Utrecht University - Faculty of Science

Master's thesis

Improving communication in the Dutch multi-agency emergency healthcare

Applying Enterprise Integration Patterns to cross communication barriers

Author:
Hugo Helder
j.h.helder@students.uu.nl

Daily supervisor:
Christian Snijder
christian@ambusuite.nl

1st Supervisor
Wouter Swierstra
w.s.swierstra@uu.nl

2nd Supervisor:
Inge van de Weerd
g.c.vandeweerd@uu.nl

This thesis is submitted in fulfillment of the requirements for the degree of Master of Science
in
Business Informatics

November 28, 2019

Abstract

In this study it was investigated what architectural design could facilitate an optimal information delivery in the emergency healthcare, what resulted in a communication infrastructure for the emergency healthcare sector. With this communication infrastructure communication between disciplines in the emergency healthcare can be achieved, exchanging information in ways that was not possible before.

It was identified that information in the emergency healthcare sector is often not shared between these disciplines, thus the objective was to enable inter-enterprise communication between these disciplines. Upon investigation of the emergency healthcare sector, it became clear that little to none IT-solutions to enable such inter-enterprise communication are available but are gravely needed to resolve the challenges that the emergency healthcare sector are plaguing nowadays. Challenges consist of but are not limited to insight in bed capacity, insight in medical history for ambulance staff on location, and transfer of medical information from the dispatch center, ambulance, and emergency department to the General Practitioner's (GP) records [26].

The result of this study is therefore a communication infrastructure for the emergency healthcare to enable inter-enterprise communication. The communication infrastructure is created and implemented based on Rozanski's guidelines for software architecture design [31]. The implementation served as an evaluation of the architectural design.

The evaluation showed that the followed methodology yielded the expected software architecture, with a notable role for the perspectives, that in iterations revisited the created viewpoints and adapted them according to the validation results. Thanks to that minimal trade-offs between the design and the implementation were found. The application of Enterprise Integration Patterns proved to provide a solid foundation for a from-theory derived practical approach to incorporate communication infrastructures in software architecture. An analysis of the implementation showed that the implementation satisfies the requirements as provided by the stakeholders, demonstrating that integration can take place as they have desired.

Preface

At the start of this project, I had no notion of the challenges the emergency healthcare sector is facing nowadays and how this public sector is plagued by failing IT-systems, insufficient communication systems and self-centered healthcare disciplines that do not communicate with each other. AmbuSuite, an IT-vendor supplying the emergency healthcare sector with among others electronic trip forms (for ambulances) and hospital software knew this as no other. When I started looking for an internship for my master thesis I was looking for a company where I could make an impact with my master thesis and could do something for the society, as this always was my personal interest. AmbuSuite had seen the communication problems in the multi-agency emergency healthcare sector and were looking for ways to improve it drastically. As an intern in AmbuSuite I could help by researching the communication problems in the emergency healthcare sector and provide an implementation of an IT-solution that enables communication between all healthcare professionals. Now, at the end of this project, I can finally present all efforts of my research of the healthcare sector, it problems regarding communication and an IT-solution in the form of a master thesis. It has been quite a journey and I am proud of this achievement. I would like to take the opportunity to thank some people that have been involved in this journey with me. First and foremost, I would like to thank Christian Snijder, lead developer at AmbuSuite and my daily supervisor, without whom I would not have been able to successfully complete this project. He has kindly devoted his time, was always reachable and involved, and through meetings and feedback, helped improve the quality and analysis of this research. His expertise on the emergency healthcare sector was of great value and his involvement in this project is highly appreciated. I would like to thank my first and second supervisor at the university, Wouter Swierstra and Inge van de Weerd, for granting me this opportunity for this project. They were always available for feedback where necessary and their advice lifted this project to a greater level. Without their thorough analysis and critical mindset I would not have spotted some anomalies in my thesis myself. I would like to thank my friends, who have supported me on this journey by listening to my stories and having discussions on the content and research opportunities. Among them was Rick de Boer, who I would like to thank for reading my manuscript and providing excellent feedback. For one I have my special thanks, Kim Stienstra, for always being there for me and helping me out from her expertise as researcher at Utrecht University. Her involvement and support meant the world to me. With the support of all these special people and with extensive effort, I have managed to perform and write about the biggest research project in my life thus far captured in the thesis laying in front of you; the creation of an inter-enterprise communication infrastructure in the Dutch emergency healthcare sector.

Glossary

This glossary provides an overview of the most frequently used abbreviations and acronyms used in the healthcare sector. The first five items are the five main emergency healthcare disciplines.

Dispatch Center or Alarm Room (Dutch: meldkamer, MKA).

General Practitioners Post for night hours (GPP, Dutch: huisartsenpost, HAP).

General Practitioners as first line care (GP, Dutch: huisarts).

Ambulance care that provide the first treatments on location at emergency incidents.

Emergency Department: a clinical department for a central organized emergency shelter that is available 24 hours a day and 7 days a week, accessible for all medical specialist emergency care for patients of all ages (ED, Dutch: spoedeisende hulp, SEH) [10].

Multi-agency healthcare: the combination of aforementioned disciplines in the emergency healthcare sector and the communication between them (Dutch: ketenzorg).

LSIV: National System Incidents and Vehicles (Dutch: Landelijke Server Incidenten en Voertuigen).

LSDV: National System Digital Pre-announcement (Dutch: Landelijke Systeem Digitale Vooraankondiging).

GMS: Integrated Dispatch Center System (Dutch: Gentegreerd Meldkamer Systeem). Refers to the systems used by the dispatch centers.

(Computer) system: software elements that are required for specification or design according to a particular set of requirements and the required hardware to run those software on [31].

Architecture of a system: the set of fundamental concepts or properties of the system in its environment, embodied in its elements, relationships, and the principles of its design and evolution [31].

Contents

List of Figures

List of Tables

1. Introduction

Healthcare professionals located in the emergency healthcare disciplines (dispatch center, general practitioners post, general practitioners, ambulances, emergency department) indicate that communication between them is in dire need of optimization [23][32][36]. They actively contribute every day towards improvement in the patients' safety and quality of healthcare received, and each discipline alone is able to provide adequate care within their focus area (e.g. ambulance). However, information about the patients and provided care is not carried over to other disciplines (e.g. emergency department) and each discipline starts again with retrieving information from the patient. There are little to none IT-solutions available that support the communication between the disciplines [29], and as of today, walls between these disciplines have hardly been torn down yet [1]. As a result, each discipline suffers from the same communication problems:

1. Critical patient information is often missing: often emergency physicians have to make critical decisions about patients based on insufficient information under time pressure. Patient's and medical records are missing, are incomplete or do not arrive in time.
2. Patient's medical records do not arrive on time to support emergency physicians' decisions, the information delivery speed is insufficient.
3. Overhead on the present communication structures, because synchronous methods such as calling are used between healthcare professionals as other systems are proven to be inadequate for communication.
4. Patient's medical records do not transcend the disciplines' boundaries, for instance, when a patient is transferred to the emergency department, the patient's medical records, treatment information and trauma assessments are not [15].

Workarounds for these communication problems are calling and oral description of a patient's traumas, records and assessments; an overhead at the expense of 36 million euro for the extra assessments and treatments that are performed because of missing or incomplete information [26]. And 1735 unnecessary deaths in 2017 because of missing medical information [30]. The quality of the emergency healthcare can be improved significantly when patient, medical, allergy and treatment records are available on time, correct and complete. At the moment, however, that still is not the case [1][36]. In daily practice this situation raises the following challenges when it comes to information sharing across multiple disciplines: insight in bed capacity, insight in medical history for ambulance staff on location, and transfer of medical information from the dispatch center, ambulance, and emergency department to the General Practitioner's (GP) records [26]. Information Technology (IT) holds the promise to deliver and coordinate that information, thus to relieve emergency physicians from the burden of making decisions based on unsuitable medical records. Unfortunately, the involvement from many parties supplying IT-systems resulted in a fragmented IT-landscape where IT-systems are built for discipline-specific use cases [29]. These systems tend not to communicate nor integrate with systems from other disciplines, where integration between disciplines could provide solutions for the current challenges in multi-agency healthcare. Continuing the old ways (by applying aforementioned workarounds on these complex challenges) only accumulates the work load on already pressurized emergency physicians. Therefore, instead of oral information transfer, information that has been entered in a system once, should be transferred to other IT-systems (from other disciplines) when needed. For this, IT-systems in the emergency healthcare sector require system integration.

Integration of IT-systems to share information is traditionally done on application level according to

the Enterprise Application Integration architectural principals (EAI) [33]. Each application is connected individually to any other application in or outside the discipline, with the advantage that each application can be integrated directly, point-to-point, in a cost-effective manner. However, with the huge number of IT-systems present in multi-agency emergency healthcare, the number of connections to have fully meshed point-to-point connections is given:

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

The number of connections and its complexity grows quadratically as each new system contributes n-1 connections.

The traditional integration approach of using point-to-point integrations is therefore not an option to consider, but instead a different architectural approach should be undertaken to achieve the desired information exchange. What this architectural approach should consist of is researched with the following research question:

RQ1: What architectural design can facilitate an optimal information delivery in the Dutch multi-agency emergency healthcare?

1.1. Emergency care market description and figures

The multi-agency emergency healthcare market consists of five disciplines: the dispatch center, General Practitioners Post (GPP), General Practitioners(GP), ambulance care, and the emergency department. Figures for these disciplines are as follows [2][23]:

1. Dispatch center: 21 dispatch centers (will be reduced to 10 in the coming years) employ 414 people.
2. GPP: 122 locations, often these are located in hospitals. These are open outside the regular working hours of GPs.
3. GP: 8879 GPs are working in the Netherlands, with multiple GPs working on one location.
4. Ambulance care: ambulances are organized in 25 Ambulance regions, in total containing 725 ambulances with 2218 staff.
5. Emergency department: 101 emergency departments exist (always located in a hospital). Also considered as emergency departments are gynecologists and heart emergency department, but the difference is that only known patients can arrive there.

In a year there are more than 7 million contact moments with patients in the emergency healthcare. From those contact moments are 4 million done by the general practitioners, and 2 million are treated at the ED. Ambulances made 1350.000 emergency trips to transport traumatized patients.

The dispatch center radios the ambulance through the P2000 (analog) or C2000 (digital) system, where it uses a Integrated Dispatch Center System (IDCS) to communicate with all ambulances from each region. Ambulance staff enters this information manually in the electronic trip form (increasingly using integration with the IDCS) and enrich it with trip and patient information. Patient information consists of, but is not limited to, situation assessment, trauma assessment, primary survey, and where necessary, a secondary survey. There is a great variety in automated support, the presence of many protocols and standards[1] and required registered information [23].

When a patient arrives at the emergency department the staffs' approach differs based whether the patient is stable or unstable. If the patient is unstable a team assembles ten minutes before the ambulance arrives. Team is provided specific instructions for each of them individually, and tasks are distributed so that everyone's exact task is known. Also the personnel is continuously updated about the treatment process, the patient's stability and provided with the necessary equipment. If the

[1] Nictiz enlists 164 healthcare standards https://www.nictiz.nl/overzicht-standaarden/

patient is stable only one staff member will indicate where to drop off the patient. In both cases SBAR(R)[2] method is used where the ambulance personnel fills the hospital staff orally in about the patient.

1.2. AmbuSuite

The research is an applied scientific study at AmbuSuite, a company enlisting 10 employees. AmbuSuite core business evolves around providing applications for the emergency healthcare sector, specifically:

1. AmbuForms: a electronic trip form used in ambulances by emergency staff on location. The application runs on iPads that each of the staff caries around. Information filled in on the iPads is send to AmbuSuite's servers.
2. AmbuView: a viewer at the Emergency Department in hospitals providing information about ambulances, their estimated times of arrival (ETAs), and patients' information. ETAs are either filled in by ambulance personnel or is calculated by rough estimations from Bing Maps.
3. AmbuFlow: back office system for trip billing. Trip information is pulled from AmbuForms, from which an invoice is created that can be send to insurance providers.
4. AmbuReports: Data Analytics Insight Application: enables data analysis on trip data to support business decisions and trip planning.

In 2018, AmbuSuite provided 41% of the Dutch ambulance market with AmbuForms, a figure that is being expended at the moment. From the 90 emergency departments, 49 departments in 35 hospitals use AmbuView. AmbuSuite offers a GPS-connection from the ambulance for the hospitals, but only 14 departments use it because of the greater price tag. The same applies to the eSpoed connection that enables patient's medical record transfer, because of the price tag the adoption rate is slow.

[2]SBAR(R): situation, background, assessment, recommendation, (repeat).

1.3. Research method

To answer the research question, the steps I take are based on the engineering cycle (figure 1) [38].

1.3.1. Problem investigation

I first performed a problem investigation by studying literature (section 2) to discover:

1. The stakeholders present in the modern Dutch multi-agency emergency healthcare that are directly responsible for the patient treatments and safety. To discover these stakeholders I study specialized emergency healthcare journals.
2. The requirements each of these stakeholders have for adequate communication. As presented in the introduction the communication between healthcare professionals in the multi-agency emergency healthcare sector is in dire need for optimization, and each of the, in previous steps discovered, stakeholders could have various requirements regarding multi-agency communication. To discover these requirements I study specialized emergency healthcare journals, following the coding approach as described below.
3. The architectural elements that could adhere to the stakeholders' requirements for digital communication. Based on the previously discovered requirements I conducted a literature review to discover architectural elements (a fundamental piece from which a system can be constructed [31]) with which an design can be made that enables multi-agency communication. For this I study software architecture to discover elements that adhere to one or more requirement. A selection of the discovered elements will be made if they fit one or more requirements, but non-fitting elements are left out.

To obtain the requirements the studied journals

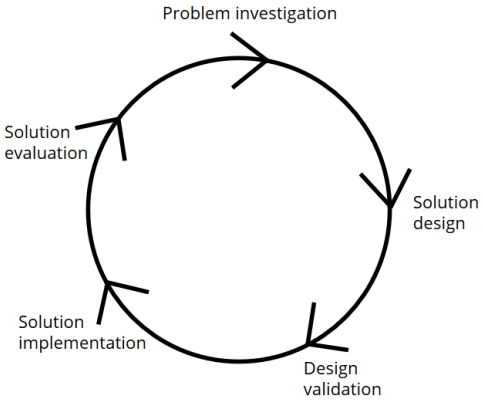

Fig. 1: Engineering cycle, adapted from [38].

were coded to the method that went as follows [21]:

1. Segmenting: determine meaningful units of information in the journal articles. Meaningful information is set as information that provides an (partial) answer meant to solve the aforementioned communication problems. Other parts are neglected and omitted.
2. Open coding: up until saturation, all meaningful pieces of information are coded with one code word that is unique for the type of information per unit.
3. Axial coding: to prevent duplicates a process of deduplicating is executed. Duplicates are merged into one code and are categorized. Finally, the categories are provided with a description of the content of the code.
4. Selective coding: in the last step, relations between categories are determined to provide a foundation for an interpretation of the requirements.

1.3.2. Treatment design

Second, I design a treatment based on the architectural elements I found in the literature study. All these fundamental pieces from which a system can be constructed are combined into an architectural description (section 3). An architectural description communicates the key aspects of the architecture to the appropriate stakeholders and as such forms a system's design [31].

1.3.3. Treatment validation

Third, I perform a validation on the architectural description using Rozanski's perspectives (section 4), that use a set of closely related quality requirements. All previously created viewpoints are revisited, validated and if the validation brings up quality requirements that are not incorporated in one of the viewpoints, these are changed accordingly [31].

1.3.4. Treatment implementation

Fourth, I implemented the treatment based on the architectural description, using a partner integration as a field-test for the implementation (section 5).

1.3.5. Treatment evalutation

Fifth, I evaluate the implementation on the methodology used, reflect on the literature and revisit the requirements whether they are satisfied with the implementation (section 6).

2. Literature review

This sections encompasses the three parts of the literature study: the stakeholder investigation, the requirements elicitation, and the architectural elements study. Each parts uses previously discovered knowledge to build upon in their respective literature study. Rather than using the traditional approach of collecting functional and non-functional requirements, the approach from Rozanski [31] is used. This does entail the collection of functional requirements for the system architecture, but predefined architectural perspectives are used to adapt existing viewpoints to incorporate quality properties (section 4).

2.1. Stakeholder investigation

A stakeholder in the architecture of a system is an individual, team, organization, or classes thereof,

having an interest in the realization of the system. As such, stakeholders drive the direction of the architecture and make the fundamental decisions about the scope, functionality, operational characteristics and structure. Therefore, the principle hold that architectures are solely created to meet stakeholders needs [31]. As stated in the introduction each discipline in the emergency healthcare sector faces similar communication problems, and the challenges faced regard inter-enterprise communication. Therefore, the identified stakeholders are the healthcare professionals in the emergency healthcare disciplines: the triagist at the dispatch center; the general practitioner at the General Practitioners Post (GPP), the general practitioner (GP), the ambulance nurse, and the emergency physicians at the emergency department (ED). Each of them are the actors within one of the aforementioned disciplines (depicted in figure 2 with their optimal communications). Their daily job is as follows:

Triagist at the dispatch center: The triagist is often the first contact for many of the emergency incidents. Their daily job is to perform traige on the incoming emergency incidents and determine its severity. Often experienced emergency physicians or ambulances nurses work there because of their field experience and ability to quickly assess a situation's severity while under time pressure.

General practitioners at GPP: These general practitioners are the first-line care for evening and weekend hours, often used for emergency assessments for which not directly an ambulance is required. The GP will often still call an ambulance when necessary because when patients go the GPP in the night, their situation is often more severe than usual and as such, often cannot wait until the next morning.

General practitioner: First-line care for all patients that are not in severe danger and require an ambulance, but for milder complaints. As such, GPs often contain extensive records of patients, information that is relevant in emergency situations.

Ambulance nurse: ambulance staff that is called in emergency situations and for patient transport. They are often the first ones to arrive at an incident and provide the first care. If necessary, they transport the patient(s) to the hospital.

Emergency physicians at ED: physicians that work at the emergency department and provide emergency treatments for severely injured patients. An emergency physician alone is a separate profession is as such on duty in an emergency department, but heart specialists and other profession-specific surgeons will be present for unstable patients as well.

Each of these stakeholders encounter emergency situations and communicate with each other in various ways. Where all stakeholders and their communications are depicted in figure 2, precise walktroughs of the emergency scenarios with the message contents are discussed in the next sections.

2.2. Scenarios

The scenarios below describe all the different real-life situations in which emergency incidents can occur. These situations are the description of the various day-to-day situations in which aforementioned stakeholders encounter emergency situations and are derived from [23]. The scenarios are divided into four parts based on who initiated the emergency incident. Each of the four scenarios encompasses various ways in which it can play out, depending on the severity of the emergency situation.

First scenario: initiated by the GP or the triagist from the GPP.

1. GP who refers a patient to the ED, GP calls Dispatch center for ambulance.

2. GP/triagist on GPP refers patient to ED.

Second scenario: initiated by family or other relatives from the patient.

1. From family or other relatives calling 112 (dispatch center), transport to ED.
2. Dispatch center provides medical advice.
3. Dispatch center advises going to a GPP.

Third scenario: initiated by bystanders.

1. Bystanders call 112 (dispatch center), transport to ED.
2. Bystanders call 112 (dispatch center), ambulance only provides medical care.
3. Bystanders call 112, ambulance provides medical care and advises GP visit.
4. Bystanders call 112 (dispatch center), dispatch center provides medical advice.

Fourth scenario: initiated by the patient.

1. Patient shows up at ED and goes to the hospital for treatment.
2. Patient shows up at ED, triagist refers him/her to GPP.

2.3. Message content

In each of the scenarios as described above the stakeholders communicate with each other. What they communicate and what messages are exchanged in each scenario are detailed here. However, the communication in the emergency healthcare sector is complex in nature, often contains feedback loops and involves many parties (figure 2). As AmbuSuite core business involves information delivery and processing for ambulances, the scope is narrowed down to the parties dispatch center, GP, ambulance care, and emergency department, with their respective communications lines. This removes the communication with the GPP and for instance the direct communication between the emergency department and the GP. Important to note is that communications between the dispatch center and ambulances are mostly verbal. The same applies for most communications with the GPs. Content of the messages send are in order (the numbers match the message numbers in the figure):

5. Dispatch center pre-announcement. This message announcement only happens in case of severely injured patients or big incidents, as it is not the usual way of communication because usually the ambulance decide to which ED to bring the patient to, whether or not based on patients preferences. Contents of the message are personal patient data, incident data, and alert data (including triage). Alert data is created when a dispatch center notes incident information; the caller; impact of the incident; its urgency; and the severity of the injuries according to the dispatch center's centralist.
6. Dispatch center command to drive for ambulances. This message happens always and is meant to support the verbal order provided by phone with patient and incident data. Contents of the message are personal patient data; incident data; alert data (including triage); destination data; and urgency rating code.
7. Professional patient record summary for the ambulance. This message's purpose is to inform ambulance staff optimally about patient to provide adequate care. Contents of the message are unprocessed messages resulting from assessment and previous emergency situations; episodic list; moment of contacts in the last ten days; medication data from the last four months; measurements performed on the patient in last ten days; summaries of correspondence in the last ten days; family medical history; treatments; and additional social data.
8. Ambulance arrival announcement to ED. This message is sent after initial anamnesis (medical history check) performed by ambulance staff on the incident location and notifies the hospital that a patient is incoming. If the ETA is known, it is send as well.
9. Intervention message, can be repeated as often as required, from small messages as ETA up-

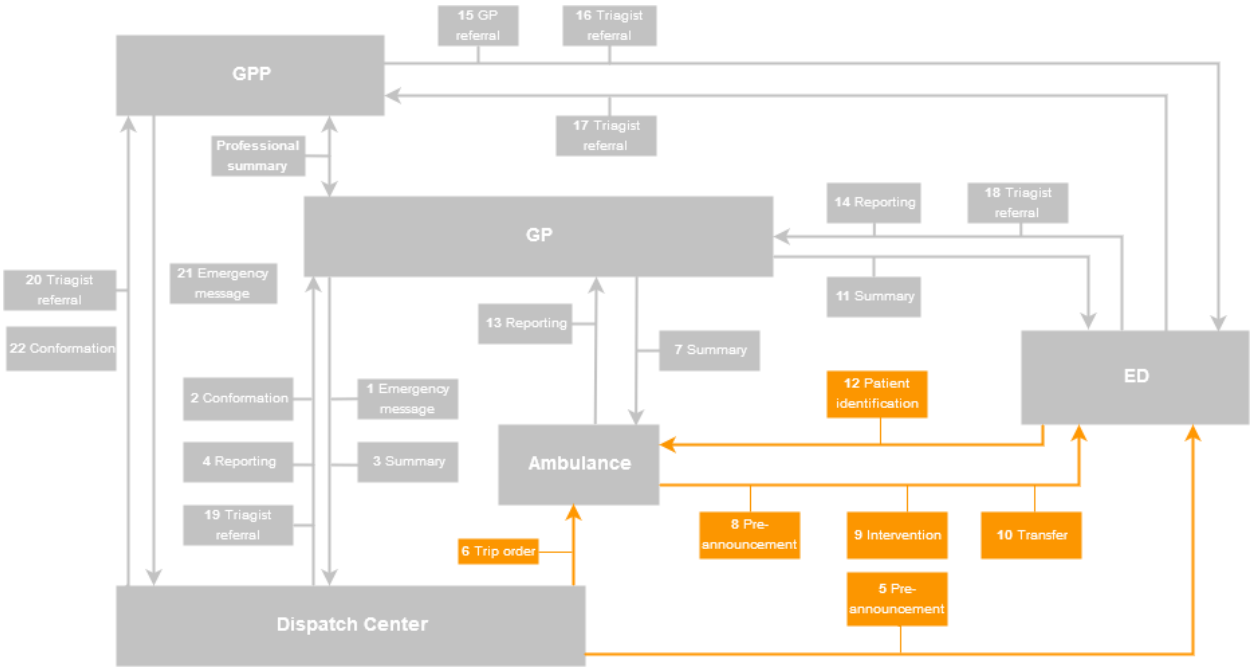

Fig. 2: Ideal communication in the multi-agency emergency healthcare [23].

dates to patient assessments or medical records update (see next item).

10. Transfer message. The patient is transferred to the ED, and communication from the ambulance is closed. Contents of this message are data regarding the patient; incident; alert (including triage); trip; departure; destination; SBAR(R); medication; infuse; primary and secondary assessment survey; work diagnosis; patient placement; and appendices.
12. Patient identification from ED to the ambulance. This message is send when ambulances did not receive patient information from the dispatch center, GP or the patient itself, and patient identification can be provided this way. Contents of the message are trip identification; patient identification; patient diagnoses; and remarks from the patients transfer or treatment (e.g. from ED to ambulance).
13. Ambulance report to GP. This message is send in case the ambulance staff performs medical treatment on the patient after anamnesis. A report of the treatment is send to the GP if this is the last medical treatment (no patient transport to a ED). Purpose is to inform the GP about the emergency situation and the provided medical treatment. Contents of the message are the reason of the message; diagnosis; and medical advice provided to the patient. In an appendix the anamnesis; physical examination; and intervention are detailed.

2.4. Requirements analysis

This section dives into the requirements that previously discovered stakeholders have for communication in the daily scenarios as detailed above, when they want to communicate the information as specified in the message content section. To recap, the stakeholders where as follows: the triagist

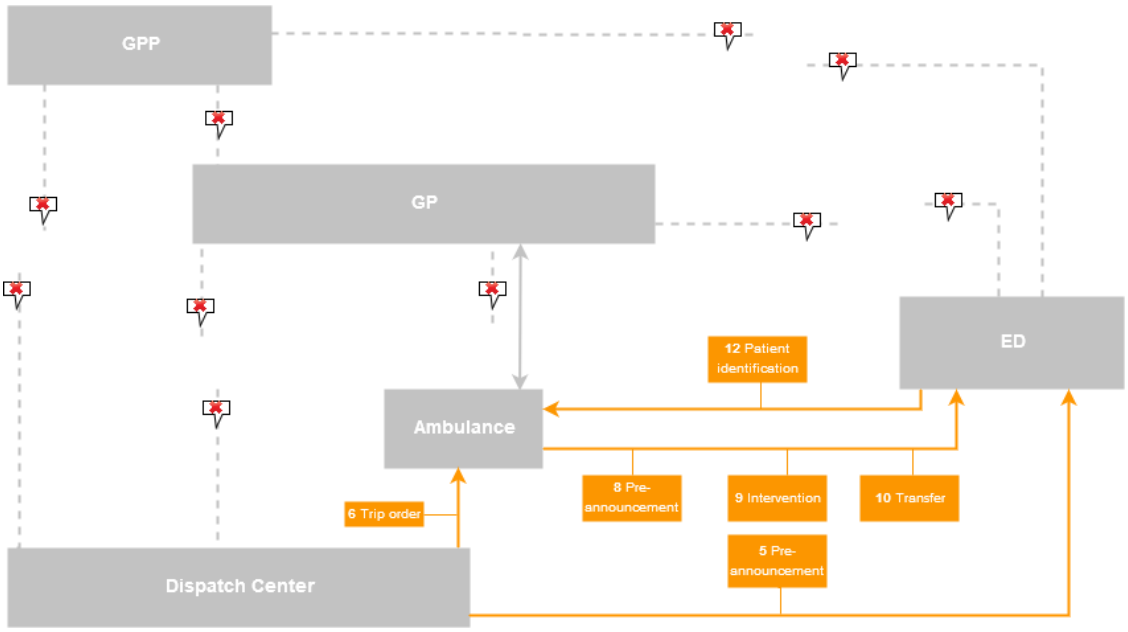

Fig. 3: Reality about communication in the multi-agency emergency healthcare [13].

at the dispatch center; the general practitioner at the General Practitioners Post (GPP), the general practitioner (GP), the ambulance nurse, and the emergency physicians at the emergency department (ED). The communication problems these stakeholders are facing (as stated in the introduction) are the following:

1. Critical patient information is often missing
2. Patient's medical records do not arrive on time
3. Overhead is present in current communication
4. Medical records do not transcend the disciplines' boundaries

These communication problems result in the following challenges: insight in bed capacity; insight in medical history for ambulance staff on location; and transfer of medical information from the dispatch center, ambulance, and ED to the GP's records [26]. These challenges require system integration and cannot be solved anymore with oral or written information transfer [1][29].

As all the stakeholders face similar communication issues, in the technical specification the stakeholders can be grouped together since all stakeholder share a common goal: sending and receiving messages. In this case, the content of the message is subordinate to the message itself, i.e. for a communication medium the content of the message is less relevant than the communication infrastructure. Therefore, the stakeholders involved in the technical specification are grouped as message senders and message receivers. Other stakeholders involved in the technical specification are the developers that implement the architectural design; system maintainers for system evolution; system administrators that operate the system during runtime; and communicators that create documentation and market features.

As can be seen in figure 3, many communication lines that had been present in figure 2 are in reality not there. Instead of digital communication,

communications between many of the stakeholders are mostly verbally, by email, and even writing. As this type of communication becomes outdated and insufficient to solve the aforementioned challenges, stakeholders want to be able to send and receive messages between them to face the challenges head-on. The stakeholders' requirements were collected from an exhaustive study of specialized medical journals. To obtain the requirements the studied journals were coded according to the open coding method (section 1.3.1). The coding process resulted in categorized set of requirements: unified messaging; reliable communication; provisioning and monitoring; dynamic scaling; flexible services; secure communication; and integration with other tools [2][15][23][26][28][29][36]. The list of literature used can be found in the appendix, along with the used codes (section 8.2). Each of the discovered requirements is further detailed below.

2.4.1. Unified messaging

Unified messaging holds that any type of healthcare message can be send through one communication infrastructure, that one system can connect with other tools and handle the messaging between them. In such a system, direct communication with each system is not desired, but instead the message provider should be decoupled from the message sender. A communication infrastructure that connects systems and acts a message broker between parties can enable such a decoupling from sender to receiver, one system can drop of a message and forgets about it.

In addition, the communication infrastructure should be able to broadcast messages. For instance, in scenario eight it is desired to perform an hospital availability check, a broadcast message that is send to multiple hospitals can check for the beds available and return the results. Each of the stakeholders however, is only interested in a few types of messages. For instance, an emergency department wants to receive data from an incoming patient and the ambulances ETA, but is not interested in ETAs from all ambulances or all patients that are being transported to other hospitals. These parties would prefer to only receive a selection of the messages and filter irrelevant messages. This will also improve the security as only data meant for the selected party is send to that party, and others cannot see any of the sensitive information.

Also, messages in the communication structure should be able to contain all kinds of information in messages regardless of the content nor the reason a message is send (section 2.3). I.e. an ambulance arrival announcement to ED should use the same message structure as a professional patient record summary for the ambulance or an ambulance's ETA update. Finally, for both the sender and the receiver it is desired that the message format is compatible with each other. Less business logic is then necessary to translate and transform incoming messages.

2.4.2. Reliable communication

A patient's medical information is time-critical so information should be published to subscribed parties as soon as available, which opts for an event-driven approach. The message delivery must be resilient to ensure that messages are delivered for a second time even when it fails once. It should also be idempotent (message should be processed without changing it characteristics). Finally, the emergency healthcare runs 24/7 and as such should the communication infrastructure have a high availability and up-time to ensure that messages can be delivered at any time.

2.4.3. Provisioning and monitoring

Provisioning is defined as the configuration of connected systems, providing users with access to data and resources. It refers to all enterprise-wide information-resource management involved and should therefore have the functionality to grant access to authorized resources. System administrators should be able to monitor the system and intervene where necessary, e.g. when messages are

not received. In addition, when the communication infrastructure is not performing as desired, logs should be available for analysis.

2.4.4. Dynamic scaling

Requests for the communication infrastructure should be load balanced and scalable to handle peak hours. Dynamic scaling also regards beside throughput and response time the number of modules that can be added, the communication structure should be able to scale up and out.

2.4.5. Flexible services

The communication infrastructure should be flexible in the variety of systems that it can communicate with, and not exclude systems because it does not support the used message format or structure. As healthcare professionals use a wide variety of systems many systems should be able to use the communication infrastructure [1][29].

2.4.6. Secure communication

Secure communication is a must have because of the sensitive nature of the sector that deals with a lot of personal medical data. As sensitive patient information is sent over digital communication channels, no data leaks or unauthenticated access should be possible. In addition, only designated parties should be able to obtain access to the data send over the communication infrastructure, and only the data that is meant for them to see.

2.4.7. Integration with other tools

The communication infrastructure should be able to expose data trough an API that handles the authorization and authentication of users. In that way, existing applications can be integrated by writing adapters that transform messages and connect systems with the communication infrastructure.

2.5. Architectural elements

This section builds upon previously discovered stakeholders and their requirements regarding communication among them in the daily scenarios they encounter. The discovered requirements are used as a guiding principle when studying the literature reviewing architectural elements. Elements that do not fit into one of the requirements or cannot act as a fundamental system element are left out, the ones that do fit are used for the system architecture design in section 3. An architectural element is defined as a fundamental piece from which a system can be constructed and can therefore vary from design patterns to complete architectures [31]. It possesses the following key attributes: a defined set of responsibilities, defined boundaries, and a set of clearly defined interfaces. The latter defines the services that the elements provides to the other architectural elements

2.6. Architectural viewpoints

It is not desirable and not even possible to include all details for an architecture in one single all-encompassing architecture model. Instead, complex systems should be represented in a way that is manageable and comprehensible by a range of business and technical stakeholders. Rozanski provides an approach to avoid a monolithic design but capture the architectural design from several directions simultaneously. With this approach the architectural design is partitioned into a number of separate but interrelated viewpoints, each of which describes a separate aspect of the architecture [31]. A viewpoint is only a partial representation of an architecture, and therefore, the entire sets of viewpoints is necessary to understand the complete architecture. Considerations when creating a viewpoint regard the scope (what structural aspects are represented), element types (what types of elements are categorized), audience (which stakeholders is the viewpoint targeted at), audience expertise (level of technical understanding), and level of detail.

For several scenarios viewpoints are already predefined. A viewpoint is formally defined as a collection of patterns, templates and conventions for constructing one type of viewpoint. It defines the stakeholders whose concerns are reflected in the view-

point and the guidelines, principles and template models for constructing its viewpoints [4][31]. Architectural viewpoints provide a framework for capturing reusable architectural knowledge, in such a way that separation of concerns is guaranteed because the focus is separately on independent aspects of a system. In addition, as different stakeholders have different concerns, addressing them can be done with the help of different viewpoints. This results in a reduced complexity for a designed architectural viewpoint. To avoid fragmentation, only viewpoints that address significant concerns for the system are included. Four core viewpoints are listed below [31]. The functional, information and concurrency viewpoints characterize the fundamental organization of the system, where the context viewpoint provide insight in the environment the system acts in.

Context viewpoint: describes the relationships, dependencies, and interactions between the system and its environment (the people, systems, and external entities with which it interacts). It concerns the system scope and its responsibilities, identifies (nature of) external entities, external interfaces, completeness, consistency and coherence.

Functional viewpoint: describes the systems run-time functional elements, their responsibilities, interfaces and primary interactions, and is as such the foundation for other viewpoints.

Information viewpoint: describes the way the systems stores, manipulates, manages, and distributes information. This viewpoint develops a complete but high-level view of static data structure and information flow.

Concurrency viewpoint: describes the concurrency structure of the system and maps functional elements to concurrency units to identify the parts of the system that can execute concurrently and how this is coordinated.

2.7. Architectural perspectives

There is an inherent need to consider quality properties (e.g. security) in each architectural viewpoint as considering what a system does is only part of the story, and that how the system provides it services is crucial as well. An architectural perspective revisits the created viewpoints with a quality property (e.g. security) in mind, to assess and review the architectural models to ensure that the architecture exhibits the required properties. As such, an architectural perspective is defined as a collection of architectural activities, tactics, and guidelines that are used to ensure that a system exhibits a particular set of related quality properties that require consideration across a number of the system's architectural viewpoints [31].

The most important perspectives for large information systems are: security; performance and scalability; availability and resilience; and evolution and maintainability [17][31].

2.7.1. Security

The security perspective copes with controlling access for system resources, regarding almost the complete system. It handles a system's user identification and authentication; control different classes of access at varying levels of granularity; and maintain secret information. When designing a system architecture sensitive resources are identified together with their potentials threats and its mitigations. This can result in partitioning and restricting parts; new security hardware; new operational procedures, certification, backups procedures.

2.7.2. Performance and scalability

The performance and scalability perspective copes with the throughput, response time, predictability etc. from a system; as well as the ease of the system distribution and duplication. Performance ensures the system continues running when the number of users, transactions or complexity increases. The way that hardware is configured, how resources are allocated and how software is written all influences

performance.

System scalability is defined as the system's ability to cope with increased load, with load not being one-dimensional [17]. Load can be expressed in parameters such as requests per second, read/write ratio, simultaneously active providers, etc. Performance in case of online systems is then the response time (time between client sending a request and receiving a response). Response time can be measured in percentile in two ways: what is the performance impact when a load parameter increases, or how much resources need to be increased when increasing load parameter to keep the same performance. As response time is different each time of request, it is a distribution rather than a single number as outliers occur in practice. Performance can be maintained by scaling machines up (using more powerful machines) or scaling out (distributing the load across multiple machines). Designing a scalable architecture to handle load evolves around the assumption of future load.

2.7.3. Availability and resilience

The availability and resilience perspective copes with ensuring a system's availability and deals with failures that could affect this. How the prevention of failures is organized is expressed in a system reliability indicator, what for the most part is proper error handling (many critical bugs are actually because of poor error handling [40]) and ensuring that systems do not fail completely. On a larger scale, reliability evolves around systematic errors within the system, such as runaway processes, corrupted responses, and cascading failures [11]. The pesky characteristic from these systematic errors is that they often lie dormant for a long time until they are triggered by a unusual set of circumstances and input. Careful thinking about assumptions and interactions in the system can avoid a lot of the damage, as well as thorough testing; process isolation; allowing processes to crash and restart; measuring, monitoring, and analyzing system behavior in production [17].

Reliability among distributed machines is more difficult to realize as with each single machine added that has P chance of failure, that when failing independently, the probability of data loss is P^N. So for any desired reliability R and any single-node failure probability P some replication N can be picked that $P^N <$ R [19]. The fundamental problem behind distributed system failure is that faults causing failures are replicated over multiple nodes or machines as they are distributed, therefore system failures are not independent but rather cascading. In other words, if one bug is on one machine, it is on others as the software runs on all the machines in a distributed system. Part of the difficulty is that code-complexity increases when coping with multiple machines, e.g. when dealing with hardware degradation that result in slower disk access times, which probability increases when more machines are added. Distributed systems typically require more configuration and more complex configuration because they need to be cluster aware and have to deal with timeouts. This configuration is, of course, shared; and this creates yet another opportunity to bring everything to its knees [19]. Another major difficulty of distributed systems are monitoring the systems and configuration management, what is difficult to execute in smaller organizations.

2.7.4. Evolution and maintainability

The evolution and maintainability perspective copes with future expansion of the system and the ease of maintenance after the initial system's deployment. Preferably a system is designed with evolution in mind, that allows for easy extension of the system when required, without requiring major code rewrites or refactoring. Maintainability evolves around the ability to maintain the system after initial deliverance, such as fixing bugs, keeping the system operational, investigating failures, modifying it for new use cases, etc. It is often overlooked but is nevertheless important as the major-

ity of the costs are in the ongoing maintenance [17]. Software can be designed in such a way that maintainability can be improved by emphasizing three principles: operability (ease of system operation), simplicity (reducing system complexity), and evolution (systems' ability to cope with changes in the feature).

2.8. Event Driven Architecture

An event-driven architecture consists of event senders that generate a stream of events, and event receivers that listen for the events. An event is an observable change in the state of an IT system; it can be triggered both by real world events, such as presence detection, timeouts etc. or by internal events like the reception of a message(e.g. a command) or the completion of a task [9]. Events are delivered in real-time, thus receivers can respond immediately to events as they occur. Senders are decoupled from receivers and a sender has no knowledge about which receivers are listening. Receivers are also decoupled from each other, and every receiver can unless otherwise arranged see all of the events. In the first interaction, the occurrence of an event (a notable thing that happens inside or outside your business) can trigger the invocation of one or many services. Those services may perform simple functions, or consist of entire business processes [37].

2.9. Micro services

Many business applications store redundant data along with implementing redundant functionality, where other systems could benefit as those functions are exposed and available as a service to other systems [14]. Micro services are a variant of the service-oriented architecture (SOA) architectural style that structures an application as a collection of loosely coupled services. In a micro services architecture, services are fine-grained and the protocols are lightweight. The benefit of decomposing an application into different smaller services is that it improves modularity. This makes the application easier to understand, develop, test, and become more resilient to architecture erosion. Services can be deployed independently, not considering each technology stack, libraries or frameworks.

A major element of the micro service approach is the service discovery, where service look up at the API gateway is used to find the endpoint for a service. The gateway forwards the call to the appropriate services on the back-end. The API gateway can aggregate the responses from several services and return the aggregated response, as well as handling authentication, logging, SSL termination, and load balancing.

A team can update an existing service without rebuilding and redeploying the entire application. Services are responsible for persisting their own data or external state. It parallelizes development by enabling small autonomous teams to develop, deploy and scale their respective services independently. It also allows the architecture of an individual service to emerge through continuous refactoring, enabling continuous delivery and deployment.

2.10. Enterprise Integration Patterns

As application rarely live in isolation, integration solutions are heavily used nowadays. But integrated systems face a few fundamental communication challenges that networks are unreliable as they come with communication links that can cause delays or interruptions and are slow in communication compared to a local method call [14]. Also, any two applications are different and require interfaces to interact, that are subject to change time as well. And changes in one application might effect more applications than just the integrated solution when applications are tightly coupled. For integrated systems to communicate, multiple styles of communication can be applied and integrated in one solution simultaneously. However, messaging is more immediate than file transfer, better encapsulated than a shared database, and more reliable than remote procedure invocation. Using messaging allows for

allow asynchronous communication, where integration can be achieved between a n-number of applications with the use of a common message channel [14].

For asynchronous messaging two principles are important: send and forget, and store and forward. The send and forget principle allows the sender to create a message, send it to the message channel and continue doing other work. In other words, no application lock is created while waiting for some response, the sender can be confident that the receiver will eventually receive the message. The second one, store and forward, the message is first stored on the senders' system, before being transmitted. This allows for separation of concerns by delegating the responsibility of delivering data to the messaging system. Where asynchronous messaging offers numbers of advantages such as remote communication, multiple platform/language integration, variable timings, reliable communication and thread management, it is also facing challenges from which some are inherent to the asynchronous model. They are among others: a complex programming model as it uses event handlers that respond to incoming messages; sequences issues as message channels guarantee delivery, but not when it will be delivered; facing synchronous scenarios where the gap between synchronous and asynchronous communication is bridged; and performance, as the store and forward principle adds some overhead.

Asynchronous messaging has a number of programming implications as the send and forget approach allows the application to continue working after sending the message, where it otherwise would be blocked waiting for the callback. First, no single tread of execution exists anymore, but multiple threads enable sub-procedures to run concurrently. It can greatly improve performance by ensuring that sub-processes are making progress while others are waiting, but makes debugging more difficult. Second, the asynchronous nature of incoming messages mandates that message handling happens upon arrival, which can interrupt the tasks the receiving system was working on. Third, using asynchronous messaging mandates that sub-processes run independently from each other, but that requires a caller that can combine the results from different processes together, even when they are out of order. Fourth, developers have often little control over the participating applications, which is a fairly limiting constraint, and standardization of communication protocols and message formats is hard. Distributed programs makes governing applications harder, as they are often spanning the entire enterprise.

2.11. Message Oriented Middleware

Message-oriented middleware (MOM) is software or hardware infrastructure that supports sending and receiving asynchronous messages between distributed systems. MOM allows application modules to be distributed over heterogeneous platforms and reduces the complexity of developing applications that span multiple operating systems and network protocols. The middleware creates a distributed communications layer that insulates the application developer from the details of the various operating systems and network interfaces. APIs that extend across diverse platforms and networks are typically provided by MOM. This middleware layer allows software components that have been developed independently and that run on different networked platforms to interact with one another. Applications distributed on different network nodes use the application interface to communicate. In addition, by providing an administrative interface, this new, virtual system of interconnected applications can be made reliable and secure. MOM provides software elements that reside in all communicating components of a client/server architecture and typically support asynchronous calls between the client and server applications.

2.11.1. Message broker

Messaging-oriented middleware relies on using a message broker, which is an intermediary module that takes incoming messages from applications and perform some action on them. It mediates communication among applications, minimizing the mutual awareness that applications should have of each other in order to be able to exchange messages, effectively implementing decoupling. Features of a message broker might consist of:

1. Translating messages from the formal messaging protocol of the sender to the formal messaging protocol of the receiver.
2. Transform messages to an alternative representation.
3. With message aggregation multiple incoming messages are combined in a larger message, sometimes involving pre-processing such as message translation and transformation.
4. Decomposing messages into multiple messages and sending them to their destination, then recomposing the responses into one message to return to the user.
5. The headers of the incoming messages are inspected for the destination and are then routed to the correct destination.
6. Provide content and topic-based message routing using the publish–subscribe pattern.
7. Other tasks can consist of message validation, buffering, providing reliable storage, guaranteed message delivery, invoking web services to retrieve data, and responding to events or errors.

2.11.2. Message queues

Message brokers use message queues for asynchronous communication, meaning that the sender and receiver of the message do not need to interact with the message queue at the same time. Messages placed onto the queue are saved until the recipient retrieves them. Message queues have implicit or explicit limits on the size of data that may be transmitted in a single message and the number of messages that may remain outstanding on the queue. In a message queue messages can be grouped into categories according to the publish-subscribe pattern. Message senders publish messages into these categories, and when receivers express interest in one of more of these categories, senders can subscribe to receive these messages. In this way, messages are not send directly to receivers, but instead to a pool of consumers where receivers can subscribe to. This enables even further decoupling of message senders and receivers as messages are published in a category and not send directly to a receiver anymore. A publish-subscribe pattern has the disadvantage that, if handled improperly, the pile of undelivered messages can pile up, what in turn can cause load surges. In addition, a message is removed from the queue once it is received. Synchronous conformation of a task execution cannot be directly provided anymore, and the non-persistent nature of the message queue disables direct control over system tasks.

To partially counteract the non-persistent nature of the message queue and improve task handling the publish-subscribe pattern can be extended with the function that sent messages are not disposed/lost upon reception, but upon confirmation. Messages send in the message queue are processed, send, acknowledged or confirmed, and only then deleted. This ensures that information-critical messages are always received and read by the receiver, or after several retries found in the dead letter.

2.11.3. Dead letter channel pattern

The dead letter channel is the place for undeliverable messages, where messages send in the message queue end up when the receiver cannot receive the message after several retries. The message queue cannot return the message as the sender is no receiver and cannot detect deliveries. The dead letter channel offers a place to store invalid or undeliverable messages, where system administrators can inspect a message and check the errors, while in-

forming the system maintainers in the process.

2.11.4. Event message pattern

An event is a message that notifies other components about a change or an action that has taken place. An event's message content are typically less important, as such they can have an empty message body, their occurrence tells the observer to react. In a push model it could occur that an event is a combined document/event message, but its only useful when receivers want the content, otherwise large messages will be ignored (if only event is useful) and will cause overhead.

2.11.5. Message store

A duplication from the message that is send over the messaging channel is send to another channel to be stored in a database. The method of fire-and-forget will not slow down messaging, only increase the network traffic because a second message is send. Not all types of messages are necessary to store and from the relevant document messages not all information may not be necessary to be stored too. The advantage of this pattern is that it enables message logging, useful when the message content is important to preserve, e.g. when the same information needs to be shared with other receivers later on.

2.12. Messaging as a Service

Distributed systems are less robust than single systems as reliability in distributed systems is harder to realize (section 2.7.3) because failures in software are replicated over machines. In turn, adding more machines increases the probability of failures [17]. A solution for this is Messaging as a Service (MaaS), effectively outsourcing the messaging infrastructure to prevent any of the concerns that trouble in-house Software as a Service (SaaS) solutions such as availability, resilience, performance and scalability. When outsourcing the messaging infrastructure, the typical messaging problems with networks, clocks and timing issues that occur as well as independent system failure are up to the specialized cloud provider's system administrators. Messaging as a service resolves aforementioned issues for the end-user as well as counteracts the non-deterministic nature of systems (the same operation does not necessarily return the same result, based on e.g. system response time and throughput). Further advantages can include elastic/on-demand resource allocation, a guaranteed minimum level of reliability, existing management tools, flexible services, and adapters for integration with other tools [20].

2.13. Cloud-based integration

Integration Platform as a Service (iPaaS) consists of four pillars: API management, orchestration of business processes and workflows, a service bus for messaging, and the event grid for notifications [20]. Moving API management to the cloud enables the connection between cloud and on-premise applications with other applications. As most applications expose their functionality through APIs, with cloud-based API management the number of invocations, authentication, speed, usage pattern analysis and documentation can be created and controlled. Results from the called APIs can be sent to other applications using the service bus (a message queue) that enables asynchronous messaging as a service. When a message is received, an event grid can notify the sender that a new messages has arrived. In this way it is avoided that receivers have to poll the service bus on a fixed time interval. The receiver registers an event handler for the event source it is interested in. Event Grid then invokes that event handler when the specified event occurs. Finally, orchestration of business processes and workflows combines data from different sources to create one workflow, whereby acting as a singular instance of a software solution that is presented to end-users.

2.14. OAuth2.0

The authorization code grant type is used to obtain both access tokens and refresh tokens and is optimized for confidential clients. Since this is a redirection-based flow, the client must be capable

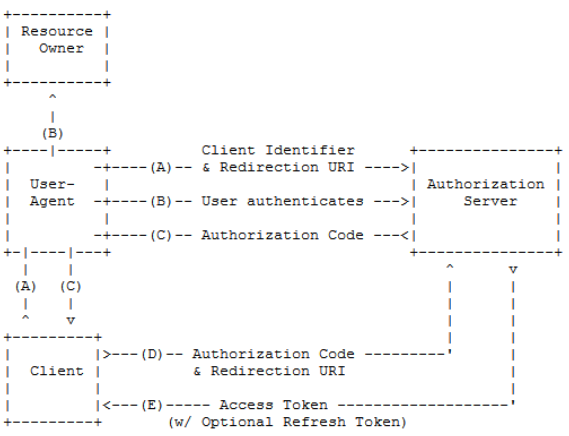

Fig. 4: The OAuth authorization flow [12].

of interacting with the resource owner's user-agent (typically a web browser) and capable of receiving incoming requests via redirection from the authorization server.

The flow illustrated in Figure 3 includes the following steps [12]:

(A) The client initiates the flow by directing the resource owner's user-agent to the authorization endpoint. The client includes its client identifier, requested scope, local state, and a redirection URI to which the authorization server will send the user-agent back once access is granted (or denied).

(B) The authorization server authenticates the resource owner (via the user-agent) and establishes whether the resource owner grants or denies the client's access request.

(C) Assuming the resource owner grants access, the authorization server redirects the user-agent back to the client using the redirection URI provided earlier (in the request or during client registration). The redirection URI includes an authorization code and any local state provided by the client earlier.

(D) The client requests an access token from the authorization server's token endpoint by including the authorization code received in the previous step. When making the request, the client authenticates with the authorization server. The client includes the redirection URI used to obtain the authorization code for verification.

(E) The authorization server authenticates the client, validates the authorization code, and ensures that the redirection URI received matches the URI used to redirect the client in step (C). If valid, the authorization server responds back with an access token and, optionally, a refresh token.

3. Architectural description

This section encompasses the treatment design, build upon previously discovered stakeholders, their requirements and fitting architectural elements. The treatment design is a set of procedures that documents an architecture in a way its stakeholders can understand and demonstrates that the architecture has met their concerns, thereby effectively and consistently communicating the key aspects of the architecture to the appropriate stakeholders [31]. The deliverables do not only encompass the architectural models, but also the scope definition, constraints and design principles. Finally, this results in a documentation of the architecture that is called an Architectural Description (AD). To trace the links between the requirements and the, from the literature derived, architectural elements Table 1 provides an overview of each architectural element linked with the requirement it fulfills.

3.1. Relation between viewpoints

All elements in an Architectural Description (AD) are linked with each other (figure 11). An architecture comprises architectural elements, has a systems can be documented by an AD. The AD itself documents the architecture for the stakeholder that has specific concerns. In addition, an AD comprises view(points) that addresses these concerns. Finally,

Requirement	Requirement Number	Architectural Element	Element Section
Unified Messaging	2.4.1	Enterprise Integration patterns	2.10
		Messaging oriented middleware	2.11
		Message broker	2.11.1
		Message queues	2.11.2
Reliable Communication	2.4.2	Dead letter channel	2.11.3
		Message store	2.11.5
		Messaging as a service	2.12
Provisioning and monitoring	2.4.3	Messaging as a service	2.12
Dynamic Scaling	2.4.4	Micro services	2.9
Secure Communication	2.4.6	OAuth 2.0	2.14
Integration with other tools	2.4.7	Event-driven architecture	2.8
		Event message	2.11.4
		Cloud-based integration	2.13

Table 1: Requirement traceability table

perspectives shape the view(points) that form the foundation of the AD.

3.2. Context viewpoint

The context viewpoint describes the relationships, dependencies, and interactions between the system and its environment (the people, systems, and external entities with which it interacts). It concerns the system scope and its responsibilities, identifies (the nature of) external entities, external interfaces, completeness, consistency and coherence [31].

3.2.1. Scope

The scope of the architectural description is framed by the research question what architectural design can facilitate an optimal information delivery in the Dutch multi-agency emergency healthcare. One of the stakeholders' requirements to achieve optimal information delivery is unified messaging (section 2.4.1), therefore the architectural design should encompass an communication infrastructure that satisfies that demand. Other requirements regarded the demand for dynamic scaling; flexible services; secure communication; and integration with other tools; any communication infrastructure designed should meet those demands. Communication should be asynchronous and support broadcasting to interested parties; support scaling up and out; usable on any device; and support integration with other existing tools. The integration should be handled by external adapters that transform messages and connect systems with the communication infrastructure.

3.2.2. Design rationale

The context diagram provides a high-level overview of the system and its environment, showing the stakeholders and how they interact with the system. The communication infrastructure is depicted in the middle (figure 5) with the connecting entities, whether internal or external, surrounding it. As the gateway (API) is part of the communication system, it is not depicted here, but in the functional viewpoint. The existing structure with information extraction from the LSIV and the dispatch center is kept intact as it suffices. Where this usually feeds into AmbuForms directly, it uses the communication structure first as a medium to post messages and to store them in databases. The incident and

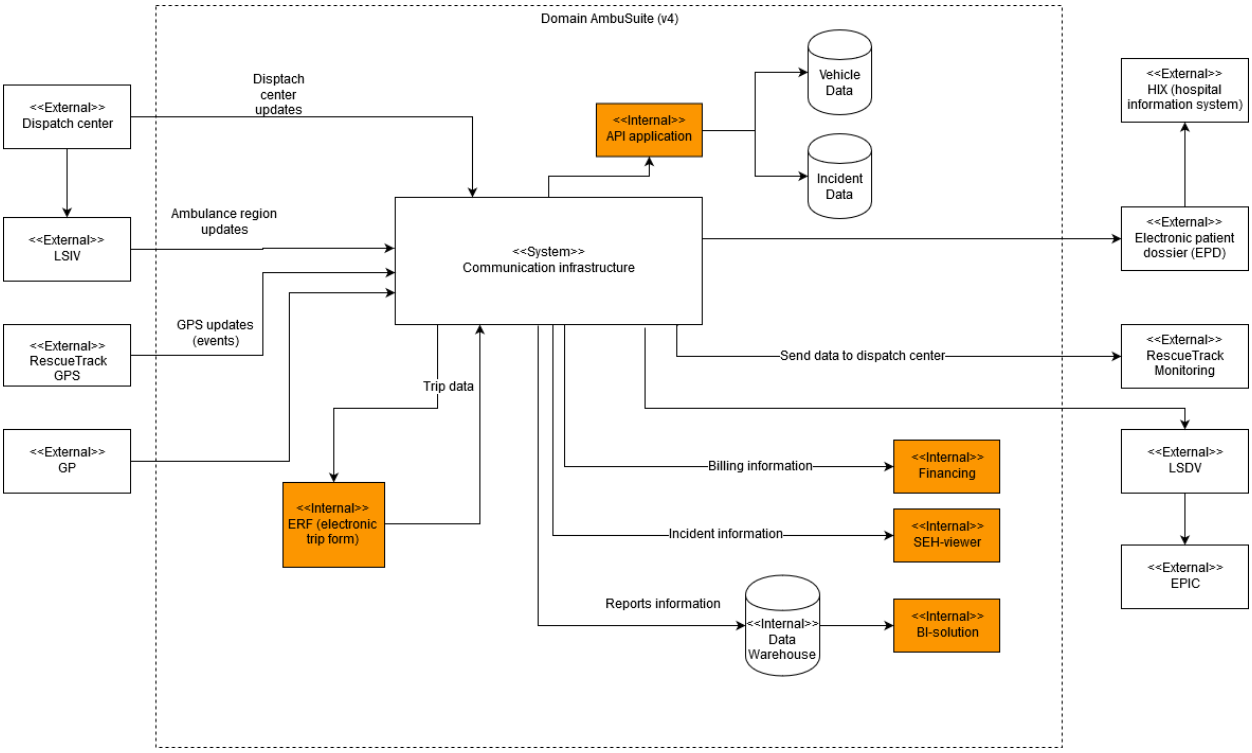

Fig. 5: Context diagram of the communication infrastructure's environment.

vehicle information that are retrieved from the LSIV and the the dispatch center are stored in individual databases separating incidents and vehicles from each other, maintaining the current database structure. AmbuSuite's current applications (depicted in orange) are replaced by generic names to indicate that these are replaceable applications. Outgoing data is send to many interested parties, such as the electronic patient dossier (EPD) and the LSDV, both obligated by the Dutch government.

3.3. Functional viewpoint

The functional viewpoint describes the systems run-time functional elements, their responsibilities, interfaces and primary interactions, and is as such the foundation for other viewpoints. The created functional architecture model (FAM) has a significant impact on the system's quality properties such as ability to change, its ability to be secure, and its run-time performance.

3.3.1. Design rationale

As can be seen in figure 6, an authorization module contains an identity provider that handles token issuing, and can verify tokens for each message or event that users want to send through the message broker. After buffering, both event messages and document messages are send through the messaging queue, that determines based on topic subscriptions which receiver wants to receive a specific message. An event invoker is watching the message queue until it fires (indicating a new message has arrived) and then wakes the receiver, that pulls the messaging queue to retrieve the new message. This avoid polling for new messages and thus unnecessary requests. As the micro services approach dictates, different functionalities are separated from each other in separate services. The authentication module is separated from the messaging service, since the authentication module only needs to be contacted in case the issued token has been expired.

3.3.2. Design decisions

Each design decision that has been made to create the functional architecture model is specified below. The literature from where these design decisions are based on can be found in Table 1 for traceability purposes.

1. An event driven architecture forms the foundation for the sector architecture, intended to support massive concurrency demands for a wide range of applications. In event-driven architecture, applications are constructed as a set of event-driven stages separated by queues. This design allows services to be well-conditioned to load, preventing resources from being over-committed when demand exceeds service capacity. Decomposing services into a set of stages enables modularity and code reuse [37].
2. The micro services architecture style form the basis for the modules such as the authentication module as this enables fine-grained services and lightweight protocols. All modules are accessible through one public API only.
3. A messaging queuing service with a publish/-subscribe pattern is used. On this pattern an extension is made to ensure delivering of messages by only deleting them upon conformation instead of reception. Topics are used so that receivers are only receiving the messages for which they are subscribed on. In this way the asynchronous remote communication can separate applications' responsibilities; distinct platforms can be integrated with each other; and an increased reliability as application can exist independently [14].
4. A dead letter channel is used to catch messages that are cannot be delivered. To avoid message congestion in the queue a message is after a couple of retries send to the dead letter channel, where the errors can be checked by system administrators.

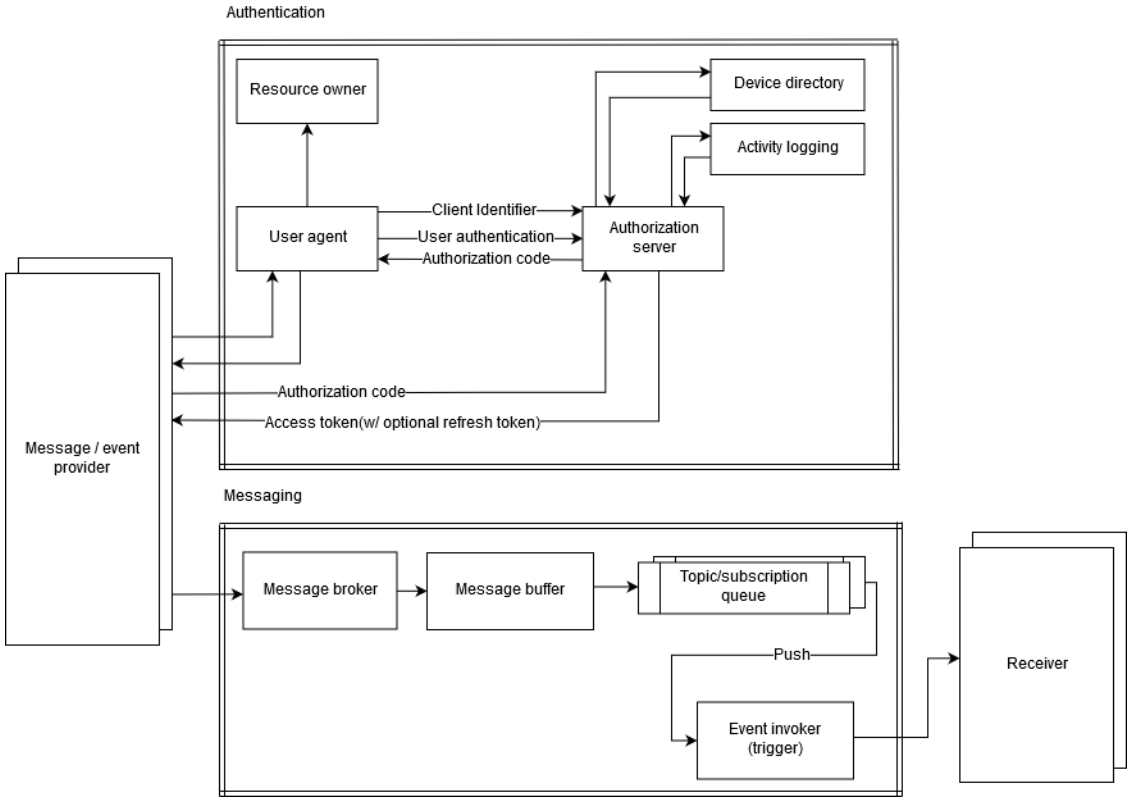

Fig. 6: Functional diagram of the communication infrastructure.

5. An event invoker is used to enable instant notifications for receivers. Instead of traditionally relying on polling for new messages, the event invoker sends a notification when a new message arrives. Event handlers are registered per receiver and are invoked upon action from the provider.
6. Event messaging is used for event notifications, useful to avoid empty document messages that are used as notification and contain as such empty bodies. Event messages are useful for status updates on e.g. ambulances' locations and reduce system load as the messages are relatively small.
7. To achieve integration the client should communicate with the REST API gateway that acts as an entry point for clients and forwards calls from clients. The message broker in the API can relay the requests to back-end services where necessary (such as the authentication module). It can aggregate or decompose messages where necessary, and provide content and topic-based message routing using the publish-subscribe pattern. The API decouples the clients from services, an function such as logging and load balancing is handled by the API. With the use of an API it is avoided that for each partner a separate integration is required and that the communication infrastructure needs to consider the many existing databases, data formats, other APIs etc.
8. The Emergency Care information standard is used to ensure compatible data exchange with third party software vendors, intended to standardize information exchange in the Dutch multi-agency emergency healthcare (section 3.3.3).
9. For the correct functioning of other applications such as data analytics software and back-office systems (e.g. for invoicing and health insurance providers) incoming messages are stored in databases. The message broker creates a copy of the incoming message that is stored in two databases, incident and vehicle. In this way the current database structure and logical separation of to entities is maintained. In addition, when third-party software vendors are using the API an use case can arise where the time-interval for the reception of messages is greater than is desired for a message queue. For instance, when data analytics software is using the API for data extraction there might be no need for real-time updates but only e.g. only once a day. This intervenes with the non-persistent nature of messaging queues, that will congest if messages remain undelivered for most of the day. Therefore, the message store pattern is used. Event messages are not saved individually, but will update the corresponding entity (e.g. ETA) if applicable. This is a functional design decision, however, because the databases are apart from the communication infrastructure they act as a receiver. Therefore, these databases are depicted in the context diagram (figure 5.
10. A device registry is used to keep a list of all connected devices to the API. A provisioning API is used to enable new devices to register themselves.

3.3.3. External interfaces

An important part of a functional architecture design are the external interfaces as they define and handle the data flow and event consumption and emission. Interface definitions considers both the interface syntax (the structure of the data or request) and semantics (its meaning or effect).

The structure of data used in the healthcare sector is since the eighties defined by the Health Level 7 standard (HL7), with version three released in 2005. HL7, as developed by the HL7 foundation, is an ISO 27932 standard for supporting healthcare work flows, its messages and clinical document

structure [3][5]. Many existing legacy systems use this standard, and is adapted and maintained for the Dutch healthcare market by Nictiz. However, it has become too complex, too inconsistent and and not adaptable enough for the increasing variety of platforms, such as web and mobile phones [6]. To counteract this problem, the HL7v3 adoption for the Dutch market was revised into the eSpoed guideline [23]. This guideline formed the basis for the information standard Emergency Care (Dutch: Acute Zorg) [24]. The aim is to standardize information exchange and speed up the process as intermediate translators are not necessary anymore for communication. Agreements are made between the parties within the multi-agency emergency healthcare that data should be made electronically exchangeable, allowing for easier integration of emergency healthcare systems [25]. With this information standard it is standardized what information is included in the electronic information exchange, the data structure format and the entity relationships in between [18]. However, the standard is not adopted by every emergency department yet, and in spite of scoping down the required data entities, the minified set is still large and complex (figure 7) [27].

To provide an overview of the content the following 24 data subjects [3] can be found in the data set: personal patient data, WID check, incident data, initial alarming message, trip data, retrieval data, destination data, mechanisms (e.g. car), working diagnoses, measurements values, treatments, patient placement, primary survey, secondary survey, distance consultancy, agreements with patients, appendices, approval data (e.g. from relatives), comment data, referrer data, triage data, GP data, data from observations, and diagnose data from the emergency department.

However, the specification of the information standards still does not specify any semantics (i.e. the technical standards) for information exchange. This allows for possible variations (and thus fragmentation) in the chosen technical communication implementation among different information providers (e.g. among ambulances), where incorporating a convectional interface definition could lower the risk of fragmentation. REST is such an defined and widespread architectural style that ensures interoperability between computer systems on the internet. When a request over HTTP is made the response can contain a payload with e.g. a JSON file that contains the message to be transmitted, formatted according to the style as defined in the Emergency Care information standard.

3.4. Information viewpoint

The information viewpoint describes the way the systems stores, manipulates, manages, and distributes information. This viewpoint develops a complete but high-level view of static data structure and information flow.

3.4.1. Design rationale

The Entity Relationship Diagram (ERD) provides information about the data that is processed by the communication infrastructure, in this case the emergency care payload that attached to a message and send over the messaging queues. The LSIV and the GMS are the two main sources of emergency trip information (the LSIV provides vehicle information, the GMS incident information). The data format these parties use is also used as the message format that combines information from both sources into one message with the Trip entity as main entity (figure 7). The complete message is send to back-office systems and hospital viewers, where information about incoming ambulances is displayed for emergency departments' staff. Cardinalities are all one to one, to avoid clutter these are not included in the diagram (except for incident to patient, which is one-to-many because multiple patients can be transported at once).

[3]The complete data set including diagrams can be found here

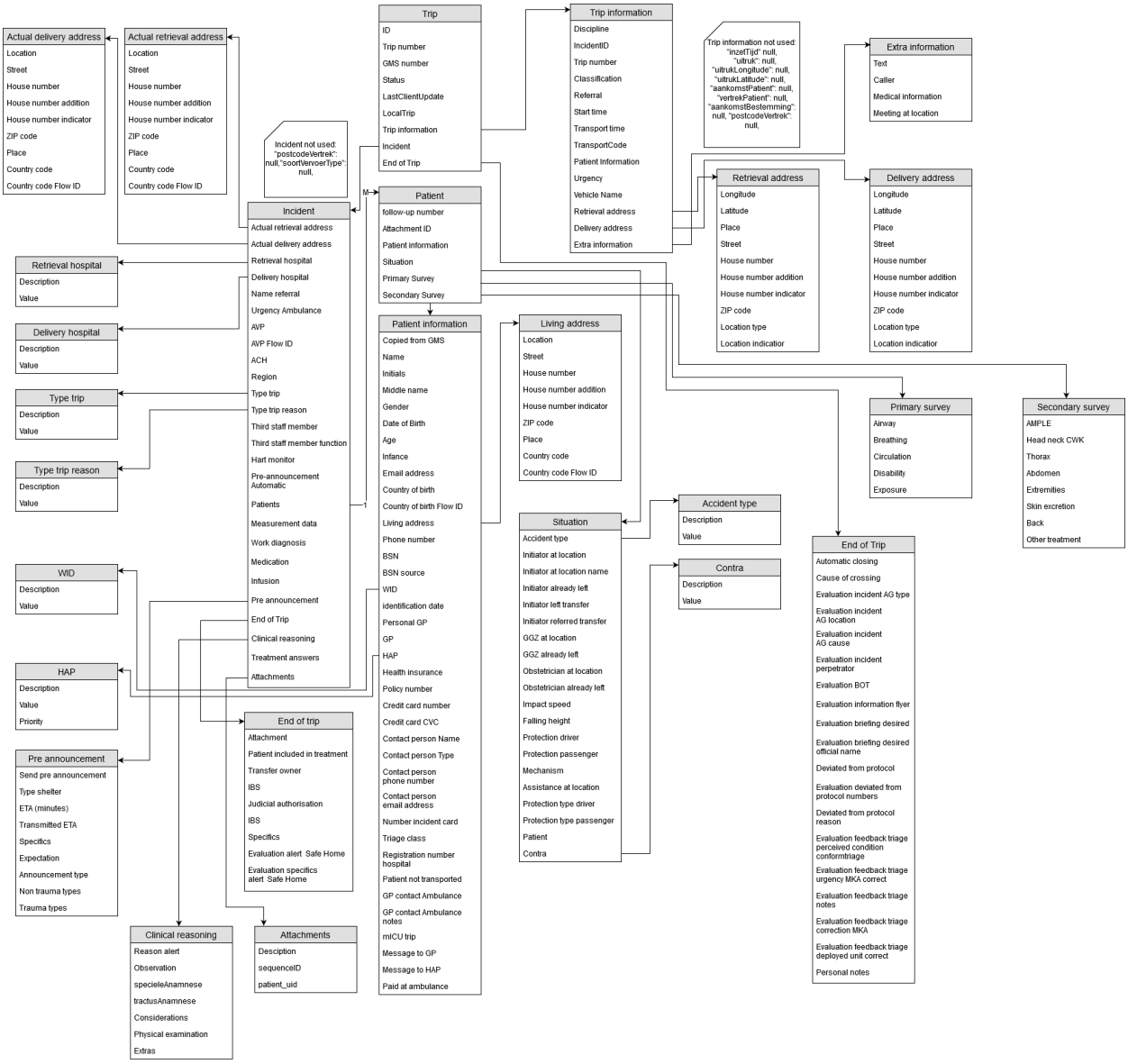

Fig. 7: Entity Relation Diagram of the communication infrastructure.

In the ERD diagram the entity "end of trip" is displayed twice. This is caused by the fact that both the ambulance staff and the emergency department (ED) staff have to close the trip and thus both produce a data entity "end of trip". The former closes their trip when the patient is being transferred to the ED, where the latter occurs when the patient is placed on the ED and their staff confirms and finish the transfer.

3.5. Concurrency viewpoint

The concurrency viewpoint describes the concurrency structure of the system and maps functional elements to state modules to identify the parts of the system that can execute concurrently, and how this is coordinated.

3.5.1. Design rationale

The concurrency viewpoint (figure 8) is captured in a state model diagram, a process flow diagram to be exact. The process flow provides an overview of the message flow and the states the message sender can move between when they offer a message to the communication infrastructure. There are some points of possible failure, e.g. when the sender is not authorized to send messages. Upon successfully meeting the message and sender requirements, the message is accepted, copied to a database and simultaneously send to the receiver. When reception fails, a dead-letter channel ensures that messages are not deleted.

4. Viewpoint validation

treatment validation is performed by applying perspectives on all of the previously created viewpoints. Perspectives are a set of closely related quality requirements (e.g. security) and are used to validate whether the previously designed viewpoints adhere to these quality requirements [31]. When these models do not validate (i.e. they do not adhere to the quality requirement) they are changed accordingly. The changes that are made to make each of the viewpoints valid are described below, the updates made to the viewpoints can be found included in previous section.

Each perspective addresses a list of concerns where a viewpoint should be checked for whether the viewpoint adheres to that perspective [31]. Each of these concerns are applicable for each viewpoint, and with these concerns in mind the viewpoints are checked if these concerns are addressed. Architectural activities describe then how apply the mitigations when the specific concern has not been addressed yet. This application results in a change in the respective viewpoint. Finally, a provided checklist is used to cross off every common misconception or pitfall from the list [31].

4.1. Security

The security perspective is applied to the functional view by adding an authentication provider to the messaging system that checks the authentication of each incoming message before authorizing the message to be send. This authentication model uses the OAuth2.0 protocol to verify incoming messages and issues an access token once a provider has been verified (section 2.14). When tokens are expired the identity needs to be confirmed again, what can be done with a refresh token or re-authentication. The result of applying this perspective and the resulting addition of an authentication provider can be found in the functional diagram (figure 6) and in the process flow diagram (figure 8).

4.2. Performance and scalability

To enhance performance and scalability the authentication module is separated from the messaging system as the micro services approach dictates. Whenever it requires to scale up or scale out, that can be done independently from other services. Not always authentication is required, for instance when authentication happened previously and only a refresh token is send. At first, a transformation adapter was integrated, but applying the perspective showed that it is not feasible to support all message formats and structures. The adapter was soon

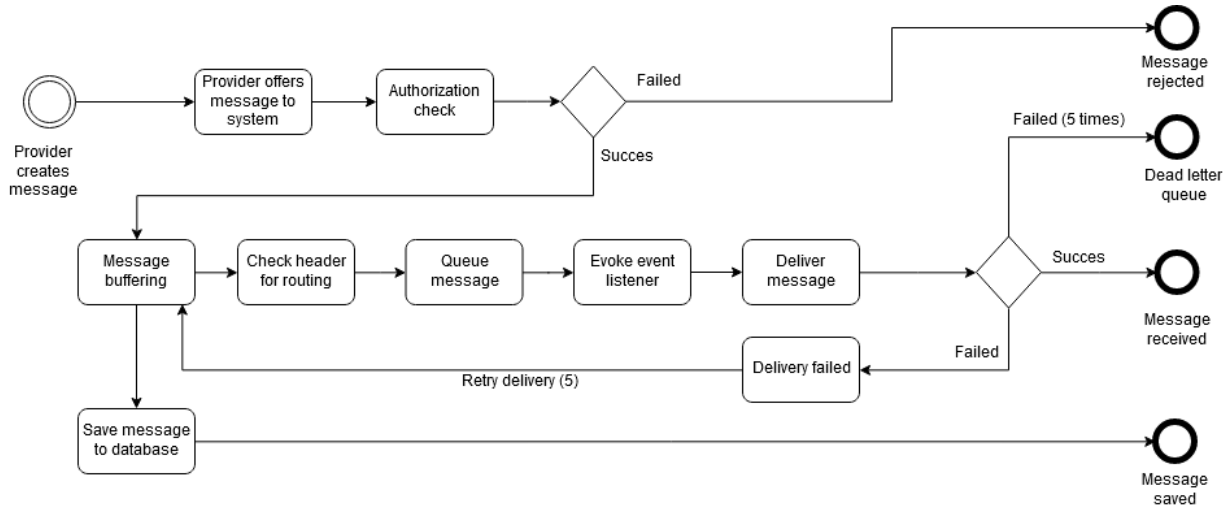

Fig. 8: Process flow diagram of the communication infrastructure.

be expected to grow into one monolithic adapter, for that reason the responsibility for message transformation was moved to adapters from external parties that connect with the API. The result of applying this perspective can be found in the functional diagram (figure 6) and in the process flow diagram (figure 8).

4.3. Availability and resilience

For a communication infrastructure that serves as communication middleware between systems from many parties it is essential to guarantee 24/7 availability and to maintain the system in that way that it is resilient against (un)planned down-time and has disaster recovery in place. Continuous present backup systems, high-availability clusters, load balancing systems, virtualized machines, dockerized system installations and extra VM's on servers in different locations in combination with extensive monitoring and management can prevent a lot of the aforementioned (un)planned downtime or system failures [31]. However, this will require an extensive investment to setup and configure all this hardware and software, which is a great challenge for an on-premise setup. Guaranteed 24/7 availability is, however, necessary due to the nature of the emergency healthcare sector, where critical messages are send 24/7 and systems need to function at every moment of the day.

4.4. Evolution and maintainability

Evolution copes with the system's possibility to change when necessary, e.g. when requirements or the environment change. Four dimensions of change need to be considered for a system in general: functional evolution, integration evolution, platform evolution, and growth. Functional evolution adds functions to the system itself. Integration evolution occurs when the system needs to connect new partners that are (found to be) vital for the business that would or cannot access the API. Platform evolution occurs when other platforms are used instead of the original one where the system was developed for. Growth occurs when more users are using the system and its API, and thus sending more data through the communication structure. All these four dimensions occur regularly, and are therefore important to consider. Functional evolu-

tion is made easier because of the modular design where the adapters and authentication modules are separated from the communication infrastructure. Integration evolution is possible for the same reason, modular designed adapters handle integration. For platform evolution different hardware and/or software is required that needs to be installed and managed, adding to the operational expenses. The communication infrastructure can change platforms as the design is platform-independent. When growing, it needs to be anticipated when e.g. extra servers and/or extra hardware such as load balancing systems are required when scaling out, and this needs to be managed at run-time as well. For both applies that they cannot foreseen at initial design (only a forecast) and form uncertainties when developing and deploying the system.

5. Implementation

The designed treatment as described in the architectural description is implemented here. It is described which key decisions are made for the treatment implementation and in which context the implementation takes place. Also, a walk-through is provided of the entire implementation process, describing each of the applications that are created or configured. Finally, it analyses any trade-offs between the implementation and the architectural description, and analysis the impact of the implementation in AmbuSuite.

5.1. Key decisions

Where many options for implementing the designed architecture are possible, it is decided to remain close to the existing technology stack within AmbuSuite (appendix section 8.1), allowing for easier adoption from the implementation decisions than when e.g. a different technique is used. Then, both a different technique must be learned and the existing functionality must be changed by the developers, putting an increased burden on the team where this can be avoided. It is, however, checked if the technologies in the existing technology stack are sufficient to implement the communication infrastructure, and are expanded upon where it ought to be necessary.

1. Applying the perspectives on the viewpoints made clear that the implementation of the communication infrastructure must to be available 24/7 and be resilient due to the nature of the emergency healthcare sector. It also made clear that extensive measures in among others operations and management must be taken to guarantee availability, resilience, performance and scalability. However, it is to expensive to maintain an on-premise server park with extensive monitoring for a small-sized team as AmbuSuite. For that reason, Azure is chosen host the communication infrastructure, to migrate existing applications where necessary and as such create a Integration Platform as a Service (iPaaS) solution (section 2.13). When using iPaaS, platform evolution and growth are taken care of as platform evolution is managed by Azure (both software and hardware) and the applications are operating in docks or in virtual machines. Growth is easier when operating from the cloud as it is easier to scale up or out with pay-per-usage pricing models.
2. In the light of the design choice to use a messaging queue 3.3.2, the Azure Service Bus is chosen to serve as the messaging queue service as is not desirable to design a messaging service from scratch [14]. Designing from scratch would differentiate from the team's core focus and writing applications that cannot match the performance from solutions already available in the market when it comes to among others availability, reliability and response time. The service bus is used to send messages between applications, both internal as external, to adhere to the send and forget principle. The service bus uses a publisher/subscribe pattern (section

2.11.2, where applications can subscribe themselves on published messages regarding their interest.

3. As security system development is a function that requires extensive security knowledge, it is not desired to develop an security implementation of e.g. OAuth but rather use an existing solution [31]. Therefore, with the use of Azure Active Directory (AD), an off-the-shelf OAuth2.0 authentication module is ready to use. AD also retains logs of connected devices and takes care of user management.
4. Application adapters retrieve information from external sources (such as the LSIV) that use legacy systems and are not data complaint. These adapters are independent web jobs with the single purpose to retrieve information from external sources, perform mapping to AmbuSuite's data model, and finally send it over the service bus. These adapters are placed in Azure Service Fabric that enables the simplification of micro services development and application life cycle management, as well as reliable scaling. However, if they do not require extra software to run (e.g. SDKs) then they are placed within Azure Functions, thus enabling serverless computing.
5. Daily monitoring is performed by Azure Application Insights. This application provides direct insight in the daily usage of application, generates error logs and performs monitoring on critical applications' functioning.
6. The Azure Application Programming Interface Management service (APIM) is used to expose endpoints from applications to third parties. The advantage of using APIM is that most of the authentication is handled by Azure services, much of the complexity abstracted, and API-calls can be executed relatively easily. APIM can provide a layer around endpoints in existing applications and can expose these endpoints to the outside world without providing information about the applications behind the API. As such it forms an easy accessible abstraction layer around those endpoints as only API-keys are necessary for authentication (section 5.4.2).
7. To meet the quality property requirement of throughput, the message header is required to contain flags that indicate the message body format and token type. If they are not matching the desired type that the subscriber expects (e.g. data structure of the message is different) the message is declined. With the use of an authorization header, throughput (defined as the amount of workload a system is capable of handling in a unit time period [31]) is increased as no invalid or unauthorized messages are being processed.
8. To save incoming messages from the messaging queue a copy of message is made and saved in Azure SQL database. This database acts as an access point for applications that do not require information immediately.
9. To remain close to AmbuSuite's technology stack that is developed in .NET, applications are written in C# and use the model-view-controller architectural pattern.

These key decisions made may enable the setup of the communication infrastructure. However, to field-test whether the communication infrastructure meets the desired user and performance requirements when put to use, an integration with a AmbuSuite partner can provide insight about whether it meets users' demands and performs as intended. For this test the software vendor RescueTrack is integrated with the communication infrastructure to field-test the performance regarding an accurate ambulance ETA at emergency departments [36].

5.2. RescueTrack

RescueTrack[4] is a software vendor from Germany that provides hardware and software for emergency services (police, fire department, ambulances, helicopters, dispatch centers and hospitals. Software for the dispatch centers evolves around critical emergency service planning, trip scheduling and resource management. In hospitals, it handles patient management and transport ordering. As it supplies the entire emergency healthcare it is an important player in Germany. As their business contains overlap with AmbuSuite only the navigation unit for emergency services (specifically: ambulances) is interesting as with that device GPS locations are logged and transmitted in real-time. This real-time transmission allows for Estimated Time of Arrivals (ETAs) that are not estimated, but accurate to the second.

5.3. Use case: providing real-time ETAs

The use case where RescueTrack's real-time ETA updates provides a solution for is the scenario of distributing workloads of emergency physicians over incoming patients [36]. At the moment this distribution is based on the ambulances ETA, but this ETA is often inaccurate. Teams of emergency departments' staff often continue with their busy work when a patient does not arrive at the estimated time as they want to avoid waiting time for existing patients awaiting treatment. When emergency department's staff is gathered for the arrival of a patient, the average waiting time for patients already being treated at the emergency department increases with 16 minutes [22]. Each arrival of a patient is therefore a costly manner and interruptions should therefore be timed as accurate as possible. However, patients arrive on average 4:17 minutes later at the emergency department (SD: 7:23), but 25% arrive 2:44 minutes earlier [36]. As such, arriving patients often face an incomplete or no team to accommodate them as the teams are respectively disbanded or not complete yet, creating a patient safety problem at the emergency department. The patients present at the emergency department risk a safety problem because adequate care is lacking, and the incoming patient risks a safety problem as no team is present upon arrival. Therefore, to optimize the staff's workflow and guarantee patients' safety, the ambulance's ETA should therefore be accurate to the minute.

In a pilot in Noord-Brabant, ambulances are provided with RescueTrack's navigation units that transmit GPS-locations in real-time. From these locations a ETA can be calculated, that is made available through the RescueTrack API. These ETAs are send to the hospitals in that region that are also included in the pilot. The test only comprehends the functional testing of the communication infrastructure and not the impact at the emergency department's staff [5].

5.4. Implementation process

The implementation process of the communication infrastructure breaks apart in six different instances, with some instances part of the communication infrastructure and some part of the integration with AmbuSuite's application suite. The order of the applications as described below follows the information flow from the source to the hospital viewer application (figure 9). The information process flow branches at the moment a copy of the message is made and saved in the central database storage.

5.4.1. RescueTrack adapter

An adapter is created as a .NET console application and placed on Azure Service Fabric to retrieve data from RescueTrack, the first partner from AmbuSuite that is going to use the communication infrastructure. The adapter is not part of the communication infrastructure, but is created to integrate the partner RescueTrack and assist in the

[4] `https://rescuetrack.de/`

[5] An intervention study will be done at Radboud UMC to test the impact on the hospital teams.

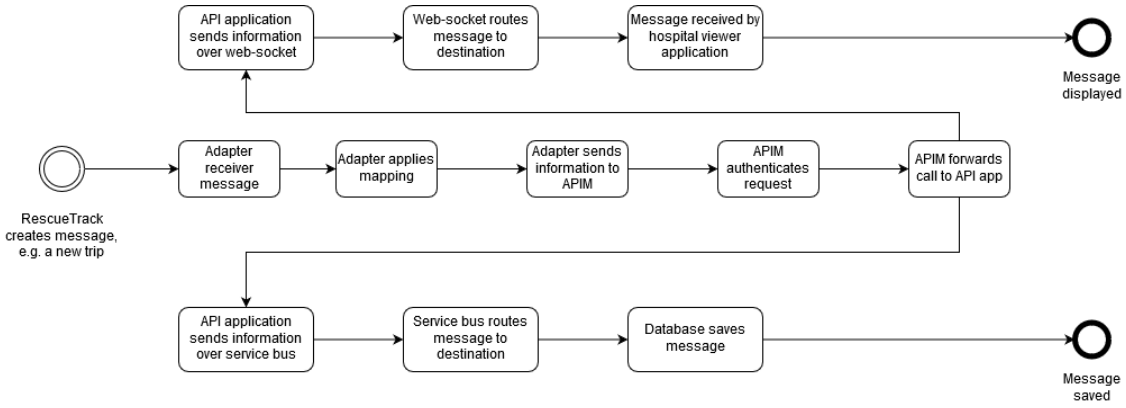

Fig. 9: Information process flow of the implemented treatment.

first use case. Usually, these adapters are meant to be written by developers from external parties, but for this use case is developed by AmbuSuite instead. The adapter polls for data on the RescueTrack REST end-point, specifying the ambulance number and trip number. Based on this information, the identified ambulance is returned in combination with the requested mission data. For this use case the adapter extracts the ETA from the mission data and serializes the array to JSON. Finally, an HTTP-client is instantiated containing the API-key for the Azure API management service; a data-format specifier; and APIM's web-URL. If the transmission succeeded it returns an OK-status.

5.4.2. API Management service (APIM)

Within Azure, Microsoft's APIM is used to expose the designed APIs. The advantage of using APIM is that most of the authentication is handled by Azure services with much of the complexity hidden away. On conceptual level the information flow (figure 10) starts when a request from an external party is received. Their request package arrives at APIM, that authenticates the subscription key and finds its connected products (a product is a composition of APIs that is offered, including quota and rate limits). It returns which APIs the external party has access to, and from those APIs, its available operations. Once the request is authenticated, the APIM authorizes itself against Azure Active Directory (AAD). Because APIM has been registered initially as trusted Azure Service, it is able to obtain an access token for the specified resource by authenticating itself against AAD. Upon success it receives from AAD an access token with the correct audience for the specific resource in combination with a refresh token (the OAuth 2.0 approach is used here). As good security practice preaches, these credentials with which these tokens are retrieved are refreshed on quarterly basis by Azure. The request URI is then changed to the back-end app URI with the obtained access token included in the request's authentication header (type bearer). The request is forwarded to the back-end the request and executed (for instance a POST operation at the Flow API) and the result is routed back to the requesting party. All this time, the only authentication method the external party uses is the subscription key (this can be configured with an optional OAuth implementation). The only URI that is visible to the requesting party is an AmbuSuite's URI or a custom domain.

When setting up an APIM service an initial registration in Azure Key Vault is required for authentication against AAD. In the Key Vault the

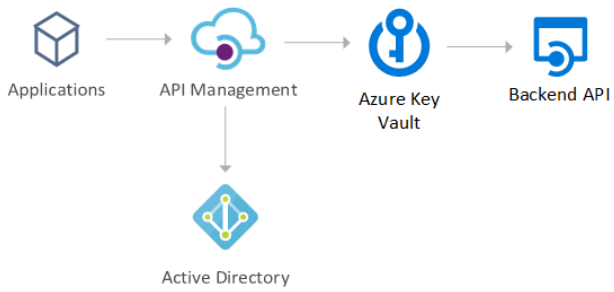

Fig. 10: APIM process flow

APIM service is registered with a service principal and all permissions available. These permissions detail which operations (such as GET or POST) the APIM is allowed to perform. Within the APIM interface, existing API applications hosted on Azure can be imported, either exposing a back-end API application or a HTTPS endpoint (REST/SOAP). Once the API is imported an XML-based policy defines its behavior (see the appendix section 8.4 for an example). Policies in APIM can be used for a wide variety of tasks, including handling the authentication between APIM and the back-end by requesting access tokens from AAD. In this case his is done by using managed identities where APIM itself will request an access token for the specified resource. The obtained access token is sent as bearer token in an authorization header to the API app. The created policy can be placed in one of the two available scopes, within this context of this API either the API scope or the operation scope. If the policy is applied to the API scope, the content of the policy applies to all operations in the API. In contrast, when a policy is applied to one operation, it is applicable to that operation only.

5.4.3. AmbuAPI

The RescueTrack adapter has send the data about the ambulance's ETA to the APIM, that acted as an intermediate layer between the adapter and the back-end API application. APIM has forwarded the call to the back-end, where the API application (.NET Core MVC) AmbuAPI is exposed as a REST end-point and is listening for incoming requests. Two POST operation for this use case are present [6], one that posts the entire trip to the hospital viewer, and one generic operation that updates the trip with any information that had come in. Both POST operations call the function MessageBrokerAsync to perform two main functions. First, the received data is saved in the MVC's view context. ASP.NET core MVC does not use model binding, and only depend on the data provided when calling into it. In this way a chunk of the request can be rendered instead of a complete response. The view context uses session state management to track updates on a model's entity state while the application is being used, using local application storage to persist data across requests from a client. The partial request is then send over a web-socket to update the hospital viewer application. In that way essential information is forwarded as soon as it is received as a web-socket offers a lightweight connection for communication. Second, the same request data is attached to a message, with the body formatted in JSON. The message broker initializes the service bus and offers the message to the service bus.

5.4.4. Service bus

The Azure service bus is a cloud-based messaging queue (section 2.11.2) that receives messages from the AmbuAPI application and sends these messages to the database. The service bus acts in this way as a load balancing service for the database; avoids application locks on tables in the database; provides a better foundation for scaling and evolution; and enables separation of concerns. The service bus supports two-way communication (writing and reading the database) and uses event-invokers at the bus's end-point to avoid frequent queries on the database and useless service bus polling. With the event-

[6] The API comprehends more functions, see appendix section 8.5

invoker, the listener in the database application can avoid frequently polling the service bus for data updates or requests. Vice versa, the service bus does not have to query the database periodically for changes, but only when events are invoked that indicate data changes, efficiently supporting a significantly higher volume of concurrent messages.

5.4.5. Database application

The database application manages an Azure SQL database service and act as the worker for all database operations. It receives information from the service bus upon invocation and sends messages to the service bus when information from the database needs to be shared, where for it queries the database. The database application is the application that distributes all incoming information over the two databases (vehicle and incident) to create logical separation. This separation is used to reduce the amount of duplicate data that is stored, and the database worker queries both databases for information where requested. The database application keeps track of the database information in an index in addition to the databases' indices to improve the speed of input/output operations. Furthermore the application provides caching for recent trips (Redis cache) where the trip has not been closed yet (which can be seen in the trip status) to further speed up the process of reading and writing trip information to the database. The application is designed to become the future central database where the existing databases are centralized into and saves information compliant to the Emergency Care standard (section 3.3.3).

5.4.6. Hospital viewer

The hospital viewer is in this use case the already existing application AmbuView (written in Angular), used in emergency departments in hospitals to display incoming ambulances ambulances, their estimated times of arrival (ETAs), and patients' information. Existing applications that require integration come with legacy functions, in this case the hospital-viewer requires a web-socket to receive data. The API application AmbuAPI has implemented the web-socket as well to avoid a rewrite of the viewer.

5.5. Implementation trade-offs

There is one trade-off made in the implementation compared to the architectural design of the communication infrastructure. The message buffer as visible in figure 6 is removed as a separate entity because it is included in the service bus. The Azure service bus solution offers a messaging queue in combination with buffering, end-points etc. and removes thus the need for an extra message buffer object to be included in the communication infrastructure.

5.6. Impact analysis

To implement the aforementioned decisions (section 5.1) made for the integration of the communication infrastructure within AmbuSuite's applications, an impact analysis of AmbuSuite's application landscape highlights where changes are necessary. In its current state AmbuSuite's applications are strongly coupled with sometimes multiple dependencies on each other. Or in case of AmbuReports, there are no communication lines, and data is transferred to AmbuReports by a table copy between databases. In order to support the new communication infrastructure, the current communication lines from each application need to be rewritten so that messages are first routed trough the service bus before providing or retrieving information from each other (1). The same applies for the adapters, which are now directly linked to one of the applications. These adapters are moved to Azure Service Fabric to enable scalable micro services development and application life cycle management (2). Incoming data from these adapters is routed over Azure Service Bus (3) before arriving where needed. These adapters (and other applications) save incident and vehicle information in databases decoupled from applications, save for temporary local storage and

caching (4). The service bus is used for communication between the applications and the databases. The communication infrastructure is event-driven, and incoming information is send immediately to each respective application. To save database waiting times, a copy from the message that is send over the service bus is send to the database (5), so that applications from external partners can use the API (6) to retrieve that information as well. With this implementation the non-persistent nature of the service bus is avoided. With this construction in place, AmbuSuite can act as a medium or adapter to pass through information that comes directly from the LSIV/GMS. Also, AmbuReports is connected to Azure Service bus to retrieve information directly from the propriety databases (7), thereby replacing the database copy operations. For information exchange with external parties the standard Emergency Care is used (8), that differs from the internal data format used.

The impact analysis section lists more items more than this infrastructure's implementation encompasses. With this implementation the communication infrastructure is created, but existing AmbuSuite's applications are not completely connected yet. Changes required for further integration are placed on the road map and scheduled for next year.

6. Evaluation

In this section the process evaluation reflects on the architectural design process and whether it yielded the expected results. It consists of a methodology evaluation; a consistency check against the literature whether the literature provided in-practice applicable theory; and an implementation evaluation. Finally, it is assessed to what extent the implementation solves the communication problems that are present in the emergency healthcare.

6.1. Methodology reflection

The methodology to design an architectural design consisted of creating viewpoints and applying perspectives on the models resulting from these viewpoints, based on [31]. This methodology was recommended opposed to creating one monolith architectural design, but instead capturing the design from multiple directions. Each of the directions captured a separate aspect of the architecture, and as such resulted into four viewpoints, including corresponding four models. As the implementation is according to design, expect for one element (the message buffer, see below) it can be stated that the methodology for designing architectures yielded the expected results with minimal trade-offs between the design and the actual implementation. The focus during design time was put on modularization, with distinct applications for different functionalities, each of them with a clearly defined task as the micro services approach dictates. As such, the created applications were loosely coupled with each other and high in cohesion. As can be seen in the architectural description (section 3), the application of the perspectives had a significant impact on the design because of the applied availability and resilience perspective. In the application of this perspective it became clear that AmbuSuite could not fulfil the availability and resilience requirements alone therefore the decision was made to transition to a cloud service provider (section 5.1). Revisiting the models with a (set of) quality requirements enabled continuous revisions of the created models by iterating over them again each time with a new set of quality requirements. These continuous revisions contributed towards the minimal trade-offs between design and implementation, thus avoiding the common pitfall of architectural erosion (occurs when the architecture's implementation drifts away from its requirements [31]).

6.2. Literature reflection

Enterprise Integration Patterns (EIP) formed the cornerstone of the literature section as these pat-

terns describe the communication possibilities between enterprises. Enterprise Integration Patterns heavily rely on messaging middleware to enable inter-enterprise communication with the main advantage that messaging enables the send and forget principle. That principle incorporated the loose coupling between systems, making an approach such as enterprise application integration avoidable. Enterprise Integration Patterns are reusable forms of solutions for design problems, describing why a particular situation causes problems, and how the components of the pattern relate to each other to give the solution. A pattern as such encapsulate knowledge that can be reused in situations, providing a generic approach that can be applied on various situations, allowing for a different outcome each time. Applied within the context of the created communication infrastructure Enterprise Integration Patterns provided knowledge regarding what treatment to use (messaging over remote procedure calls, file transfer or shared databases). Also, it provided knowledge regarding the advantages and possible pitfalls regarding messaging (e.g. separation of concerns and multiple platforms integration versus a complex programming model and synchronisation issues) allowing for a proper consideration and trade-off analysis of which communication medium to choose. Furthermore, along with the patterns tactics were provided for implementation so that good practices were followed and common pitfalls avoided. For instance, the provision of code highlighted code structure that explained the usefulness of cleaning up resources manually to avoid synchronization problems with asynchronous operations. In short, applying the Enterprise Integration Patterns provided good reusable knowledge and a practical approach for designing a communication infrastructure between enterprises.

6.3. Engineering cycle reflection

There is a discrepancy between what Wieringa's engineering cycles prescribes and how certain treatment steps are executed, based on two accounts [38]. First, the software architecture design methodology uses perspectives as a validation of created models belonging to the viewpoints [31]. Wieringa, however, describes validation as the investigation of the effects of the interaction between a prototype and a model of the problem context and of comparing these with requirements on the treatment [38]. The validation step of the engineering cycle should thus have consisted of a theory of the prototype's interaction within the problem context. This theory exists in section 5.6, but is made after the propriety choices for the implementation are made and is rather analyzing the impact of the implementation than the architectural design. The reason for this is that Rozanski provided a valid validation approach with the use of perspectives, where validation practices and methodologies were listed for usage. These validation practices and methodologies were specifically designed for software architecture design and form a coherent whole in combination with the viewpoints. In addition, using this validation approach yield validations that are directly applicable on the designed models, resulting in changes that that affect the functionality and behavior of the designed communication infrastructure. Therefore, the validation method from Rozanski was favored over Wieringa's. A table tracing the perspectives with the full-filled criteria can be found in the appendix (table 3).

Second, the treatment evaluation is found to be insufficient as no field test results are in yet and is therefore as such not complete. A treatment evaluation would normally consist of the investigation of a treatment as applied by stakeholders in the field, but consists here of reflections on the methodologies and literature used [38]. In this evaluation, there is the benefit of hindsight that enables aforementioned reflections, but the results from the tests with stakeholders in the field are not in yet (section 6.5 for more).

6.4. Implementation validation

The implementation of the communication architecture is used as an practical validation of the architectural design. Only one trade-off was found, that the need for a message buffer was removed with the use of Azure Service bus that already came with a message buffer included (section 5.5). A use case for a field-test was found in the partner implementation of RescueTrack with a pilot launched in fall 2019. To test whether it behaves as desired the implementation is checked against the requirements. To recap, the requirements are as follows: unified messaging; reliable communication; provisioning and monitoring; dynamic scaling; flexible services; secure communication; and integration with other tools.

6.4.1. Unified messaging

The requirement of unified messaging is met because of the use of a messaging queue with a publish/subscribe pattern that also supports any message type. A messaging queue enables information transfer between systems without requiring integration, and as such avoids high coupling between systems. In turn, this allows for all types of system to communicate with each other, including the opportunity to send any type of message. The only limitation with the message type is that the message format needs to be specific for the Azure service bus, otherwise the service cannot extract routing information. The payload however, can be any format as desired, but JSON is preferred. To maintain uniformity, XML (WDSL/SOAP) code examples for adapters to transform messages to JSON are provided. In addition, the API exposes the service bus's functions and as such, enables system integration with the service bus.

6.4.2. Reliable communication

The requirement of reliable communication is met by hosting the messaging queue on Azure. The cloud service provider removes the need for infrastructure considerations, and provides a minimum availability and up-time percentage. Any infrastructure failure of any kind is taken care of by Azure, and if some of the hardware fails, Azure dynamically swaps servers, communication lanes or even entire server parks. As the up-time is guaranteed, communication in the respect of hardware is therefore reliable considered reliable. In addition to that, the dead-letter channel on the Azure service bus and the message acknowledge function (section 2.11.2) ensure that in case of software failure, messages are guaranteed to be saved and can be delivered later on.

6.4.3. Provisioning and monitoring

Provisioning was defined as providing users with access to data and resources, referring to all information management systems involved. The requirement of provisioning is met by using the Azure service bus and the creation of an API that enables the access to available resources. The database whereto information is copied upon reception provides data availability and access for users, even when the timing of accessing the data differs. The requirement of monitoring is met by using Azure Application Insights, that provides in depth information about performance; throughput; response time; general availability; and up-time. This enables in-depth analysis of performance bottlenecks or any failures. Average of 50 requests is 2.56 sec to finish the request, with the note that this is measured on a test-server, production environments tend to be equipped with better hardware and are as such faster. The request start at retrieving data from the RescueTrack server, applying transformation on the format (XML/SOAP to JSON), sending the information through APIM management console where it is saved in the database. The database takes up most of the time with 1.12 seconds to process the results.

6.4.4. Dynamic scaling

The requirement of dynamic scaling is met by using a cloud service provider that supports pay pay-

per-usage pricing models. Scaling up (adding more hardware to a machine to improve performance) or scaling out (adding more machines) is done through the Azure portal. It can even be done automatically, dynamically when the situation requires in e.g. peak hours. The communication infrastructure is modular by design, with separate services for authentication and transformation.

6.4.5. Flexible services

The requirement of flexible services is met by not requiring any specific type of system integration because of the use of an API in front of the messaging services. Adapters that are written can connect to the API, but there are no requirements for the adapters since they are written by external developers. This approach enables flexible services as any type of system can integrate an adapter that connects with the communication infrastructure's API. In this way it is avoided that communication can only take place between a selected number of systems.

6.4.6. Secure communication

The requirement of secure communication is met by the adaption of an OAuth-based authentication provider. The authentication provider mandates that adapters first obtain an access token and manages in this way who gets access to which resources (e.g. to which APIs or databases access can be granted). In this way only authorized users can be granted access to sensitive resources such as medical patient data. Azure is certified with the necessary certifications guarantee the safety of that data. With the use of the Azure service bus, secure communication is guaranteed as well.

6.4.7. Integration with other tools

The requirement of tool integration is met by providing a REST API. Developers from external parties can after permission write adapters that integrate their system with the communication infrastructure and have documentation and support provided. Integration with these adapters is relatively easy as only a provided API-key and a managed identity are required to communicate with the APIM.

6.5. Impact

The architectural design and its implementation are effective once it solves the communication problems the emergency healthcare is facing nowadays. To recap, the main communication problems they are facing (as stated in the introduction) are the following: critical patient information is often missing; patient's medical records do not arrive on time; overhead is present in current communication; and medical records do not transcend the disciplines' boundaries. Unfortunately, within the current stage of the field test it is hard to state whether the present potential to solve these communication problems is fulfilled. In any case, integration is made possible to break through the boundaries that are located between the emergency healthcare disciplines and enable inter-enterprise communication (figure 3). The field test with RescueTrack has not yet been completed, but preliminary results show enthusiastic responses from the ambulances staff that find the software easy to use, complete and fast. This holds however, that treatment evaluation cannot be fully completed yet as the evaluation from the stakeholders in the field is lacking. In the hospital the use case for the field test was the distribution of emergency physicians' workloads over incoming patients. This field test provided the hospitals with technical foundation for a communication infrastructure that enabled displaying real-time updates on the ambulances' ETAs. However, to measure the impact that an real-time ETA has on the distribution on emergency physicians' workloads an separate intervention study will be done at Radboud UMC. Measuring the effect on a team is outside the scope of this research and no expertise to perform such a measurement is available.

7. Conclusion and discussion

This chapter concludes the thesis and summarizes all findings by answering the main research question which is stated as follows:

RQ1: What architectural design can facilitate an optimal information delivery in the Dutch multi-agency emergency healthcare?

The research question explored what architectural design could facilitate an optimal delivery and resulted in a communication infrastructure for the emergency healthcare. An implementation was made for the architectural design in combination with a field-test with the partner RescueTrack. Where results of this field test are not available yet, the implementation served as a practical validation of the created architectural design with almost no trade-offs found. The architectural design that could facilitate an optimal information delivery consisted of a communication infrastructure that included a messaging queuing system, adapters and an API. To ultimately create the architectural design an approach based on the engineering cycle first investigated the literature to discover the stakeholders present in the emergency healthcare market; their requirements regarding inter-enterprise communication; and software architecture for architecture design [38]. Next, a treatment was designed that comprehended a communication infrastructure based on Rozanski's guidelines for software architecture design [31]. The validation of the designed treatment with the application of perspectives took place according to the same guidelines. An implementation was created that integrated RescueTrack with AmbuSuite, for which a field test was setup. Also, the implementation served as an evaluation of the architectural design. The evaluation showed that the followed methodology yielded the expected results, with a notable role for the perspectives, that in iterations revisited the created viewpoints and adapted them according to the validation results. Thanks to that minimal trade-offs between the design and the implementation were found. The application of Enterprise Integration Patterns proved to provide a solid foundation for a from-theory derived practical approach to incorporate communication infrastructures in software architecture. An analysis of the implementation showed that the implementation satisfies the requirements as provided by the stakeholders, demonstrating that integration can take place as they have desired. As previously noted (section 6.5) it is too early to tell whether the communication infrastructure can fulfil its potential of enabling inter-enterprise communication in the emergency healthcare sector. Where the technical foundation has been laid out, results of the field-test still have to show if communication between the ambulance and the emergency department has been improved and how emergency department teams are effected (more on this in future work). Foremost, this study provides technical means for communication in multi-agency emergency healthcare, now it is up to users to integrate their systems with the communication infrastructure.

7.1. Scientific implications

Inter-enterprise communications are still fairly new, and a literature gap exists on this topic as no research is available that connect the fields of enterprise collaboration/communication and enterprise architecture. Companies must manage the increasing technological complexities while they add value to business processes through the strategic alignment between business and IT. At the same time, companies must achieve integration and coordinate their processes with their partners in the sector or supply chain in the pursuit of efficiency [35]. This study shows that regardless of the legacy systems present integration between enterprises is possible by offering a generalistic communication infrastructure. Instead of only theorizing such a concept, an implementation showcases the practical validation of such a communication infrastructure. The the-

ory of designing the communication infrastructure is based on two theoretical methodologies, the Enterprise Integration Patterns (EIP) methodology and software systems architecture methodology.

Enterprise integration patterns offer patterns that encapsulate reusable knowledge and act as a template for problem-solving, and EIP focuses on offering templates for enterprise software integration. The application of these patterns offers insight for other researchers how these patterns can be used in inter-enterprise communication, both in the architectural design process as the result in the implementation. Knowledge can be obtained by studying the applied patterns and how they attempt to resolve the communication problems between enterprises.

The software systems architecture methodology is more hardened in practice as the aim was to deliver a practical approach for software system design [31]. Other researchers can examine the software design methodology and how its resulting system design yields the expected results in the implementation according to design with minimal tradeoffs. Knowledge can be obtained by studying the applied viewpoints and perspectives and analyzing how these are applied.

The implementation shows that both methodologies are suitable for use in practice and yield valid, usable results. If researchers are investigating how inter-enterprise communication problems can be resolved (with an IT-solution) they find practical validations of two tested methodologies with evaluations discussing the practical advantages and disadvantages. Often methodologies are not field-tested, e.g. from the engineering cycle the implementation step is skipped [38], resulting in a gap between theory and practice.

Other researchers can use this study to investigate what methodologies (enterprise integration patterns and software systems architecture) are validated and work in practice when used to design software architecture. They can also learn how to solve communication problems as the communication infrastructure in a more general sense can be applied to other use cases as well, e.g. to other (non-medical) sectors that suffer from legacy systems that need to share information. Therefore, the architectural design is not healthcare specific and is generalizable to other domains. Web services are an ideal implementation platform for integrating disparate legacy systems because they are platform-independent (especially when using software as a service models). Enterprise integration patterns (EIP) represent achievable design solutions that may be used to construct these enterprise integration solutions [34].

What was unknown until this point was how inter-enterprise communication could be realized as many differences between enterprises exists regarding their data formats, interfaces, legacy systems [14]. Attempts have been undertaken, but are mostly focused within an enterprise and not between enterprises [39]. Adaptions in larger companies also struggle with security and hindering business process characteristic for larger companies. With the rise of cloud computing, delivering infrastructure as a service to facilitate integration and information sharing trough web-services have been made easier [7]. This study therefore shows that inter-enterprise communication is an achievable goal to strive after, helped by the use of infrastructure as a service model that simplified implementing technical ways for inter-enterprise communications.

With the implementation created the question rises whether the patterns, architecture and implementation are not designed for cloud computing specifically, and Azure in specific. Designing data-intensive applications [17] stated that designing distributed systems is quite difficult and off-the-shelf solutions are the most viable. The same applied for Enterprise Integration Patterns (use an exist-

ing messaging queue) and software systems architecture (do not design security facilities yourself), where in the beginning the intentions was to develop everything in-house [14][31]. With especially the availability and resilience perspective applied on the architectural design, it became clear that besides in-house developing even in-house hosting was not viable. Therefore the decision was made to use Azure as a hosting platform, also because Microsoft is certified according to data protection regulations. Other cloud service providers do lack some of the required healthcare certifications and were therefore not an option as data security is rigidly enforced by Dutch legislation.

Certifications for data regulations are important in the (emergency) healthcare domain as patient's medical information is sensitive data. Data is therefore stored in European data center and is not meant to leave Europe, according to the GDPR. Furthermore, data is not exposed unless partners have signed agreements with AmbuSuite and are certified according Dutch certification programs for data protection in healthcare. Even then, only the bare minimum of data is send to the partners, where possible removing medical data and all other data that is not relevant for the business functioning of AmbuSuite's partners. Even AmbuSuite itself cannot access (medical) data easily but data is often masked. The trade-off for ensuring the least amount of data sharing is that sometimes data can be made available to help the patient, but is not allowed by law to use that data. Or, when a partner is insufficiently certified, that these partners are not allowed to receive any data anymore, effectively preventing them from proper functioning. These practices are results of the certification processes and apply as such for each party that stores or transfers data within the healthcare sector. The data storage methodology is therefore, besides some minor details, the same for each party.

7.2. Limitations

Limitations of this study are mostly related to the field wherein this study was conducted. The emergency healthcare is often described as "being in its infancy", visible in the lack of integration of any kind, not necessarily technical alone but also in the lack of cooperation between disciplines [29]. It was also visible in the lack of information present in the emergency healthcare sector, that encompasses the communication problems that has been studied here, but also the information about the disciplines available in literature and documentation. Therefore it was problematic to find relevant literature as some literature was available in Dutch, but scare. Or literature was available in English, but was not applicable on the Dutch market. Sometimes papers has been locked behind (pay)walls of UMCs, a result of the second limitation: the independence of each emergency healthcare discipline. Each discipline can function as a stand-alone unit and be self-sufficient, being able to provide adequate care in their discipline. However, there is little to none incentive to share information with other disciplines, where this since the seventies is a topic of improvement from the government [8]. This is partly the result of the current financing structure in the emergency healthcare, where each discipline is financed separately. Therefore, there is no financial stimulation to share information as it does not benefit the discipline itself. Alas, patient safety is subordinate in some cases to information sharing, e.g. in the case of Nationaal Schakelpunt, a nation wide switching relay to access medical records between GPs and medical specialists. Most of the debate revolved around who was the owner of the medical data as holding the information was related to financing [30].

The other reason why there is little to no incentive to share information is the workload and time pressure that is present in the emergency healthcare, sometimes there is no time to even write down

information and patient information is shared orally during treatments of transfer. For instance, the burden on ambulance care has increased with introduction of the GPP since they increasingly rely on ambulance care outside offices hours [16]. Therefore time is scare for ambulance staff to share information properly as they barely have time to write information down and trauma assessments forms are extensive forms to fill in.

Finally, the treatment evaluation could not be fully completed yet as the evaluation from the stakeholders in the field is lacking. The field test is still ongoing and only the first informal response is received that users find the software easy to use, complete and fast. However, this is only an initial result and not a valid statistical result to perform a treatment validation on. As such, the treatment evaluation is incomplete and evaluates only the methodologies used and

7.3. Future work

Now that the technical foundation for a communication infrastructure has been created, field tests on a bigger scale and in different use cases can expand on the current field test with RescueTrack. Use cases can comprehend a wide variety of disciplines, e.g. the desired integration with the GPs, that can have the most intermediate effect as patients medical records are provided to emergency physicians [8]. As one of the identified communication problems regarded the missing medical records, integration with GPs could be first area to expand in. More studies can be done in the same area to integrate the Nationaal Schakelpunt, the switching relay that can access medical records on demand, but patient data remains in the GPs own systems. As such, it avoids the vital question of data ownership, as who owns the data is a hot debate in the medical world [30].

With faster communication available the collaboration in pre-hospital care (the provided treatments before going or arriving at the emergency department) can be intensified, which can improve the quality and safety of hospital care. The more information is already known (e.g. medical records, or hospital availability), the more can be anticipated on adequate patient care. More treatments can be provided before or during the trip to the hospital, thus relieving the burden on hospitals. Future work can expand on this topic.

A future study that is already mentioned (section 6.5) is the intervention study planned by Radboud UMC to measure the effect of an accurate ETA on an emergency department's team and the potential optimization of the emergency physicians' workload distribution. In this study the impact of a real-time ETA on an emergency department team will be measured, because at the moment other activities are abandoned in favor of incoming patients, who arrive often later than estimated. The only roughly available ETA estimation removes therefore patient care, where an accurate estimation instead can improve the workflow of emergency physicians [22].

Another future study that is going to take place is the integration of emergency departments' bed availability. Right now, ambulance staff just use their experience to choose a hospital to deliver their patient to, and call whether there are beds available. Ambulances can show up at emergency departments who have a patient stop, or pass a nearby hospital because the bed availability is unknown. An external party has created an portal that have the hospitals availability presented, and integration with that party can resolve that problem by showing an emergency department's availability in the trip form.

References

[1] Kjeld Harald Aij. "Ketendenken". In: *Wie vraagt wordt beter!* Springer, 2017, pp. 74–80.

[2] Dirk Alkema. ""O, komt u voor mij?"". In: *Vakblad van Ambulancezorg* 4 (2018), pp. 25–27.

[3] Johan Groen Anneke Goossen-Baremans. *Meetbare kwaliteit van zorg in Nederland*. Aug. 2016. URL: https://www.hl7.nl/phocadownload/Whitepapers/Whitepaper%20-%20Meetbare%20kwaliteit%20van%20zorg%20in%20Nederland%20v1.6.pdf (visited on 04/24/2019).

[4] IEEE Standards Association. *IEEE 42010-2011 - ISO/IEC/IEEE Systems and software engineering - Architecture description*. Oct. 2011. URL: https://standards.ieee.org/standard/42010-2011.html (visited on 04/25/2019).

[5] George W Beeler. "HL7 Version 3 - An object-oriented methodology for collaborative standards development". In: *International Journal of Medical Informatics* 48.1-3 (1998), pp. 151–161.

[6] Duane Bender and Kamran Sartipi. "HL7 FHIR: An Agile and RESTful approach to healthcare information exchange". In: *Proceedings of the 26th IEEE International Symposium on Computer-Based Medical Systems*. IEEE. 2013, pp. 326–331.

[7] Rajkumar Buyya, Chee Shin Yeo, and Srikumar Venugopal. "Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities". In: *2008 10th IEEE International Conference on High Performance Computing and Communications*. Ieee. 2008, pp. 5–13.

[8] Roeland Drijver. "Continuiteit in de acute zorg". In: *Huisarts en wetenschap* 49.11 (2006), pp. 810–811.

[9] Luca Filipponi et al. "Smart city: An event driven architecture for monitoring public spaces with heterogeneous sensors". In: *2010 Fourth International Conference on Sensor Technologies and Applications*. IEEE. 2010, pp. 281–286.

[10] Menno MI Gakeer et al. "Landelijke ontwikkelingen Nederlandse SEHs: aantallen en herkomst van patienten in de periode 2012-2015". In: *Nederlands Tijdschrift Voor Geneeskunde* 160 (2016).

[11] Haryadi S Gunawi et al. "What bugs live in the cloud? a study of 3000+ issues in cloud systems". In: *Proceedings of the ACM Symposium on Cloud Computing*. ACM. 2014, pp. 1–14.

[12] Dick Hardt. "The OAuth 2.0 authorization framework". In: (2012).

[13] HL7-affiliation. *Betere gegevensuitwisseling in de spoedzorgketen*. Feb. 2019. URL: https://www.informatieberaadzorg.nl/binaries/informatieberaad-zorg/documenten/publicaties/2019/2/15/presentatie-consultatiesessie-ambulancezorg-nederland-woensdag-6-februari-2019/190206_01_Presentatie_+Ambulancezorg.pdf (visited on 07/02/2019).

[14] Gregor Hohpe and Bobby Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[15] TFM Hooghiemstra. "Transmurale ict in de zorg: stapsgewijze invoering is nabij". In: *Zorg en Financiering* 4.3 (2005), pp. 13–29.

[16] T Jansen et al. "Tussen ambulance en huisarts: ontwikkeling in de spoedeisende ambulancezorg en het draagvlak voor de verpleegkundig specialist acute zorg in Zuid-Holland Zuid." In: (2016).

[17] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems.* " O'Reilly Media, Inc.", 2017.

[18] Gert Koelewijn. *Acute Zorg*. Apr. 2019. URL: https://www.nictiz.nl/standaardisatie/informatiestandaarden/acute-zorg/ (visited on 03/06/2019).

[19] Jay Kreps. *Getting Real About Distributed System Reliability*. Mar. 2012. URL: blog.empathy%EF%BF%BD%EF%BF%BD%EF%BF%BDbox.com (visited on 17/05/2019).

[20] Microsoft. *Azure Integration Services*. 2018. URL: https://azure.microsoft.com/mediahandler/files/resourcefiles/azure-integration-services/Azure-Integration-Services-Whitepaper-v1-0.pdf (visited on 05/14/2019).

[21] Alireza Moghaddam. "Coding issues in grounded theory". In: *Issues in educational research* 16.1 (2006), pp. 52–66.

[22] Michael M Neeki et al. "Accuracy of perceived estimated travel time by EMS to a trauma center in San Bernardino County, California". In: *Western journal of emergency medicine* 17.4 (2016), p. 418.

[23] NHG/Nictiz. *Richtlijn gegevensuitwisseling huisarts huisartsenpost ambulancedienst afdeling spoedeisende hulp*. May 2014. URL: https://www.nhg.org/sites/default/files/content/nhg_org/uploads/richtlijn_acute_zorg_2014_versie_3.0.pdf (visited on 05/01/2019).

[24] Nictiz. *Acute Zorg*. Apr. 2019. URL: https://www.nictiz.nl/standaardisatie/informatiestandaarden/acute-zorg/ (visited on 03/06/2019).

[25] Nictiz. *Basisdataset Terugrapportage meldkamer, ambulance en spoedeisende hulp naar huisarts*. June 2012. URL: https://www.nictiz.nl/wp-content/uploads/2012/06/AORTA_dHA_BDS_Tl_DataSet_terugrap.pdf (visited on 06/05/2019).

[26] Nictiz. *Snel vooruit in de acute zorg*. Mar. 2018. URL: https://www.nictiz.nl/wp-content/uploads/2018/03/Infographic-acute-zorg.pdf (visited on 09/16/2019).

[27] Nictiz. *Snel vooruit in de acute zorg*. Apr. 2019. URL: https://www.nictiz.nl/standaarden/acute-zorg/ (visited on 09/17/2019).

[28] Johan Oosterwold. "Spoedeisende ambulancezorg en ouderen". In: *Vakblad van Ambulancezorg* 4 (2018), pp. 32–35.

[29] Duco Roolvink. "Keuzes maken als iedere seconde telt". In: *Skipr* 5.10 (2012), pp. 28–30.

[30] Diana van Roon. "Schakelpunt tussen lokale zorgsystemen". In: *Tijdschrift voor praktijkondersteuning* 12.5 (2017), pp. 26–29.

[31] Nick Rozanski and Eoin Woods. "Software systems architecture: working with stakeholders using viewpoints and perspectives". In: Addison-Wesley, 2011.

[32] DMJ Schalk et al. "Professioneel Handelen in de Spoedzorg: Ontwikkeling en implementatie van richtlijnen en protocollen". In: *Triage* 2009.3 (2009), pp. 17–20.

[33] Tariq Rahim Soomro and Abrar Hasnain Awan. "Challenges and future of enterprise application integration". In: *International Journal of Computer Applications* 42.7 (2012), pp. 42–45.

[34] Karthikeyan Umapathy and Sandeep Purao. "Designing enterprise solutions with web services and integration patterns". In: *2006 IEEE International Conference on Services Computing (SCC'06)*. IEEE. 2006, pp. 111–118.

[35] Alix Vargas et al. "Towards the development of the framework for inter sensing enterprise architecture". In: *Journal of Intelligent Manufacturing* 27.1 (2016), pp. 55–72.

[36] Wouter Verhoef. “Beter te laat dan te vroeg arriveren op de SEH!?” In: *Vakblad van Ambulancezorg* 3 (2019), pp. 13–17.

[37] Matt Welsh et al. “The staged event-driven architecture for highly-concurrent server applications”. In: *University of California, Berkeley* (2000).

[38] Roel J Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[39] Tomasz Wiktor Wlodarczyk, Chunming Rong, and Kari Anne Haaland Thorsen. “Industrial cloud: Toward inter-enterprise integration”. In: *IEEE International Conference on Cloud Computing*. Springer. 2009, pp. 460–471.

[40] Ding Yuan et al. “Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems”. In: *11th Symposium on Operating Systems Design and Implementation (14)*. 2014, pp. 249–265.

8. Appendix

8.1. Technology stack AmbuSuite

1. Storage
   (a) Azure SQL DB
   (b) Redis
   (c) Azure Tables
   (d) Azure Shares (images)
2. Applications
   (a) Service Fabric (legacy connectors)
   (b) Azure Functions
   (c) Azure Webapps (applications)
   (d) Azure Service Bus
   (e) Azure Active Directory
   (f) Azure Key Vault
3. Communication
   (a) Service bus (pub/sub, queues)
   (b) REST APIs
   (c) Websockets
   (d) UDP
4. Monitoring
   (a) Application Insights
   (b) PagerDuty
5. Collaboration
   (a) Atlassian (Jira, confluence)
   (b) Slack
6. Support
   (a) Zendesk
   (b) DevOps (ci/cd)

8.2. Inductive open coding

List of articles that have been used in the inductive open coding process:

1. Dirk Alkema. O, komt u voor mij?. In: Vakblad van Ambulancezorg 4 (2018), pp. 2527 [2]
2. TFM Hooghiemstra. Transmurale ict in de zorg: stapsgewijze invoering is nabij. In: Zorg en Financiering 4.3 (2005), pp. 1329 [15]
3. NHG/Nictiz. Richtlijn gegevensuitwisseling huisarts huisartsenpost ambulancedienst afdeling spoedeisende hulp. May 2014 [23]
4. Nictiz. Snel vooruit in de acute zorg. Mar. 2018 [26]
5. Johan Oosterwold. Spoedeisende ambulancezorg en ouderen. In: Vakblad van Ambulancezorg 4 (2018), pp. 3235 [28]
6. Duco Roolvink. Keuzes maken als iedere seconde telt. In: Skipr 5.10 (2012), pp. 2830 [29]
7. Wouter Verhoef. Beter te laat dan te vroeg arriveren op de SEH!? In: Vakblad van Ambulancezorg 3 (2019), pp. 1317 [36]

The table with the codes found in the inductive coding process can be found below (Table 2). All of the codes are subjected to a concept found in the research question: what architectural design can facilitate an optimal information delivery in the Dutch multi-agency emergency healthcare. The research question revolves around three concepts: architectural design, information delivery, and emergency

RQ Concept	Related codes
Software architecture	Communication infrastructure Inadequate/out-dated IT-solutions Healthcare staff support System integration Lack of uniformity
Healthcare	Patient journey ETA Bed capacity Patient care (quality) Costs/financing Isolated disciplines Emergency department Ambulance Treatments/assessments Medical history(Difficulty with) work(flow) Scattered decision chain
Information delivery	Communication (problems) Insufficient information Overhead on communication Late information delivery Oral/written communication Unsupervised/uncontrolled information flows

Table 2: Codes

healthcare. The codes are the ones after deduplication (axial coding). Note that the concept of inductive open coding means that upon repetition, the exact codes can differ, depending on the researcher.

8.3. Perspective traceability

Table 3 details the perspectives' architectural tactics and whether they are full-filled in the design or not [31]. These architectural tactics are provided to help applying the perspectives on the viewpoints, however, not all tactics might be applicable, depending on the design.

Perspective: Security	**Full-filled**	**Not applicable**
Apply recognized security principles	Yes	
Authenticate the principals	Yes	
Authorize access	Partially	
Ensure information secrecy	Yes	
Ensure information integrity	No	
Ensure accountability		Outside scope
Protect availability	Yes	
Integrate security technologies	Yes	
Provide security administration	Partially	
Use third-party security infrastructure	Yes	
Perspective: Performance & Scalability		
Optimize repeated processing	Yes	
Reduce contention via replication	Yes	
Prioritize processing	Yes	
Consolidate the workload	Yes	
Distribute processing over time		24/7 servicing
Minimize use of shared resources	Yes	
Reuse resources and results	Yes	
Partition and parallelize	Yes	
Scale up or scale out	Yes	
Degrade gracefully	Yes	
Use asynchronous processing	Yes	
Relax transactional consistency		24/7 real-time requirement
Make design compromises	No	
Perspective: Availability & Resilience		
Select fault-tolerant hardware		Cloud computing
Use high-availability clusting		
Yes		
Log transactions	Yes	
Apply software availability solutions	Yes	
Create fault-tolerant software	No	
Design for failure	Yes	
Allow for component replication	Yes	
Relax transactional consistancy		24/7 real-time requirement
Identify backup and recovery solutions	Yes	
Perspective: Evolution		
Contain change	Yes	
Create extensible interfaces	Yes	
Apply design techniques for change	Yes	
Apply meta-model architectural styles	Yes	
Build variation points into software	Yes	
Use standard extension points	Yes	
Achieve reliable change	Yes	
Preserve development environments	Yes	

Table 3: Perspective traceability

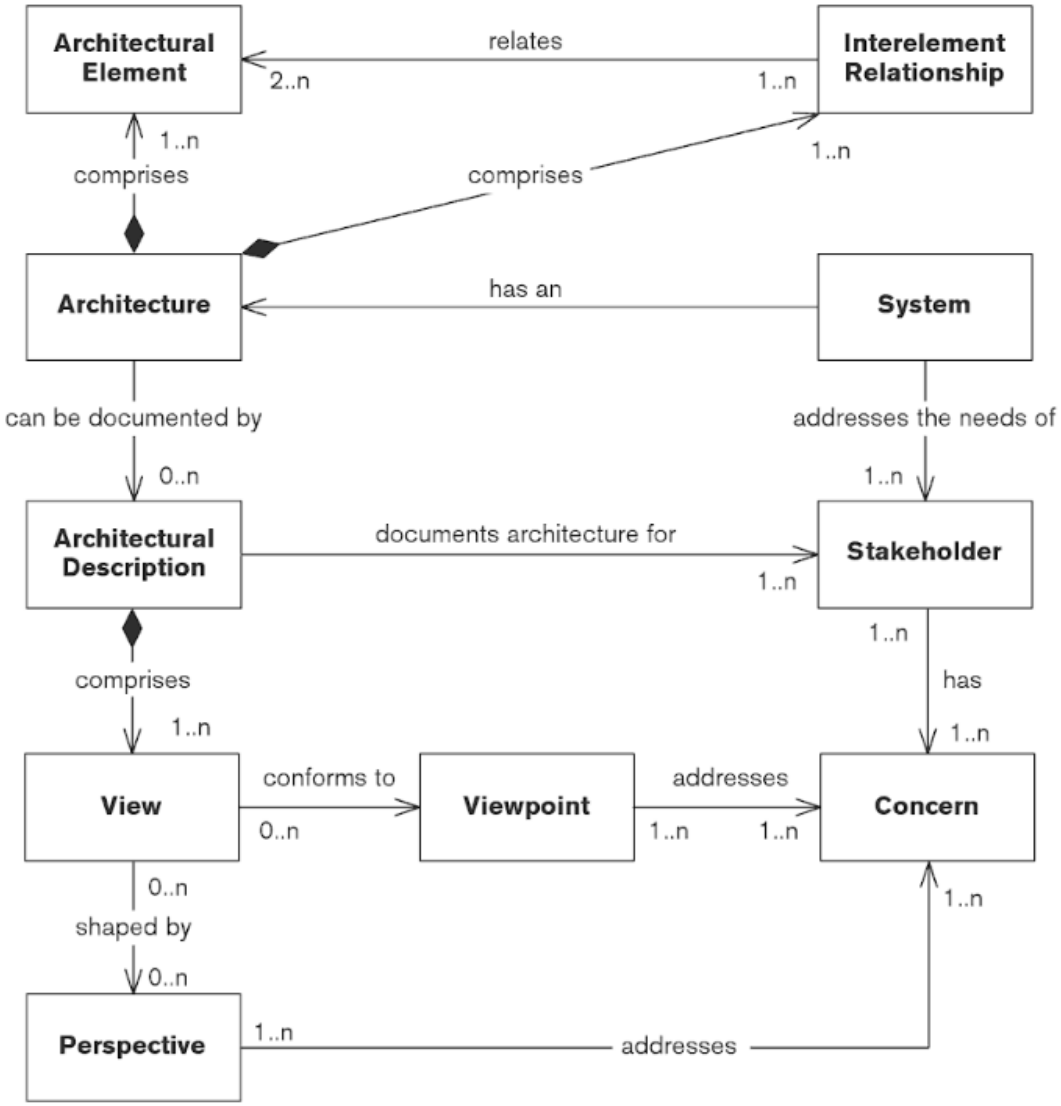

Fig. 11: Architecture core concept relations, derived from [31].

8.4. APIM Policy

The policy consists of six elements that enable authentication and the forwarding of requests to the back-end. Within the <inbound> section, at first the <set-backend> sets the back-end web service URL if not done previously. Second, <authentication-managed-identity> requests an access token for the specified resource. The policy uses a Named Value to avoid hard-coded identifiers or credentials. Third, <set-header> is used to remove the Ocp-Apim-Subscription-Key for the back-end. Fourth <set-header> is used to create an authorization header including the access token obtained earlier. Fifth, the <set-header> is used to set the content type to JSON. Sixth, in the <backend> section, a <forward-request> is used to forward the request to the back-end.

```
<policies>
    <inbound>
        <base />
        <set-backend-service id="apim-generated-policy" backend-id="ApiApp_ambu-flow-api-
            ↪ dev" />
        <authentication-managed-identity resource="{{resourceFlowID}}" ignore-error="false
            ↪ " output-token-variable-name="accessToken" />
        <!-- Remove the subscription key from the header -->
        <set-header name="Ocp-Apim-Subscription-Key" exists-action="delete" />
        <set-header name="Authorization" exists-action="override">
            <value>@("Bearer " + context.Variables["accessToken"])</value>
        </set-header>
       <set-header name="Content-Type" exists-action="override">
            <value>application/json</value>
        </set-header>
    </inbound>
    <backend>
        <forward-request timeout="60" buffer-request-body="true" />
    </backend>
    <outbound>
        <base />
    </outbound>
    <on-error>
        <base />
    </on-error>
</policies>
```

8.5. API technical functions

The GetAll[] functions from each data entity are used when it is desired to retrieve all the information available in a data entity without having to call each separate method for all the information, e.g. in case of a database write operation. A [] indicates that function parameters can be passed into the API-methods.

Trip: GetTripByID, GetAllTripInfo[], GetUserMissions[], UpdateMission, GetMissionUpdates[], GetMissionRequests, ApproveMission, UpdateTrafficCongestions[], GetETA, GetStatusCode, GetStatusTime, GetEndOfTrip[], GetUrgency, GetPreAnnouncement[].

Incident: GetIncidentByID, GetAllIncidentInfo[], GetRetrievalAddress[], GetDeliveryAddress[], GetEmergencyCallInfo[].

Ambulance: GetAmbulanceByID, GetAllAmbulanceInfo[], GetPosition, GetInstitutionAvailability.

Patient: GetPatientsByID, GetAllPatientsInfo[], GetPatientPersonal[], GetInsurance[], GetMedication, GetSituation[], GetPrimarySurvey[], GetSecondarySurvey[], GetTreatment[], GetTriage[].