Every time a parcel is assigned to a dock, loaded, or unloaded; this updates itself.	-
Depot Information Management Service	Broadcast Requests	Transport Request Service: - Create Transport Request, Depot Connection Service: - Connect vehicle to relevant depots	The Transport Request is visible to the connected vehicles	By broadcasting the requests of the parcels within the depot, vehicles can read the requests and send bids to the parcels.	-
Depot Information Management Service	Broadcast Bids	Bid Creation Service: - Create Bid, - Create Equivalent Bid Depot Connection Service: - Connect vehicle to relevant depots	The vehicle's bid is visible to the parcels	By broadcasting the connected vehicles' bids at the depot, parcels can evaluate bids and select a vehicle	-
Parcel Assignment Service	Load Depot Capacity	Depot Information Management Service: - Read and Update Capacity, Transfer Evaluation Service: - Request Capacity to Depot	Sends response to the parcel, there is capacity yes/no	If there is capacity for the parcel, then the Transfer Evaluation Service begins its operations	
Parcel Assignment Service	Assign Parcel to Loading Dock	Transport Request Service: - Create Assignment to Dock Request	Parcel is assigned to a loading dock and a message is sent to the vehicle	A parcel needs to be assigned to a loading dock to be (un)loaded. The relevant vehicles are notified, and the Depot Capacity is updated	Parcel Assignment Service: - Load Depot Capacity

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.

Parcel Assignment Service	Order transport to Loading Dock	Parcel Assignment Service: - Assign Parcel to Dock	The Assignment Service orders the transport of the parcel to the assigned loading dock	The parcel is sorted to the assigned loading dock where the vehicle will (un)load it	-
Depot Wallet Service	Create Quote for Parcel	Parcel Assignment Service: - Order transport to Loading Dock	A quote is created for the assigned parcel		-
Depot Wallet Service	Receive Payment from parcels	Parcel Wallet Service: - Pay Depot	Depot receives payment for handling	Depot receives payment for handling	-
Depot Location Service	Create Depots List	-	Creates a Depot List that can be loaded by requesting services. Updates the Depot List if there are new depots.	The Depot list created is useful for several services. This service also updates the list.	-
Depot Selection Service	Load Depots List	Depot Connection Request Service: - Request Connection to Depots	Loads the Depots List.	After loading the Depots List, the Depot Selection Service can begin the Find Relevant Depots operation.	Depot Location Service: - Create Depots List
Depot Selection Service	Find Relevant Depots	Depot Selection Service: Load Depots List	Creates a Relevant Depots List	The relevant Depots list can be loaded by the Depot Connection Service	-
Depot Connection Service	Load Relevant Depots List	Depot Selection Service: - Find Relevant Depots	Loads the Relevant Depots List.	After loading the Depots List, the Depot Connection Service can begin the Connect Vehicle to Relevant Depots operation.	-
Depot Connection Service	Connect vehicle to relevant depots	Depot Selection Service: - Find Relevant Depots	Connects vehicles to the depots in the Relevant Depots List	The Depot Platforms connects vehicles to depots and therefore vehicles can receive requests from parcels within	-

This document is confidential and is not to be distributed without the permission of Prime Vision and TU Delft.