Unified messaging control platform

PDEng THESIS REPORT

Unified Messaging Control Platform

Priyanka Patel

October 2020

Department of Mathematics & Computer Science

PDEng SOFTWARE TECHNOLOGY

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Unified Messaging Control Platform

October 2020

Eindhoven University of Technology
Stan Ackermans Institute – Software Technology

PDEng Report: 2020/073

Confidentiality Status: Public

Partners	Philips	Eindhoven University of Technology
Steering Group	Priyanka Patel Marcel Quist (Philips) Tanir Ozcelebi (TU/e) Zoran Stankovic (Philips)	
Date	October 2020	

Contact Address	Eindhoven University of Technology Department of Mathematics and Computer Science MF 5.080B, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands +31 402472759
Partnership	This project was supported by Eindhoven University of Technology and Philips.
Published by	Eindhoven University of Technology Stan Ackermans Institute
PDEng-report	2020/073
Preferred reference	Unified Messaging Control Platform. Eindhoven University of Technology, PDEng Report, 2020/073, October 2020
Abstract	In order to improve the existing and traditional communication workflows between Philips and its customers, demands a need for a consolidated platform enabling communication by means of real-time and asynchronous channels. This report summarizes design of a multimedia messaging platform which easily integrates real-time features with web and backend applications and can be deployed on cloud or on-premise. A cloud-based system was developed and some of the high-level functionalities are real-time multimedia group messaging, presence, optimized data synchronization, and offline messaging. Communication via different channels such as SMS, Email, and Web Push are also supported. The platform is extendable which makes it easy to integrate additional features and third-party services. A selected few AI services are integrated as prototypes to demonstrate its potential for further extension. This messaging platform was part of a pilot application and was tested at three hospital sites with selected customers.
Keywords	Messaging, WebSocket, Real-time, Publish-subscribe, GraphQL, AWS, Philips, TU/e, Software Technology, PDEng
Disclaimer Endorsement	Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the Eindhoven University of Technology or CLIENT. The views and opinions of authors expressed herein do not necessarily state or reflect those of the Eindhoven University of Technology or CLIENT, and shall not be used for advertising or product endorsement purposes.
Disclaimer Liability	While every effort will be made to ensure that the information contained within this report is accurate and up to date, Eindhoven University of Technology makes no warranty, representation or undertaking whether expressed or implied, nor does it assume any legal liability, whether direct or indirect, or responsibility for the accuracy, completeness, or usefulness of any information.
Trademarks	Product and company names mentioned herein may be trademarks and/or service marks of their respective owners. We use these names without any particular endorsement or with the intent to infringe the copyright of the respective owners.
Copyright	Copyright © 2020. Eindhoven University of Technology. All rights reserved. No part of the material protected by this copyright notice may be reproduced, modified, or redistributed in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the Eindhoven University of Technology and CLIENT.

Foreword

It is such a privilege that we have been able to witness the growth path of the PDEng education from multiple angles. First of all, being a PDEng student myself in the past, I learned to appreciate the multi-disciplinary technology design approaches. Secondly, with the PDEng SW Technology international team we worked closely on a fun AI demonstrator assignment with 18 fellow international PDEng students and our team, and that was where we first met. And now, over the last 10 months, we witnessed your personal growth from closeby within our team. A warm and positive experience!

Where the first team assignment was about exploration for a fun and inspiring demonstrator, showing what you can do with already available AI (Artificial Intelligence) in the cloud, your final assignment has been a major step in technical complexity, while adapting to the stakeholders' expectation levels. You were able to do that in a Philips research settings where we seek and design health solutions based on customer insights and needs. Where the requirements are typically in flux and never carved in stone. You actually supported and drove the process to get these requirements clear and focused by rapid prototyping, inspiring by tangible examples and – most of all nowadays – empowering co-creation to unleash the talents of many others. This is in high level terms our envisioned assignment.

Now more specifically on the topic, Priyanka turned a preliminary idea of a new interfacing concept into a comprehensive new set of multi-media platform services. Interfacing between human chat conversations and multiple AI-to-knowledge-sources. Why multiple? Because we envisioned that solutions will quickly involve multiple parties, each having their own set of AI-sources. And from a user perspective, another important aspect of such messaging platform is that solutions will quickly need to become indifferent to the commonly available media channels such as email, sms, WhatsApp, WeChat, etc. including application embedded proprietary chat channels. Most of these channels exist and are provided by cloud vendors. Priyanka's platform approach bundles all, such that application designers and developers are free to select most appropriate channels suiting their use case (instead of being locked in by the available choices of a selected vendor). Plus, for future sustainability, the platform encompasses a vendor abstraction API, such that same application designers can easily switch to other vendor's services if these become more innovative or economically attractive. Another prevention of lock in.

In a report that is published in 2020, at least one reference to COVID-19 must appear somewhere, and where better than in this foreword? The team circumstances changed considerably in your 3rd month of the assignment (March 2020) when the offices basically closed down and working from home became the new norm. Thank you for the quick and easy adaption to this new reality – in keeping up the team spirit alongside impressive contributions to the 'Scalable Service Delivery' team and Philips.
As said, I consider it a privilege knowing you personally and being in the position to 'add your name' to the annals of our joint working history in Philips!

Thank you!

Zoran Stankovic and Marcel Quist
25th September 2020

Preface

This report summarizes the “Unified Messaging Control Platform” project carried out by the author as the graduation project of the Professional Doctorate in Engineering (PDEng) program in Software Technology. This program is a two-year technological designer program offered by Eindhoven University of Technology, Stan Ackermans Institute. The project duration was ten months and was conducted at the Professional Healthcare Services and Solutions (PHSS) department within Philips Research, Eindhoven.

The goal of the project is to design and implement a messaging platform that supports real-time integration capabilities with web applications and backend applications and that can be implemented on-premise or on cloud. This report describes the successful realization of the project and elaborates the software development and project management processes. This document is constructed such that the reader is given an overview of the problem and the high-level vision and is led to the solution through research, design, implementation, and validation at pilot sites.

Audience for this report can be both technical as well as non-technical readers. Readers who are interested to know the context and high-level goals of the project can refer Chapters 1, 2, and 3. Readers willing to learn about the domain aspects of messaging and latest services can read Chapters 4 and 5. Chapters 6 and 7 are intended for technical readers who wish to learn about the design of the system, the functionalities, and the implementation. Readers mainly interested in the results of the project can read Chapters 9 and 10. For getting a general idea of the project along with requirements, and the project management process, refer Chapters 1, 2, 3, 11, and 12. Readers who are interested in knowing about all aspects of the project are welcome to read the entire report.

October 2020

Acknowledgements

I would like to thank and express my gratitude to everybody who supported, collaborated, and assisted me over the course of the project.

At Philips, I thank my project supervisor and project manager, Marcel Quist, for giving me the opportunity to be a part of his team and this project. Your introduction towards how a leading technology company like Philips functions and how contributions from individual departments and team add to the bigger picture helped me in understanding the vision. Your inputs towards keeping helicopter view for this project and insights regarding other interesting work happening around has always been helpful. Your support and patience provided me with an encouraging environment where I enjoyed the freedom to explore and learn.

I am thankful to Zoran Stankovic, my project supervisor and mentor at Philips for providing me with the guidance, knowledge, and expertise that I needed during the project without which I could not have achieved the final results and deliverables. You were always there for any technical questions that enabled me to get acquainted to the domain and it accelerated my learning process. You have always encouraged me to learn more and excel. I would also thank you for believing in me that the work done during the project could be used for the pilot.

At TU/e, I thank my project supervisor, Tanir Ozcelebi, for his advice and constant encouragement during the project meetings. Your guidance and confidence in me assured me to feel comfortable in the process and the way in which I was working and progressing.

I also thank the PDEng, Software Technology management, Yanja Dajsuren and Desiree van Oorschot for their constant encouragement and organizational support during the entire PDEng program.

Additionally, my gratitude goes to the entire team of Scalable Service Delivery, including Arjan Draisma, and Robin Mennens. It was nice to learn from each other during the standups and demo sessions and was fun to know more about each other during the team building activities.

I also extend my gratitude to Danny Schaefer and Thasneem Moorkan from the Service Connect team for the pleasant collaboration.

I thank my fellow PDEng colleagues for the engagement and collaboration in the workshops and module projects during the first year of the program. I would like to specially mention my PDEng colleague, Robin Mennens, for being the companion during the graduation project at Philips.

Special thanks to all my friends for their never-ending support and encouragement. A special mention to my friend Akarsh Sinha for always encouraging and guiding me through the process.

I thank my parents for their love and prayers and for always being there and I am grateful to them for everything they have done for me. And most of all, I thank my husband, Sabyasachi Neogi, for his constant affection, encouragement, and understanding. You supported me every step of the way, that enabled me in being the best version of myself, both personally and professionally.

Priyanka Patel
September 2020

Executive Summary

Philips is a leading health technology company with the vision of improving people's lives and well-being through meaningful innovations for healthy living, diagnosis, treatment and home care. Philips' products and services need to be serviced to ensure top quality and performance to customers and there is always a need to continuously innovate in service delivery methods to ensure scalable delivery at all times and circumstances.

This project was conceptualized in the context of digitization of remote customer support Philips provides to its medical equipment customers, which primarily are hospitals. The goal is to address the known issues of existing phone-based system such as long waiting and resolution times, frequently transferred customer calls, and internal fragmented tools and processes. The company's vision is to revamp this experience by having a unified platform which comprises of high-quality connected services to track and monitor cases and to deliver an effortless service experience via an omni-channel digital platform.

One of the integral building blocks of the digital support experience is multimedia based messaging, through various modes such as real-time channels and asynchronous channels. Though the idea started off for improving the customer support workflow, there are other contexts experiencing similar issues and could potentially reap the benefits of such a digital platform. Remote communication between patients and caregivers facilitating a tighter bond with families, or other signaling based applications are such instances.

The objective of this project is to design and develop a multimedia-based messaging platform that supports messaging across different channels so that it acts as a comprehensive platform for different systems to easily incorporate messaging functionalities.

This report describes the project, technical design and processes followed in order to realize the project goals. The result of the project is a cloud implementation of the messaging platform which can be tweaked for an on-premise implementation. With the platform, the following functionalities are delivered.

- Real-time and core messaging features which are multimedia group messaging, optimized data synchronization, real-time presence updates, and offline messaging.
- Messaging via asynchronous channels which are SMS, Email, and Web Push.
- Low latency real-time behavior for both front end and backend applications.
- Prototypes demonstrating integrations with a selected set of AI algorithms and services on the multimedia data.

The cloud messaging platform exposes different types of APIs to integrate any client web application. For backend client applications, user defined callback APIs are also supported through which the messaging platform transmits real-time events. The platform also provides APIs in order for client applications to run AI algorithms and services to their multimedia data. The platform is highly configurable and extendable, which makes it easy to add additional features and third-party services. The implemented platform also exhibits non-functional properties, such as modularity and interoperability.

The messaging platform developed as part of this project is tested and verified in a pilot execution with three hospitals in North America. An application named Service Connect was launched by Philips Research in order to demonstrate digitization of the remote customer support experience and the communication was facilitated by the messaging platform.

Table of Contents

List of Figures

List of Tables

1.Introduction

This chapter introduces the company and explains the context in which the work was carried out during this project. The chapter describes the context of remote customer support experience in detail as it helps in understanding the foundation of this project. The problems being faced by Philips customers and their root causes are discussed. It also elaborates on the company's vision in improving the current experience and underlying processes. A brief description to other use cases and some limitations are discussed. Finally, the project objectives are discussed and the outline for the rest of the report is explained in the closing section.

1.1 Context

The "Unified Messaging Control Platform" project is conducted by the author, as part of her Professional Doctorate in Engineering (PDEng) program. The PDEng program in Software Technology is provided by the Department of Mathematics and Computer Science at Eindhoven University of Technology in the context of the 4TU.School for Technological Design, Stan Ackermans Institute [1].

A Professional Doctorate in Engineering is a full-time, two-year technological designer program and falls within the 3rd cycle of higher education and is an advanced training consisting of two parts. The first 14 months of the program includes courses to gain extensive knowledge and experience of the latest design methods and their applications and also includes three industry driven training projects. The last 10 months is working on an individual design project at a company.

This project is the individual design project of the author and was initiated by Philips Research, Eindhoven. Philips is a global health technology company which focuses on improving people's lives and enables better outcomes across different aspects such as healthy living, diagnosis, treatment, and home care. Philips delivers integrated products and solutions by state-of-the-art technologies and deeply value customer insights. The company is headquartered in the Netherlands, and is a leader in diagnostic imaging, image-guided therapy, patient monitoring, health informatics, consumer health, and home care. Philips believes in the power of innovation and strives towards making the world healthier and more sustainable. The company goal is to improve the lives of three billion people a year by 2030 and deliver superior value for customers and shareholders [2]. The next section gives a brief introduction to Philips Research and the department in which the project was carried out.

1.2 Philips Research

Philips Research introduces meaningful innovations for customers in order to improve people's lives and keep customers at the center of the process. It has a global presence including both developed and emerging markets. They work from spotting ideas and trends, to developing advanced proof-of-concepts and developing novel technologies and products. [3].

This project was conducted at Professional Health Services and Solutions (PHSS) department in Philips which is a digital proposition research department that works towards creating data-driven innovative services and solutions that enable healthcare providers to deliver improved outcomes at lower cost with improved patient and employee satisfaction. Using patient and care provider data, as well as data from both medical devices and public information sources, this group does big data analytics, process improvement, and patient preference techniques to improve care delivery systems via innovations delivered through information systems and consulting across the health continuum.

This project was completed with the Scalable Service Delivery (SSD) team, which is part of the PHSS department, and the project contributes towards the team's vision. The team focuses on the development of innovative services and solutions to digitize the communication experience by exploring remote communication capabilities. They also enable healthcare providers to deliver better results with improved patient and employee satisfaction. Philips wants to grow in services but currently relies on conventional delivery methods. The team adds scalable and enriching service delivery capabilities based on adaptive intelligence, which enhance efficiency for routine services and facilitate growth in new, added value services whilst improving customer response times and satisfaction.

1.3 Remote Customer Support

The existing customer support workflow that Philips provides to its medical equipment customers, which primarily are hospitals, is an extensive and complex process to ensure coverage of many cross-functional aspects like customer experience, resolution efficiency, risk management, and regulations [4]. The support is a remote phone-based communication between hospital staff members and Philips customer service personnel and if required, a service engineer is sent on site for support. The personnel involved in direct interactions are called front stage personnel and there are backstage personnel who work in the background towards resolution. The frontstage people are the BioMed Technician (BMT), a Customer Care Centre (CCC) agent, a Remote Service Engineer (RSE), and a Field Service Engineer (FSE).

1.3.1. Workflow

The end-to-end support workflow [5] can be broken down into stages and is explained below.

1) **Issue Detection** – Hospital staff detects an issue and gets a ticket created in the hospital system. The BioMed technician at the hospital is informed about the new case and the affected system.
2) **Issue Assessment** – The BMT checks if the issue can be resolved locally by first calling the personnel at the hospital ticketing system to understand the problem and then investigates similar old cases and system manuals.
3) **Issue Reporting** – If the BMT decides that the problem cannot be resolved locally, the Philips CCC agent is contacted. The CCC agent is the first level of contact for the BMT and gives the system details and answers safety questions. This information is logged in the Philips support system and a case is created. Then the Service Level Agreements are verified and an RSE is notified.
4) **Remote Assessment** – Once the RSE receives the ticket, a first level of analysis is done by the RSE and after that either an FSE is assigned to visit the customer on site or the spare parts delivery process is started. The hospital BMT is called to discuss the assessment and next steps. If spare parts need to be ordered, the case is transferred from the RSE to the CCC and the quotation process is started, and the hospital is intimated.
5) **Quotation Process** – The CCC receives the quotation from the dedicated team and forwards it to the customer. The customer forwards the quotation to their finance team and waits on approval of the quotation.
6) **Ordering of spare parts** – Once the customer approves the purchase order, the CCC orders the parts and waits for delivery information. The details about the spare parts are shared with the customers.
7) **Assigning FSE** – An FSE can be assigned directly after remote assessment or after the ordering of spare parts to assist the customer on-site with installation. The availability of FSEs is checked and an FSE is assigned to visit the customer site. The customer is informed about the FSE visit.
8) **On Site Visit** – The FSE visits the site, assesses the issue, and installs the spare parts. If another visit or more assistance is required, the FSE contacts the RSE and CCC and work towards a plan.
9) **Reporting** – After the customer confirms that the problem is fixed, the ticket is updated, closed, and a follow up with customers is done for feedback.

The above description provides a simplified version of the workflow and has its associated shares of problems faced by both customers and support personnel. The next section briefly discusses some of the recurring issues.

1.3.2. Customer Issues

The front stage personnel at the hospital side are the BioMed, nurse and other staff members who directly interact with Philips support and the front stage personnel at the Philips support side are the CCC, FSE, and RSE agents. Table 1 lists the problems faced by hospital staff members and potential causes from the perspective of support agents [5].

Table 1 – Remote Support – Customer issues and potential causes

Issues: Hospital Staff (Biomed, Nurse)	Causes: Support Agents (CCC, RSE, FSE)
Long waiting times	Large volume of cases and analyzing one case at a time could lead to agent's unavailability.
Calls getting transferred to multiple agents and having to repeat information	Not having the right skillset to support a case could lead to rerouting to other agents

No way to share screen or share media	Only a phone-based support, which currently does not support sharing media (The ability to share media could help understand the problem better and could lead to faster resolutions and possible reductions in FSE visits)
Not a transparent process	Many agents and background tasks are involved without having a control center to unify all the individual processes
No clear update or visibility on the case status	Some information could be lost between different process and different tools for logging
Experience varies with different agents	Agents have loosely defined protocols to follow and hence approaches could be based on their experience
Long resolution times	Challenges in finding agents availability with the required expertise, dependency on third party tools, no prioritization in the ticketing system.

1.3.3. Vision

The *Scalable Service Delivery* team envisions to address the customer issues faced in the present phone-based support and take this experience to the next level by delivering high quality connected services, to provide an effortless service experience via an omni-channel digital support journey, and to have one platform to track and monitor cases. Figure 1 illustrates a high-level view on the enhanced digital support experience, which simplifies the process of creating cases, answering safety questions, routing to concerned individuals, sending and receiving messages on preferred channels, attaching media files, and always having a visibility on the case status [6]. This experience could be further enhanced by adding video conferencing and self-help capabilities.

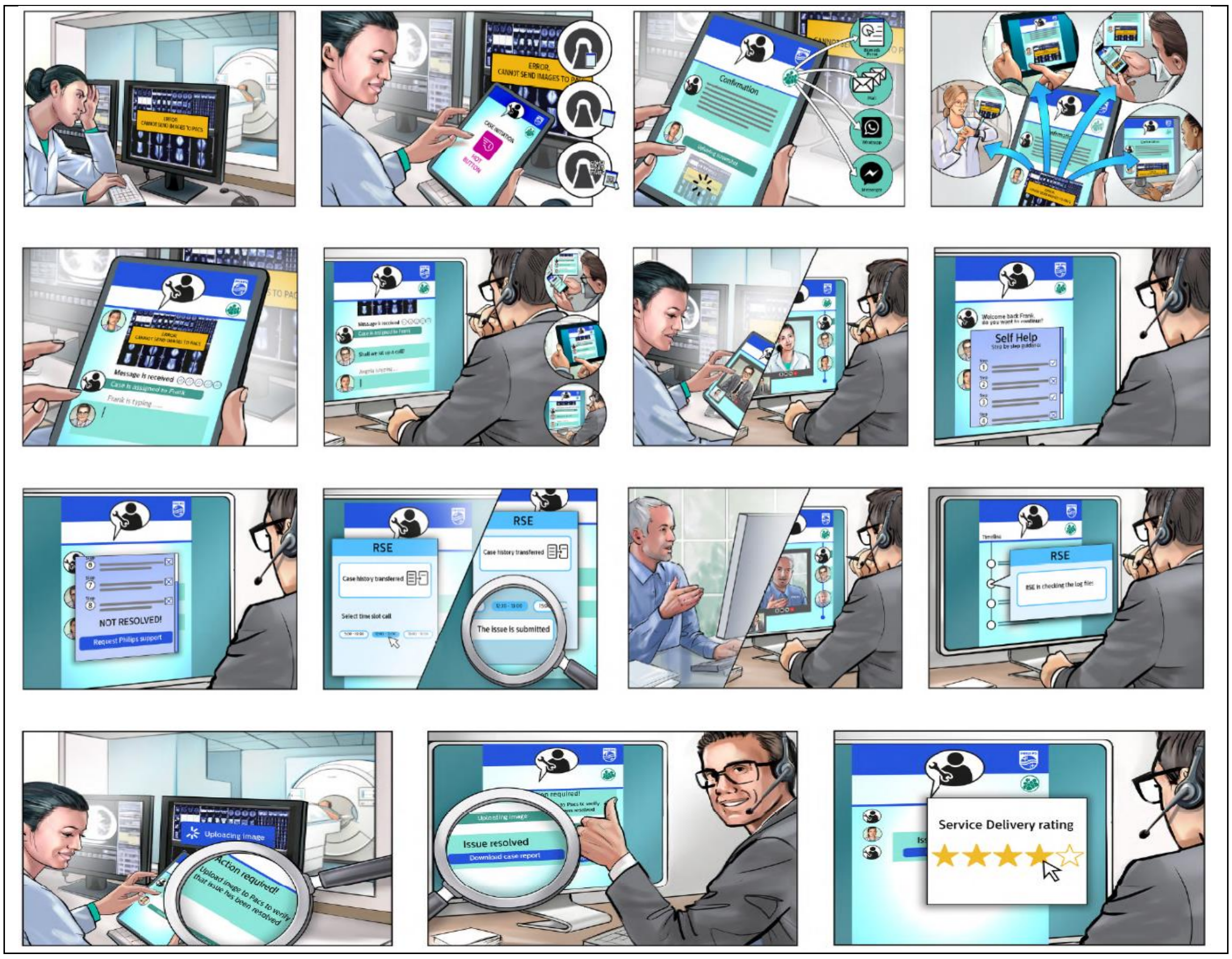


Figure 1 – Envisioned customer support experience mockup [6]

1.4 Additional Use Cases

Section 1.3 explains the remote customer support use case and the vision for the enhanced digital experience. Though this project was initiated in the customer support context, it is not only directed towards this use case.

This project also explores additional use cases and contexts experiencing similar problems and their specific vision towards a potential resolution. Two such use cases are described below.

Telehealth – One of the stakeholders, which is a software department within Philips introduced this use case. Telehealth is access and management of health care services remotely and are an alternate to the in-person hospital visits. While this is very efficient and works well for most of the cases, there are patients involved in long-term care with hospitals, such as pregnancy or chronic diseases. For simple and trivial queries, the patients may not want to go through the general procedure. The vision is to have a simple text and multimedia-based communication system which groups selected individuals where both patients and care givers can enquire or respond to questions as per their convenience. Sharing of multimedia files and gaining insights from defined algorithms could augment the experience.

WebRTC Signaling – WebRTC is a technology for peer-to-peer real-time audio and video communication, which establishes media exchange between devices without an intermediate server to facilitate this communication. There is a need to establish a connection between devices before they can communicate with each other. This process of device discovery and the negotiation process is known as signaling. Signaling involves connecting over an agreed server through which the devices can identify each other and then communicate. The SSD team owns a WebRTC platform and uses signaling service of a third-party provider. The vision here is to have an in-house signaling mechanism which would eliminate the dependency on third party providers and is a potential use case for the project.

1.5 Project Objectives

The goal of the project is to develop a unified messaging platform that support the vision of digitization of remote communication capabilities derived from different use cases. The objective of the project is to design a messaging platform supporting communication via low latency real-time channels and asynchronous channels. The platform should be developed at an appropriate abstraction so that different applications can use this platform for their specific needs. The platform should be easily integrable with both browser applications and backend applications in order for any system to add messaging functionalities.

The platform should facilitate messaging via asynchronous channels such as SMS, Email, and social media channels. Since third-party services would be used to integrate these channels, the platform should not expose any direct dependency in its interfaces to these services, so that they can be changed without impacting end users and client applications. The final objective is to enrich the platform and develop a few prototypes to demonstrate integration with AI services and have extension points in order for additional services to be integrated.

1.6 Outline

The subsequent chapters in this report are divided as follows.
Chapter 2 introduces the stakeholders of the project, their interests, and how stakeholders are managed.
Chapter 3 describes the low-level functional requirements and the non-functional requirements.
Chapter 4 discusses few protocols and methodologies which were assessed and reasonings behind the selected choice and gives direction towards the solution.
Chapter 5 briefly describes messaging vendors and services and are compared.
Chapter 6 contains the architecture and design of the project and explains all the system components and the way in which individual functionalities are realized.
Chapter 7 explains the resources are used in order to implement the cloud platform,
Chapter 8 explains the deployment process and the pilot context in which the platform was tested.
Chapter 9 explains the process of validation and verification.
Chapter 10 discusses the conclusions and future work.
Chapter 11 explains the project management process.
Finally, Chapter 12 reflects upon the project from the author's perspective. ■

2.Stakeholder Analysis

The first chapter gives an overview of the context and objectives of the project. This chapter discusses stakeholder analysis that was conducted during the initial phases of the project, and the engagement process that was followed during the project lifespan. The analysis and engagement steps included: 1) Identification of all direct and indirect stakeholders and listing their interests and inputs, 2) Assessment of each stakeholder for their interest and influence in the project 3) Stakeholder prioritization for high interest and influencing parties, and 4) Development of stakeholder communications plans and engaging with stakeholders.

2.1 Stakeholders and Concerns

The stakeholders are categorized as direct and indirect stakeholders. Direct stakeholders are active people or entities involved in the project having a visible role. Indirect stakeholders are entities who are interested in the ultimate results and are not involved frequently and with whom the interactions are limited.

The *Scalable Service Delivery* team of Philips Research owns this project. The team guides the design, technology choices, and implementation. TU/e has a stake through the PDEng trainee who is managing and conducting the project. The project is dependent on service providers and their products, which are used for design, implementation, and deployment purposes. This project was tested in a customer support pilot and the teams associated with the pilot are directly involved and are also the direct stakeholders. Table 2 lists the project direct stakeholders, their role, the concerns they are trying to address with the project, and the inputs they provide for project execution.

Table 2 – List of direct stakeholders, their interests and provided inputs

Marcel Quist, Role: Company Supervisor – Project Manager	
Interest	• Ensure that the project adds value to the company • Innovate solutions for scalable service delivery
Inputs	• Providing project context, business values, and introducing use cases • Monitoring and providing feedback on the progress • Reviewing PDEng thesis and documentation

Zoran Stankovic, Role: Company Supervisor – Lead Architect	
Interest	• Assess different methods and technologies and choosing appropriate stacks • A solution that fits different use cases and contexts • An extendable design for the messaging platform
Inputs	• Providing domain knowledge, technical inputs, and feedback • Monitoring the progress, design and development process • Reviewing PDEng thesis and documentation

Tanir Ozcelebi, Role: TU/e Supervisor	
Interest	• Proper and successful completion of graduation project • Trainee working towards interesting solutions
Inputs	• Monitoring quality, process, and progress of the project and report • Supporting by providing relevant information

Yanja Dajsuren, Role: PDEng ST Program Director	
Interest	• Proper and successful completion of graduation project • Have future engagements between TU/e and company
Inputs	• Providing TU/e related guidelines • Helping in removing roadblocks

Priyanka Patel, Role: PDEng Trainee	
Interest	• Gain research experience in software design • Complete the project successfully on time
Responsibility	• Managing and executing the project • Prioritizing requirements and delivering results • Completing PDEng graduation report and other relevant documentation

Service Providers, e.g. Twilio, AWS. Role: Customer support of service providers	
Interest	Client engagement with their services
Inputs	Supports by helping in integrations and working on feature requests

Service Connect Team, Role: The teams developing other pilot components which interface with the messaging system, and together deliver a unified experience to customers	
Interest	Pluggable and simple to use interfaces
Inputs	Feedback on integration

Table 3 lists some of the indirect stakeholders which are end users or potential users of the system. Interactions with them were either very limited or their concerns were represented and addressed by a direct stakeholder. These stakeholders have relatively low interest and influence.

Table 3 – List of indirect stakeholders

Stakeholder	Role	Interests / Inputs
Vital Health	Software department within Philips	Introduces healthcare use-cases to the project
Chief Architect Office	Architects group which develops and scales innovation research POCs at company level	Use the outcomes of the project for additional use cases. Enhance and release to a wider audience
Pilot hospitals	Service Connect application end users at hospitals sites	Provide iterative feedback on the digital remote service experience with every release.

2.2 Stakeholder Prioritization

A stakeholder map representing the interest and influence levels of the stakeholders is prepared in order to prioritize stakeholders. Figure 2 shows the stakeholder map, the *Influence* axis depicts the of power the stakeholder has on the project and the *Interest* axis depicts their level of interest in the project. The stakeholders placed in different segments of the matrix were handled in slightly different ways which are explained below.

- High influence – High Interest
  They are the key project stakeholders and needs to be managed closely with regular interactions.

- High influence – Low Interest
  Due to high influence, their impact needs to be assessed and managed over time.

- Low influence – High Interest
  Keep these stakeholders informed about the progress as they have interesting insights.

- Low influence – Low Interest
  Informing these stakeholders from time to time about the project status is sufficient.

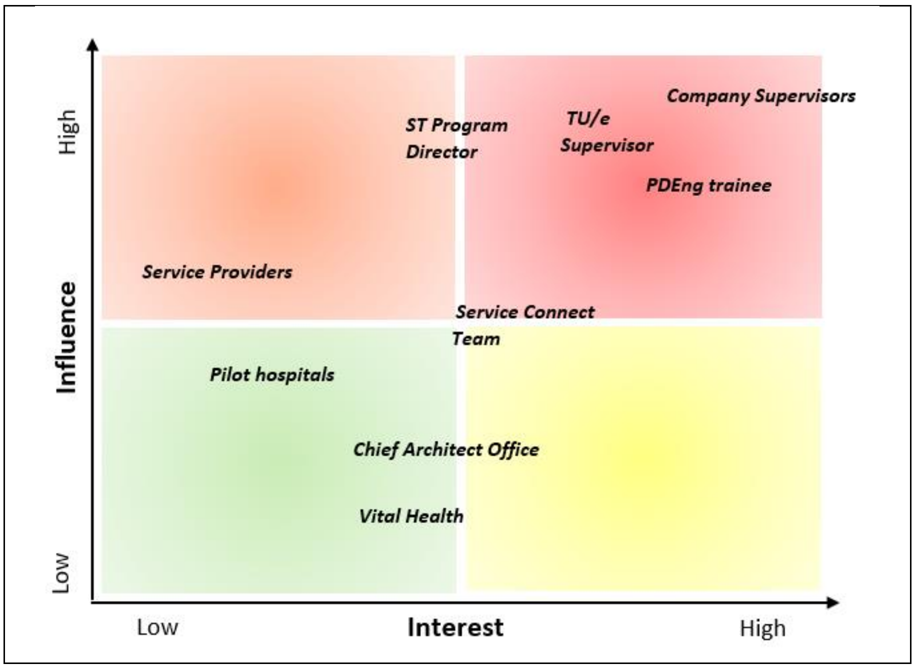


Figure 2 – Stakeholder map

2.3 Stakeholder Communications Plan

After stakeholder prioritization, a communication plan was created in order to manage communication with stakeholders that focused on the ones with either high interest or high influence, or both. The communication plan outlines the stakeholders, the frequency of interactions, and the way in which the interactions would be carried out. Table 4 lists the stakeholder communications plan. The plan was agreed with the stakeholders and was executed during the project.

Table 4 – Stakeholder communications plan

Stakeholder	Mode of Communication	Frequency
Philips Supervisors	In-person, emails, remote meetings, sprint demos	Informal – Twice a week Formal – Every two weeks PSG – Every month
TU/e supervisor	In-person, remote meetings, emails	Status updates – Every two weeks PSG – Every month
ST Program Director	Email, meetings	On demand
Service Providers	Email, phone support	On demand
Service Connect Team	Remote meetings, sprint stand-ups, sprint demos	Twice a week

■

3.System Requirements

The previous two chapters introduces the context and stakeholders of the project. After the analysis of the problem and discussion with the concerned stakeholders, the high-level requirements or features were extracted. After in-depth discussions, the low-level functional requirements were formulated. This chapter describes the priority model that was used to weigh the requirements. Generic system requirements, system functional requirements, and system non-functional requirements are listed. Requirements associated with a specific feature are grouped into categories.

3.1 Introduction

The application of the messaging platform in various contexts were discussed and are described in Chapter 1. From the high-level use cases, the features were extracted and then features were broken down into requirements. The MoSCoW model [6] is used to prioritize the requirements. Table 5 summarizes the MoSCoW model.

Table 5 – MoSCoW model

M	Must Have	Mandatory system requirements
S	Should Have	Important requirements that are not mandatory but adds significant value
C	Could Have	Nice to have requirements that will have less impact if they are not fulfilled
W	Will not Have	Not essential in the project time frame

The system requirements are divided into the three categories, general requirements, functional requirements and non-functional requirements and are described in the following section.

3.2 General System Requirements

These requirements capture certain generic attributes and behaviors of the system and are listed in Table 6.

Table 6 – General system requirements

Req Id	Description	Priority
GR-001	The system shall be integrable with at least the following web browsers and operating systems: • Google Chrome and Safari for iOS • Google Chrome and Mozilla Firefox for Android • IE9, Microsoft Edge, Google Chrome, Mozilla Firefox, and Safari for Windows, macOS, and Linux	**M**
GR-002	The system shall be integrable with any front-end or back-end application irrespective of the application technology stack.	**S**
GR-003	The client shall not require downloading of additional apps to use the system.	**M**
GR-004	The interface between the system and the client applications shall be generic enough to fit multiple use cases and be easy to use.	**S**
GR-005	The system shall be deployable on a cloud platform.	**M**
GR-006	The system shall be deployable on premise or a hybrid platform.	**S**
GR-007	The system shall support sending messages in all languages with different character sets.	**M**
GR-008	The system shall support interactive messaging where users exchange messages in real-time to engage in conversations.	**M**

3.3 Functional Requirements

The functional requirements define and describe the behavior of the system. They are further broken down into categories, where each category groups requirements associated with a certain feature. Table 7 lists these categories and their associated functional requirements.

Table 7 – Functional requirements

Registration – Enables independent applications to be registered to the system

Req Id	Description	Priority
FR-001	The system shall allow registration of new applications so that attributes are identified per application, and specific customizations can be performed.	**M**
FR-002	The system shall uniquely identify each registered application and its associated configurations.	**M**
FR-003	The system shall allow adding identities, groups, and messages only after successful registration with the system.	**M**

Identity Management – Enables new users to be added to the system

Req Id	Description	Priority
FR-004	The system shall allow adding new identities to a specific application.	**M**
FR-005	The system shall uniquely identify identities across an application.	**M**
FR-006	The system shall allow building an identity profile (e.g., name, contact details, and any other application specific data) for each identity.	**M**
FR-007	The system shall allow deletion of identities.	**M**
FR-008	The system shall be able to distinguish identities by roles (e.g., admin role, user role).	**S**
FR-009	The client application shall be able to retrieve the list of currently registered identities.	**M**

Group Management – Enables new groups to be added to the system and each group is a set of identities.

Req Id	Description	Priority
FR-010	The system shall allow adding new groups to a specific application.	**M**
FR-011	The system shall uniquely identify groups across an application.	**M**
FR-012	The system shall have the ability to add and remove identities to a group.	**M**
FR-013	The system shall have the ability to close and re-open a group.	**M**
FR-014	The system shall allow deletion of groups.	**M**
FR-015	The system shall allow building and updating of a group profile (e.g., name, description, state, and other application specific data).	**M**
FR-016	The system shall allow adding administrative permissions for certain group operations.	**S**
FR-017	The system shall inform other users in a group when group details are modified (e.g., identity being added or removed from a group).	**S**
FR-018	An identity can be part of more than one group at a time.	**M**
FR-019	A group can be configured to be short-lived.	**C**
FR-020	A group can be configured to live indefinitely.	**C**

Sending messages – Enables sending text messages to peers and groups

Req Id	Description	Priority
FR-021	The system shall allow sending peer-to-peer messages between identities associated with the same application.	**M**
FR-022	The system shall allow sending messages from an identity to a group.	**M**
FR-023	When messages are sent, the system shall deliver messages to connected recipients.	**M**
FR-024	The system shall allow users to send messages to offline members who are currently not connected to the system.	**S**
FR-025	The system shall allow sending messages to an open group.	**S**
FR-026	The system shall not allow sending messages to a closed group.	**S**
FR-027	The system shall allow at least 256KB of text data to be sent in one operation.	**M**

Multimedia Support – Allows sending multimedia messages

Req Id	Description	Priority
FR-028	The system shall allow sending images, videos, audio recordings, and documents.	**M**
FR-029	The system shall support at least these image formats: jpg, png, gif, bmp.	**M**
FR-030	The system shall support at least these video formats: avi, mp4, flv, mov.	**M**
FR-031	The system shall support at least these document formats: doc, xls, ppt, txt, pdf.	**M**
FR-032	The system shall retain all the multimedia files.	**M**
FR-033	The system shall allow deletion of multimedia files by the original sender.	**S**
FR-034	The system shall identify each multimedia file uniquely.	**M**
FR-035	The system shall support at least 500MB upload of a single multimedia file.	**M**
FR-036	Multimedia files shall be available at the original size and resolution.	**S**

Receiving messages **–** Allows receiving of messages by identities

Req Id	Description	Priority
FR-037	The system shall allow receiving peer-to-peer messages between identities associated to the same application.	**M**
FR-038	The system shall allow an identity to receive group messages.	**M**
FR-039	An identity shall be able to retrieve entire conversation history from the system.	**M**
FR-040	A connected user shall receive messages in real-time with minimal latency.	**M**
FR-041	The stored offline messages shall be delivered to identities when they re-connect to the system.	**M**
FR-042	The system shall allow filtering of messages based on groups and time intervals.	**S**

Conversation History – Allows storing conversations

Req Id	Description	Priority
FR-043	The system shall store conversation history.	**M**
FR-044	The system shall allow restoring both peer-to-peer and group conversations.	**M**
FR-045	The system shall allow a configurable maximum storage time for conversations after which they shall be removed from the system.	**C**
FR-046	The system shall allow storing conversations indefinitely.	**M**
FR-047	The system shall allow deletion of messages by sender and admins.	**S**
FR-048	The system shall allow message filtering by admins.	**C**

Delivery and Synchronization – Enables successful message delivery and data synchronization

Req Id	Description	Priority
FR-049	The system shall deliver messages in real-time to connected recipients.	**M**
FR-050	The system shall maintain message sequencing to preserve the order of a conversation.	**M**
FR-051	When connected, the system shall not deliver a specific message multiple times to a recipient.	**M**
FR-052	The user shall be notified when a message is successfully acknowledged by the system.	**S**
FR-053	The system shall never fail to deliver messages to connected users.	**M**
FR-054	The system shall allow a way to optimize data synchronization.	

Offline Messaging – Enables messaging in an offline mode

Req Id	Description	Priority
FR-055	The system shall provide a seamless experience while sending and receiving messages in offline mode.	**S**
FR-056	The system shall allow sending of text messages in offline mode.	**S**
FR-057	The system shall allow sending of multimedia messages in offline mode.	**C**
FR-058	For offline users, the system shall deliver messages at a later point in time when the user is online.	**S**

Presence – Allows knowing the connected status of group members

Req Id	Description	Priority
FR-059	The system shall allow to know the online and offline status of group members.	**M**

FR-060	The system shall allow defining custom statuses, for example, *Busy, Away, DND.*	**S**
FR-061	The system shall capture the last connected time of individual identities.	**M**
FR-062	The system shall inform group members about the real-time presence changes of other group members.	**M**
FR-063	The system shall capture the read status of messages.	**M**
FR-064	The system shall capture the typing status of group members.	**M**
FR-065	The system shall inform group members about the typing status of other group members in real time.	**M**

Asynchronous channel integrations – Allows sending messaging via other channels

Req Id	**Description**	**Priority**
FR-066	The system shall allow sending SMS messages.	**S**
FR-067	The system shall allow sending Email messages.	**S**
FR-068	The system shall allow sending WhatsApp messages.	**S**
FR-069	The system shall allow sending Web Push notifications.	**S**

Third-party integrations – Allows integrating external services

Req Id	**Description**	**Priority**
FR-070	The system shall be extensible in order to add external bots.	**C**
FR-071	The system shall be extensible in order to add virtual assistant systems.	**C**
FR-072	The system shall be extensible to add text-based AI services	**C**
FR-073	The system shall be extensible to add multimedia-based AI services.	**C**

3.4 Non-functional Requirements

Non-Functional Requirements elaborate system characteristics and impose quality constraints that the system shall satisfy. ISO standard 25010 [7], which is a quality model, is used to evaluate the non-functional requirements. This quality model is used to determine which characteristics shall be considered when evaluating the quality attributes of the system. Table 8 lists the non-functional system requirements.

Table 8 – Non-functional requirements

Req Id	**Aspect**	**Description**
NFR-001	Performance efficiency – Time Behavior	The system shall send and receive messages with minimal latency.
NFR-002	Performance efficiency – Resource utilization	System shall be lightweight in handling multimedia messages.
NFR-003	Compatibility – Interoperability	System shall be usable across all user devices and platforms.
NFR-004	Compatibility – Interoperability	The system shall allow messaging across different types of network connection and shall handle network changes in a seamless manner.
NFR-005	Usability	The system interfaces shall be simple and easy to use by client applications.
NFR-006	Security	The system shall encrypt the data during network transmission.
NFR-007	Security	The system shall authenticate and authorize user operations.
NFR-008	Maintainability – Modularity	The system shall be implemented in a modular way so that parts of the system can be developed and maintained independently.
NFR-009	Maintainability – Modifiability	The system shall be designed in such a way that it is easy to add new components or services to the system.

■

4.Domain Analysis

The previous chapter lists the low-level functional requirements and the non-functional requirements of the system. This chapter explores and assesses various architectural models, methodologies, networking protocols, and standards which are relevant in order to realize the system. The objective of this chapter is to broaden the understanding of the domain by analyzing the above-mentioned elements, their advantages and limitations, and identifying the appropriate options that resonates closely with the project requirements. The client-server architectural model, messaging paradigms, messaging protocols, and web API standards are discussed. It also describes ways in which low latency real-time behavior can be achieved.

4.1 Client Server Model

The client-server model is an architecture style where the resource providers are the servers and resource consumers are the clients. The clients and servers communicate over a network connection using a network protocol. The World Wide Web and HTTP have become the universal communication platform and communication protocol for web-based applications.
The peer-to-peer model is in contrast with the client-server model where a central server is not required, and each peer can act both as a client and as a server depending on whether it responds to requests or initiates requests. Described below are two models which are based on the client-server architecture.

4.1.1. Client Pull

In this model, the client first establishes connection to a server to send requests and the server accepts these connections in order to serve these requests and sends back responses, also commonly known as the request-response model. The server cannot initiate connections nor can send asynchronous events to clients. The two entities can keep the request-response exchanges until one of them drops out. This model has a lot of heavy lifting done by the infrastructure to avoid burdening the client and server with delivery issues. A request-response protocol operating on top of TCP/IP can be guaranteed that requests or responses will be delivered in order, at most once, and without corruption. If data needs to be protected in transit, the TLS protocol operating on top of TCP provides the necessary protection. The most known protocol based on the request-response pattern is Hypertext Transfer Protocol (HTTP) [8].

4.1.2. Server Push

This model enables a server to push events directly to clients without an explicit request from the client. The client subscribes to some sort of information from the server beforehand. As and when data is available on the server and if it matches the client's subscription, the server can directly push data to the client.

Selected Approach - Server Push
Real-time applications are more optimized where the server has the ability to push data to the clients without the clients requesting for data. Server push model is more efficient than client pull model as it reduces the server load for incoming network bandwidth, CPU power, and memory usage since it does not need to handle large number of incoming requests. This leads to timely delivery of data to clients as the latency of frequent opening and closing of connections is eliminated.

4.2 *Messaging Paradigms*

This section explains two paradigms which are widely used for messaging applications. The three main components are message sender, receiver, and a message broker, which facilitates communication between the sender and receiver.

4.2.1. Publish-Subscribe (Pub-Sub)

This is a form of asynchronous communication which allows messages to be broadcasted. The senders and receivers are decoupled and have no information of each other. The receivers subscribe to a particular topic and the senders publish messages to that topic which are then distributed to subscribers. An example of a simple publish-subscribe system is represented in Figure 3. The components are as follows.

1) Publisher and Input Channel – In this model, the message sender is also known as the publisher. The sender packs the messages and sends them via the input channel.
2) Subscriber and Output Channel – The message consumers are called subscribers that subscribe to a particular subject and can then receive messages published on that subject. Each consumer requires a separate output channel.
3) Message Broker – This component acts as an intermediary that is responsible for copying every message published on the input channel and distributing them to the output channels for all appropriate subscribers.

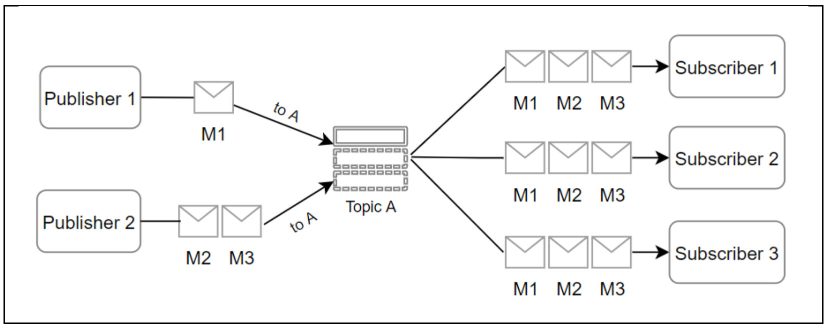


Figure 3 – Publish-subscribe messaging

Advantages:

- Decouples publishers and subscribers and therefore can be independently managed
- Improves scalability, reliability, and responsiveness of the system
- Easy integration between systems running on different protocols or environments

Considerations:

- Security of the channels, message ordering, and prioritization must be managed.
- As the channels are unidirectional, if an acknowledgement service is needed, it needs to be built separately.

4.2.2. Point-to-Point (P2P)

This is also a form of asynchronous messaging which uses queues to deliver messages. Queues must be defined before a transaction and the sender must know information about the receiver before it can send messages. A P2P model can have any number of senders and receivers, but each message can be consumed by only one receiver. An example of a simple P2P system is represented in Figure 4. P2P messaging components are as follows.

1) Sender – Creates and sends message to a queue and attaches receiver details
2) Message broker – Takes the message from the queue and delivers it to the appropriate receiver. It also deletes the message after it is acknowledged by the receiver
3) Receiver – Binds to a queue, consumes messages, and acknowledges message receipt to the broker.

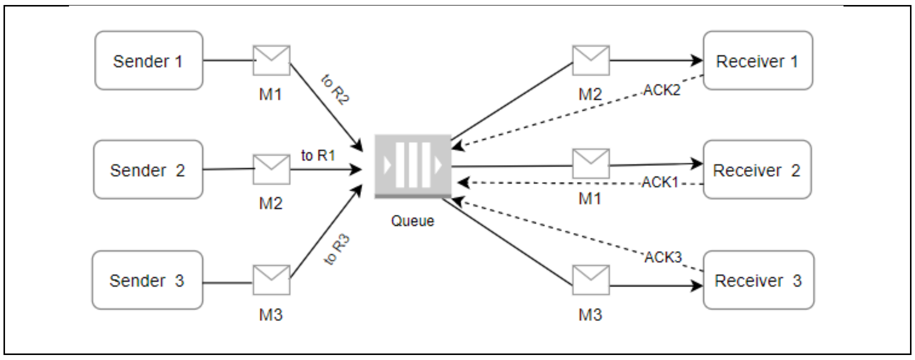


Figure 4 – Point-to-point messaging

Advantages:
- Message delivery is ensured, and message acknowledgement is known
- Messages are retained in the queue until they are delivered to the recipient

Considerations:
- Each message is sent to a specific queue and each message can be received and processed by a single receiver.
- High coupling between senders and receivers

Selected Approach - Pub/Sub

As per the system requirements, both peer-to-peer and group messaging shall be supported by the system. The ability of publish-subscribe systems to broadcast messages to multiple users in contrast to P2P messaging with only one recipient is an obvious choice. Also, the sender and receiver subsystems being decoupled and the ability to manage and scale them independently makes this approach more suitable for the project.

4.3 Real-time Messaging Methods

This section describes some of the widely used techniques to implement real-time behavior with minimal latency between clients and servers. Web based applications were traditionally designed as a request-response architecture. To realize near real-time behavior, the HTTP protocol [8] [9] and its request-response model have been continuously adapted and modified over the years.

4.3.1. Short Polling

In this traditional method, the client sends requests to the server repeatedly at a certain frequency and the server sends a response for every request, shown in Figure 6. A low polling frequency can result in less updates from the server, hence the information is not real-time for frequently changing data. Data in most applications is sporadic, hence not every request may contain new events or data from the server, though the polling cycle continues at the set frequency. This unnecessarily consumes high server and network resources and can over burden them.
Each polling cycle consists of three steps: a) setup and establishment of the TCP connection, b) the request-response cycle, and c) closure and cleanup of the TCP connection. Though this method is simple to implement, it has many disadvantages and remains one of the very traditional methods to achieve real-time behavior.

Drawbacks:
- Connection overhead is a resource consuming process
- Network latency, bandwidth, and polling frequency impacts real-time performance
- High incoming traffic leads to server and network overloading and wastage

4.3.2. Long Polling

Long polling is an optimization to polling where the server elects to keep the client's connection open for as long as possible thereby minimizing the latency in delivering data, shown in Figure 6. The server delivers response only after new information becomes available or a timeout threshold has been reached to prevent the client from being stuck indefinitely. After a response is returned, the open connection is closed, a new request connection is immediately sent, and the process continues. Challenges of this method are a) Connection overhead (though it is low compared to short polling) b) It is intensive for the server and network to maintain long lived connections as they consume resources. c) It is difficult to scale effectively d) Timeout issues and intermediate caching mechanisms can affect long polling.

4.3.3. HTTP Connection Types

The HTTP connections vary in terms of how they are opened, maintained, and closed. This can impact the performance of real-time applications. Described below are types of connections which are applicable for both short polling and long polling modes. Figure 5 represents these HTTP connection variations.

Short-lived connections

The original model, HTTP/1.0 is based on short-lived connections in which every request-response cycle is completed on its own connection. A TCP handshake is established before every request and connection is closed after each response and the requests are serialized. In HTTP/1.1, this connection type is used when *Connection: close* header is sent.

Persistent Connections
A persistent connection remains open for a certain time period and multiple requests can use the same connection, which saves the overhead of opening new TCP connections. With HTTP/1.1, an additional header *Keep-Alive* is used to specify a minimum time a connection should be kept open to ensure that the TCP connection is not dropped after each request-response cycle [9]. The drawback is that server resources are used even when connection is open.
Pipelining
An optimization of persistent connections where the client can make multiple requests without waiting for a response over the same kept-alive TCP connection reducing network latency. The disadvantage is that the HTTP protocol requires that the server return responses in the order in which they were received; therefore, a long running operation can block other requests from completing.

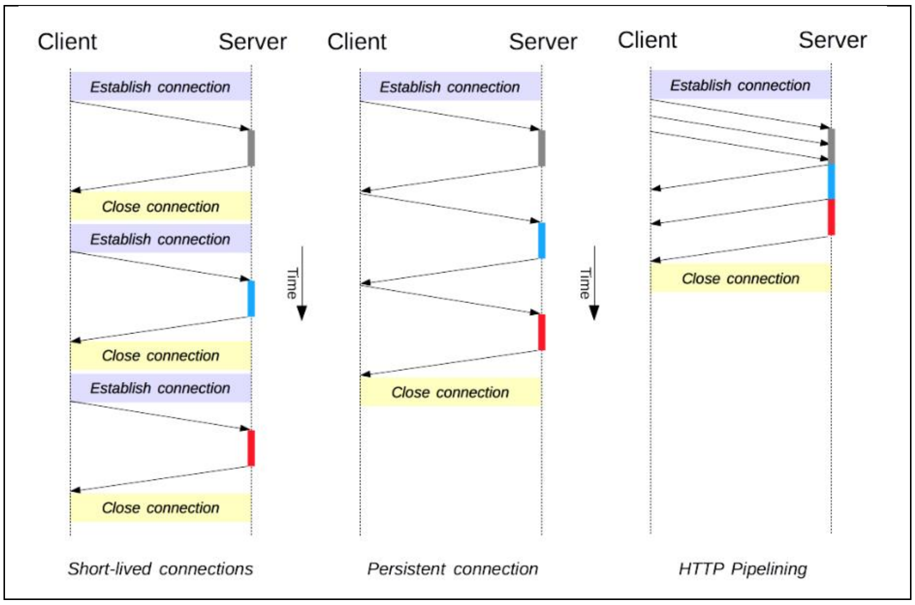


Figure 5 – HTTP connection types [10]

4.3.4. Server-Sent Events (SSE)

This is a mechanism that allows the server to asynchronously push data or events to the client once the client-server connection is established without having the clients to initiate requests. This is a one-way communication channel. The client initially subscribes to a stream from the server and the server sends event streams to the client as and when data becomes available. The connection remains open until one of the parties closes the stream. Disadvantages are a) SSE are not supported by all browsers b) One-way channel c) Binary data needs encoding. Figure 6 shows the working of polling, long polling, and SSE.

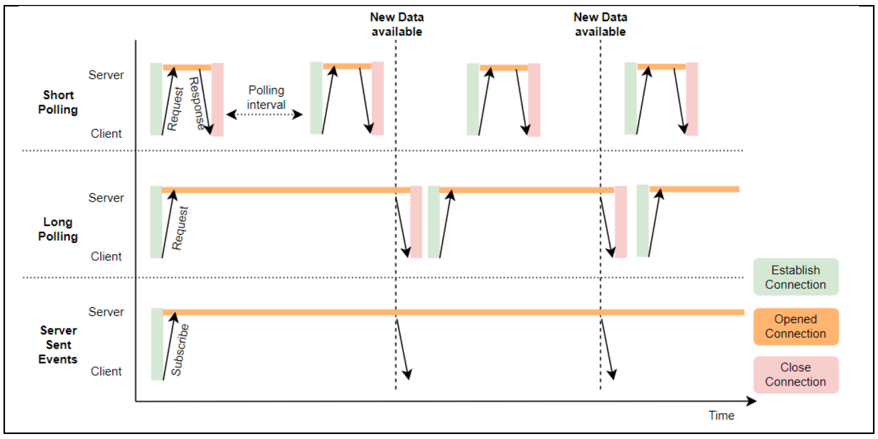


Figure 6 – Short polling, long polling, and server-sent events

4.3.5. HTTP Streaming

In this method, the server keeps the HTTP connection open indefinitely. It never closes the connection even after the data is pushed to the client. For streaming data, *Transfer-encoding: chunked* is sent along with the response which enables the server to send data in chunks over the same connection. The response is considered complete only when the server sends an EOF or either side explicitly closes the connection. This method reduces network latency as both the client and server eliminate the overhead of opening and closing connections. The drawback is that it is difficult to work with proxies and gateways, and also faces buffering issues for chunked data.

4.3.6. WebSocket

WebSocket creates bi-directional communication channel between the client and server where data can be passed back and forth without having to create new connections and requests. This data channel is set up by using two new HTTP headers, *Connection: Upgrade* and *Upgrade: websocket,* which updates the protocol from HTTP to WebSocket protocol [11]. The server accepts the setup request with an HTTP Response code *101 Switching Protocols* and reflects back the *Connection* and *Upgrade* headers. After a successful setup, application data can flow from either side over this channel. The interactions have minimal overhead and provides real-time data transfer from and to the server. The connection remains open until one of the sides closes the connection. The advantage is that it is compatible with the HTTP protocol and works over ports 80 and 443 (for SSL encryption) and supports HTTP proxies.

Selected Approach – WebSocket
For real-time applications with minimal latency, methods based on server push are more efficient than client-pull methods. Among the server-push methods explained in the Section 4.3 (HTTP Streaming, SSEs, and WebSocket), WebSocket remove the limitations of SSEs of one-way communication and encoding binary data. WebSocket also reduces the latency and processing inefficiencies of HTTP Streaming and the polling-based techniques [12]. WebSocket is also standardized across all major browsers and are supported by advanced open source client facing libraries. WebSocket protocol is explained in detail in Section 4.4.2.

4.4 Messaging Protocols

There are variety of communication protocols that can be used as the backbone for messaging. Below is a comparison of two such protocols, which are the top choices based on the project requirements. As one of the primary requirements is to have the messaging system run in a browser environment and also the ability to be used by any backend applications, many of the machine-to-machine protocols are not a good fit for the project.

4.4.1. Extensible Messaging and Presence Protocol (XMPP)

XMPP is an open source and extensible protocol [13] and can be applied effectively by using transport methods such as TCP/IP or HTTP. Based on a decentralized architecture, XMPP based communications assign a unique XMPP address that consists of an IP address, domain name, and username to all the users processing the communication channel.
The data is further bound with a secure transport layer such as TCP/IP or HTTP before it is exchanged between the users. In an XMPP based communication, there are three types of stanzas: a) Message stanza to exchange messages. b) Presence stanza to exchange online and subscription status and c) IQ (Info/Query) stanza to control dynamic settings of the communication that is controlled over the server.

Advantages:
- Decentralized architecture can help clients take control of their specific communication experience
- Robust security standards are built into the core specifications
- Quite extensible and flexible and hence many additional functionalities can be easily added

Disadvantages:
- Does not provide the ability to know the message-delivered status. They have to be configured manually.
- Uses XML for data transmission; hence does not support binary data transfers. Extensions must be used for binary data.
- The decentralized architecture allows anyone to run their own server, which can impact the system performance.

4.4.2. WebSocket Protocol

WebSocket is a communications protocol that provides full-duplex communication channels over a single TCP connection. This means both the client and server can simultaneously send data to each other without having to wait for a response or only send one way at a time. RFC 6455 [11] states that WebSocket "is designed to work over HTTP ports 80 and 443 as well as to support HTTP proxies and intermediaries" thus making it compatible with the HTTP protocol. To achieve compatibility, the WebSocket handshake uses the HTTP *Upgrade* header to change from the HTTP protocol to the WebSocket protocol [10]. Both HTTP and WebSockets are located at the application layer of the OSI model and depend on TCP at the transport layer.
WebSocket is standardized and is supported by all major browsers and there are libraries available for most of the high-level programing languages. Being able to support a full duplex connection, they are also termed as persistent connections.

WebSocket communication workflow:

- To establish a connection, a WebSocket handshake is performed through an HTTP request by the client
- If the server supports WebSocket, the server acknowledges the request by sending an identifier in the header.
- Updated URL replaces original HTTP connection with a WebSocket connection.
- With an active WebSocket connection in place, both the client and server can exchange data in real time.
- The data or messages are exchanged as frames, which can be one or many depending upon the content.

Advantages:

- With a centralized and persistent connection, it is one of the fastest online communication methods.
- There can be an unlimited number of user sessions on a single application.
- As a connection is active for long durations, server network traffic is reduced.
- Faster message delivery with negligible delay due of low network latency and the ability to send data from the server without an explicit client request.

Disadvantages:

- Although WSS is available, the technology still lacks security and is prone to certain attacks like.
- WebSocket connection masks and frames transmitted data, hence making it incompatible with a telnet client.
- Active WebSocket connections always keeps the ports open; hence there is no way to delay message delivery.

Table 9 – Comparison between XMPP and WebSocket

XMPP	WebSocket
De-centralized architecture	Centralized architecture with persistent connection between client and server
Slow compared to WebSockets	High speed data transfer capacity compared to XMPP
XMPP Core server + additional gateways for cross-origin users	Cross-origin built in architecture
Difficult to send binary data	Easy to send binary data
Inbuilt security with two layers of encryption	Though WSS adds a secure layer, it is not rich in security

4.5 Web API standards

In the last decade, REST (Representational State Transfer) has been the standard for designing web APIs. It offers features such as being stateless and provides structured access to resources. REST APIs are inflexible to keep up with rapidly changing client requirements. GraphQL was developed by Facebook in order to reduce the shortcomings of REST. It enhances flexibility and efficiency and provides a better experience for developers. Differences between REST and GraphQL are summarized in Table 10.

4.5.1. REST

REST (Representational State Transfer) is an API design architecture used to implement web services. REST-compliant web services allow the requesting systems to access and manipulate textual representations of web resources by using a uniform and predefined set of stateless operations. The resource methods for performing operations are *GET, POST, PUT,* and *DELETE.* All operations in REST are stateless and should allow caching at client side unless explicitly indicated otherwise.

4.5.2. GraphQL

GraphQL is a data query language and specification for APIs and also serves as a server-side runtime for executing queries by using a type system that encapsulates data [12]. GraphQL gives clients the flexibility to retrieve exactly the information they need instead of the server sending pre-defined fields and hence makes it easy to extend the APIs. The operations in GraphQL are *Query, Mutation,* and *Subscription*. GraphQL backend consists of a schema which is shared with client applications and contains API definitions. GraphQL resolvers extract data from underlying data sources and translate requests into data source specific operations.

Table 10 – Comparison between REST and GraphQL APIs

Criteria	REST	GraphQL
Network Requests	Requires multiple requests to fetch related resources	allows nesting of resources, hence reduces network requests.
Data fetching	Resources are available at different endpoints	there is only one endpoint at which data is accessed.
Operations	GET, POST, PATCH, PUT, DELETE	Query, Mutation, Subscription
Over or Under fetching	As the response in REST is decided by the server, it is probable that client pulls more data / less data than what is required	As GraphQL is a query language, the client decides what data it needs and only that data is fetched from the request.
Caching	Puts caching into effect as HTTP implements caching	has no caching mechanism and needs to be managed by client applications.
Error Handling	HTTP status codes make it easy to distinguish error responses and handle errors	GraphQL responses are always with a status code 200 with the error message; hence just the HTTP error codes are not enough to distinguish between responses
Versioning	They are usually versioned resources	No versioning is required as new types and fields can be easily added to the schema without impacting the existing schema.

■

5. Vendor and Service Evaluation

The previous chapter introduced the messaging domain and explains some of the protocols, methodologies, and standards. Messaging vendors offer end-to-end messaging capabilities which can be integrated with any system. This chapter compares different messaging vendors and the features they provide. The reasons for not using direct messaging vendors to realize the project are explained. Next, services at a lower abstraction are compared which are WebSocket, Publish-Subscribe, and GraphQL services. These services function as frameworks and only manages the connections and data sent across the connection.

5.1 Messaging Vendor Comparison

Messaging vendors are service provider companies that sell their end-to-end messaging solutions. After the requirements analysis, a number of messaging vendors were compared based on criteria extracted from the requirements. Four well known providers that offer end-to-end messaging services were compared. In addition to the core messaging functionalities, the vendors were also evaluated based on system extendibility, data ownership and security aspects. Table 11 compares features of WebSync [14], Twilio [15], PubNub [16], and Pusher [17]. The feature check was performed in the time frame of February and March 2020 and may be subject to changes as per their latest releases.

Table 11 – Comparison between messaging providers – WebSync, Twilio, PubNub, Pusher

	WebSync	**Twilio**	**PubNub**	**Pusher**
Service type	Pub/Sub service	Messaging service	Messaging service	Messaging service
Text messaging support	Yes	Yes	Yes	Yes
Identity management	Yes	Yes	Yes	Yes
Channel management	Yes	Yes	Yes	Yes
Multimedia support	Yes	Yes, 150MB limit	Yes, 5MB limit	Yes, 20MB limit
Data encryption	SSL	SSL and JWT Tokens	SSL	SSL
Storage of conversation history	No	Yes	Yes	Yes
Who has data ownership?	NA	Twilio	PubNub, but can be routed to own storage	Pusher
Presence support	Yes	Yes	Yes	Yes
Ensure message delivery	Yes	Yes	Yes	Yes
How can the service be hosted?	Any cloud or on-premise platform or WebSync cloud	Hosted on vendor servers, cannot be hosted by clients	Hosted on vendor servers, cannot be hosted by clients	Hosted on vendor servers, cannot be hosted by clients
Underlying technology	WebSocket	WebSocket	Not disclosed	WebSocket
Fallback technology	Long Polling	Not disclosed	Not disclosed	HTTP Streaming and Polling
Is the system extendable?	Yes	No	No	No
Scalability	Has to be managed	Managed	Managed	Managed

5.2 Vendor Evaluation

All the compared messaging vendors support the core messaging functionalities such as text messaging, presence, and message delivery guarantees and use the latest technology in their architecture to realize their system. The blocking factors were the following.

- Limitations on file size during multimedia transfers

- Services are not extendable to support additional custom features
- Not having control on the data as they are managed by vendors without end-to-end encryption guaranteed
- Not being able to host the service on our own cloud or servers

The conclusion of the vendor evaluation process was to not use messaging solutions from vendors as many of the non-functional requirements were being violated. The next step was to evaluate services that provides only the platform and infrastructure by using inbuilt real-time methodologies and protocols.

5.3 Service Evaluation

After the vendor evaluation, the focus of evaluation shifted towards exploring services at a lower abstraction level, that is, services which offers only the platform to facilitate real-time communication. The features of messaging system based on the functional requirements will be developed on top of the platform. Leveraging platform services enables to host and manage the system backend, have control on the data layer and security, and can easily extend the system to accommodate new features. The following describes three kinds of such services.

WebSocket services – These are the services that enables and manages real-time bi-directional communication between clients and servers. The services are generally offered at a low level of abstraction where the connection level interfaces are exposed and can directly be used by client applications. For examples, Socket.IO, AWS WebSocket API Gateway.

Publish-Subscribe services – These services facilitate subscribing to data, publishing data, and receiving real-time data. They are at a higher-level of abstraction compared to WebSocket as publish-subscribe utilizes an underlying connection level protocol for communication, which can be the WebSocket protocol. For example, Pusher Channel, Google PubSub.

GraphQL services – These services enable to develop GraphQL APIs. They are at the highest abstraction level compared to the previous two as GraphQL subscriptions are based on the publish-subscribe paradigm. GraphQL queries and mutations use HTTPS protocol while subscriptions use the WebSocket protocol for real-time communication. For example, Apollo GraphQL, AWS AppSync.

The conclusion after evaluation of these services was to use GraphQL services in order to develop APIs. GraphQL services also abstracts away the complexity of publish-subscribe messaging broker. WebSocket services also have the potential to be used in order to directly access and manage client connections.

■

6.Architecture and Design

This chapter presents the architecture and design of the messaging platform conforming to the project requirements. It explains the structural components, feature design, and the design decisions that were made. It starts with explaining the three types of servers that are part of the platform, which are the GraphQL, REST, and WebSocket server and the design of their respective APIs. The data models used for the platform and the process of data storage is described. Next, individual functionalities of the platform and the way they function are discussed. The third-party integrations with asynchronous channels and cognitive services are explained. Finally, Webhook is explained which is a way for non-WebSocket clients to receive events from the messaging platform. Enterprise Architect version 14 is used to model the architecture diagrams.

6.1 Messaging Platform

The messaging platform exposes APIs for client applications to incorporate real-time messaging features in their application. Front-end browser applications can use GraphQL and WebSocket based APIs as WebSocket is standardized across all browsers. REST APIs and Webhook REST APIs and callback events can be used by backend systems to add real-time functionality. This section elaborates on the different types of servers hosted by the messaging platform, their APIs, and other relevant information.

6.1.1. GraphQL Server

The GraphQL server consists of two types of endpoints which handles two types of connection requests.

- HTTPS endpoint handles the *Query* and *Mutation* operations over HTTPS connections.
- WebSocket endpoint handles *Subscription* operations that creates real-time bidirectional channel between clients and the server and facilitate distribution of events to connected clients.

Complete design of a GraphQL API consists of three steps which includes defining the GraphQL schema, connecting data sources, and defining resolvers. Resolvers connect API definitions in the schema to data sources.

Step 1: Defining GraphQL Schema

The GraphQL schema of the messaging system consists of:

- **User defined types –** They serve as the building blocks for the schema. The user defined types are *Application*, *Identity*, *Group*, *Message*, *Presence*, and *TypingIndicator*. A domain model representing relationships between these user defined types is shown in Figure 7.

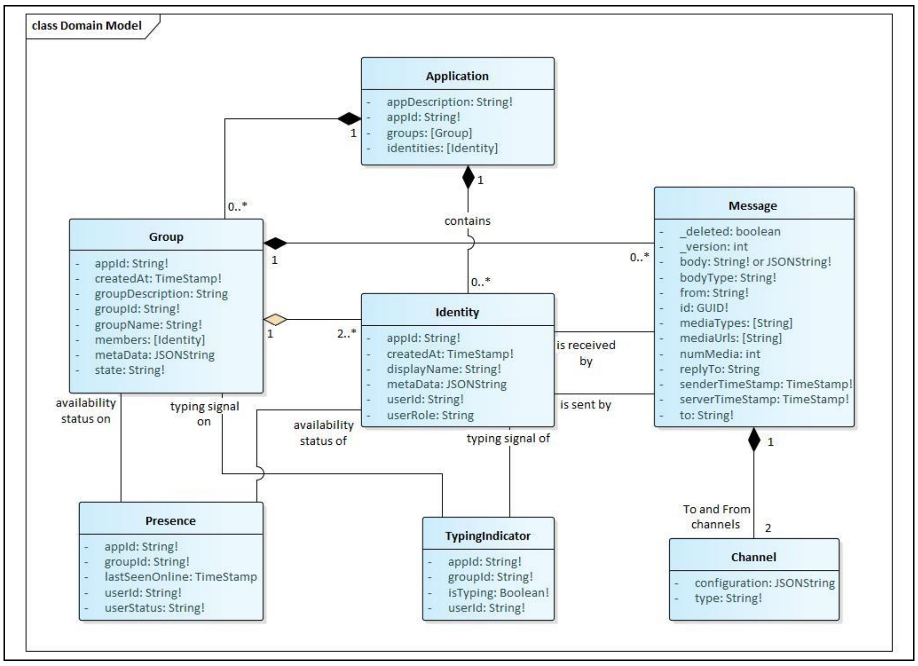


Figure 7– Domain model representing GraphQL user-defined types

- **Base types –** The base types in any GraphQL schema are *Query*, *Mutation*, and *Subscription*. They consist of operations which are used by clients as APIs. The base types use the pre-defined and user-defined data types. Figure 8 visually represents the base types and their dependency on user defined types. The detailed GraphQL schema can be found in Appendix A.

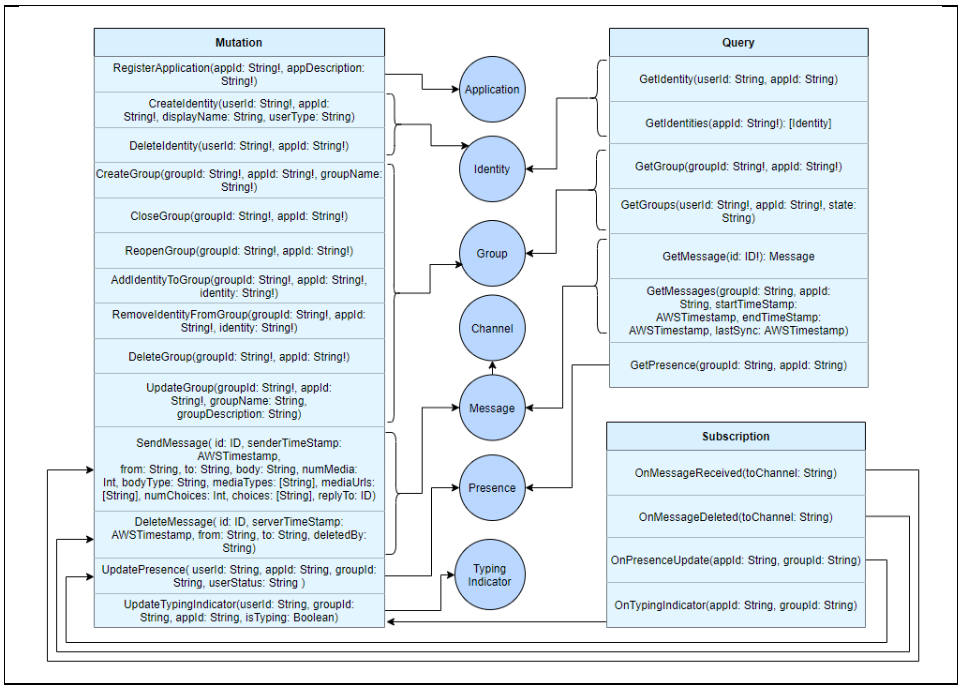

Mutation
RegisterApplication(appId: String!, appDescription: String!)
CreateIdentity(userId: String!, appId: String!, displayName: String, userType: String)
DeleteIdentity(userId: String!, appId: String!)
CreateGroup(groupId: String!, appId: String!, groupName: String!)
CloseGroup(groupId: String!, appId: String!)
ReopenGroup(groupId: String!, appId: String!)
AddIdentityToGroup(groupId: String!, appId: String!, identity: String!)
RemoveIdentityFromGroup(groupId: String!, appId: String!, identity: String!)
DeleteGroup(groupId: String!, appId: String!)
UpdateGroup(groupId: String!, appId: String!, groupName: String, groupDescription: String)
SendMessage(id: ID, senderTimeStamp: AWSTimestamp, from: String, to: String, body: String, numMedia: Int, bodyType: String, mediaTypes: [String], mediaUrls: [String], numChoices: Int, choices: [String], replyTo: ID)
DeleteMessage(id: ID, serverTimeStamp: AWSTimestamp, from: String, to: String, deletedBy: String)
UpdatePresence(userId: String, appId: String, groupId: String, userStatus: String)
UpdateTypingIndicator(userId: String, groupId: String, appId: String, isTyping: Boolean)
Application
Identity
Group
Channel
Message
Presence
Typing Indicator
Query
GetIdentity(userId: String, appId: String)
GetIdentities(appId: String!): [Identity]
GetGroup(groupId: String!, appId: String!)
GetGroups(userId: String!, appId: String!, state: String)
GetMessage(id: ID!): Message
GetMessages(groupId: String, appId: String, startTimeStamp: AWSTimestamp, endTimeStamp: AWSTimestamp, lastSync: AWSTimestamp)
GetPresence(groupId: String, appId: String)
Subscription
OnMessageReceived(toChannel: String)
OnMessageDeleted(toChannel: String)
OnPresenceUpdate(appId: String, groupId: String)
OnTypingIndicator(appId: String, groupId: String)

Figure 8 – Relationships between GraphQL base types and user-defined types

Step 2: Attaching Data sources

Three types of data sources are added to GraphQL server, which are a) Database sources mapped to underlying system databases, b) HTTP sources maps to external HTTP APIs which are used to integrate third-party services or other hosted services, and c) Computation function sources that maps to the source code. Every *Query* and *Mutation* operation is attached to a data source while a *Subscription* operation is tied to a *Mutation*. *Query* operations retrieve data and *Mutation* operations manipulate data from the data sources.

Step 3: Defining Resolvers

Resolvers contain the logic to query or manipulate the data sources. Each API operation defined in the schema is linked to a resolver, e.g., mutation *AddMemberToGroup* is attached to a database source *Groups* and a computation function data source. This resolver adds members to a group and returns the modified group object as the response. Figure 9 represents how every GraphQL request is processed using pipeline resolvers. Resolver pipelining is when two or more resolvers work together to respond to a request.

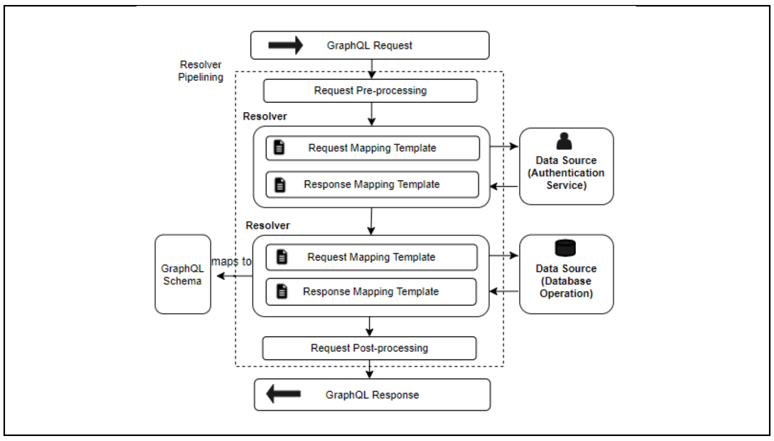


Figure 9 – GraphQL pipeline resolvers

6.1.2. REST Server

The REST server is primarily added for backend applications to communicate with the messaging platform which may not support WebSocket. It works on the request/response paradigm. REST interfaces are easy to integrate as they have been standardized for quite some time and has libraries across programming languages and platforms. All the operations which are supported by the GraphQL APIs are also supported by REST APIs. The underlying components for REST APIs are the data later and compute functions. REST interfaces expose the following functionalities.

- The set of messaging operations that are exposed by GraphQL APIs are also available as part of REST interfaces which includes grouping, sending and receiving messages, and presence information.
- Webhook registrations – To send real-time events to backend systems, Webhooks are supported by the messaging platform, and clients can register their URL via REST interfaces. More details are in Section 6.5.
- Multimedia integration – Interfaces to upload and retrieve binary files.
- Third-party service integrations – Interfaces for integrating additional services, e.g. AI services.

6.1.3. WebSocket Server

Most of the GraphQL services do not provide a callback mechanism to track the connected clients or to retrieve information when a client connects or disconnects. In order to know the online or offline status of each user, a separate WebSocket server is included in the messaging system to track client connections. The WebSocket server is responsible for the following functions and an API is exposed to track active channels.

- Track new users and their connection details.
- Maintain records for all the connected clients.
- Track the active channel of the client and track changes to the active channel.
- Track every connection drop and its associated user.

6.2 Data modeling and Data Storage

This section elaborates on the data layer of the messaging platform and how text-based messages and multimedia messages are stored. It also explains the database schemas and the data model used by GraphQL and REST APIs.

6.2.1. Messaging Data Model

The message data model describes how individual messages are represented and identified by the system. These data objects are response objects for GraphQL and REST APIs. Table 12 describes this data model.

Table 12 – Messaging data model

Field	Field type	Mandatory	Description
id	GUID	No	Unique identifier for every message. Auto-generated if not provided.
sender-TimeStamp	Timestamp	Yes	Time (in milliseconds) at which the message was sent by the client.
server-TimeStamp	Timestamp	No	Time (in milliseconds) at which the message was received by the messaging platform.
from	String	Yes	Represents sender information. Refer Section 6.4 for details.
to	String	Yes	Represents recipient(s) information. Refer Section 6.4.
bodyType	String	No	Data type of the message *body*. Defaults to String. Any other data type e.g., JSON can also be used
body	String	Yes	The actual information to be transferred and the format varies based on *bodyType* parameter.
numMedia	Integer	No	Number of media files attached with the message.
medi-aTypes	List [String]	No	Multimedia files types, e.g., image or video formats.
mediaUrls	List [String]	No	Unique media URLs associated with individual files.
replyTo	GUID	No	Reply to an existing message by referencing *id*.
version	Integer	No	Version information of a message, to enabling message editing and retaining each update.

_deleted	Boolean	No	Flag to denote if message was deleted. The message body and media fields are cleared if the value is true.

6.2.2. Database Modeling

A NoSQL database is used for data storage of the messaging platform as data is scattered and they are better at handling sporadic data, e.g., MongoDB, DynamoDB. Figure 10 represents all the physical tables, the fields, data types, and the primary keys and indexes associated to the tables. It is modelled in Enterprise Architect and it currently does not support modeling of NoSQL databases. Therefore, SQL database modeling is used, and the SQL column names and mapped to NoSQL fields based on the Note shown in the figure, e.g., CLOB in SQL mapped to JSON in NoSQL.

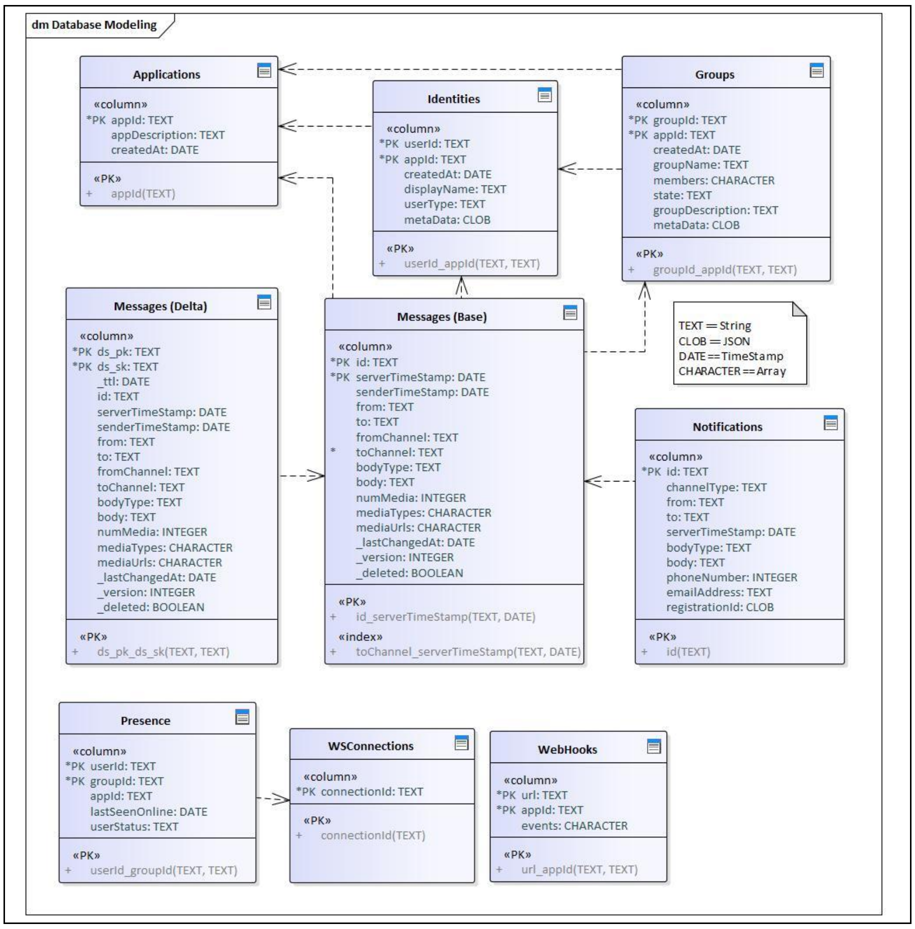


Figure 10 – Database modeling

Applications, *Identities*, and *Groups* tables stores registered applications, their associated identities and groups respectively.

The *Messages (Base)* table stores all the messages and the secondary index is added for faster execution of channel specific queries. The *Messages (Delta)* contains messages for 24 hours in order to support data synchronization and faster data retrieval. This table acts as a journal of the changes since the user was last seen online. The primary key is a combination of *ds_pk* and *ds*_sk. *ds_pk* is formed by concatenating the base table name and the date when the message was received (e.g., Messages:2020-07-28). *ds_sk* is formed by concatenating server timestamp, id, and the message version. The combination of these fields guarantees uniqueness for every entry in the Delta table (e.g., for time 03:44:18, id of bf45513c-4b72 and version of 2, *ds_sk* would be 03:44:18:bf45513c-4b72:2)

Notifications table stores all the SMS, Email, and Web Push messages.

Presence table stores the real-time status of all the users specific to groups and *WSConnections* table tracks all active user connections with the messaging system.

WebHooks table stores all the registered webhooks registrations.

6.2.3. Multimedia Storage

The multimedia files shared through the messaging platform are stored in an object storage system which can store and retrieve any amount of binary data. The distribution of files can be very resource intensive for the message broker, especially for large sized binary files as each file needs to be copied and sent to the clients. Hence, to optimize multimedia sharing, a unique URL is assigned to every multimedia file and the URL is distributed to the clients instead of the binary file. Sending multimedia is a two-step process.

1) Retrieve the signed URL and the media URL from the messaging system. (Signed URL is used to directly upload multimedia to the object storage system and is valid up to one hour. Media URL is the link at which multimedia file is available). The media URL is available only after the file has been uploaded successfully.
2) Use the signed URL to upload multimedia file.

6.3 System Feature Realization

This section elaborates on how different features and functionalities are realized by the messaging platform. Various UML activity diagrams are used to demonstrate their behavior.

6.3.1. Real-time behavior

Query and *Mutation* operations use the HTTPS request-response cycle which establishes a TCP connection and the connection is closed after every request. *Subscription* operations keep the connection between the client device and the messaging platform open which is facilitated by the GraphQL real-time server. The *Subscription* operations supported by the messaging system sends real-time events via the WebSocket connection to the clients when a corresponding *Mutation* is triggered. The subscriptions supported by the platform are listed in Table 13. For example, the subscription *OnMessageReceived* is tied to the Mutation *SendMessage* and accepts *groupId* and *appId* as filters, which means every time a message is sent and it matches any of the clients' subscription filters, the data associated with that *Mutation* is sent to all the message subscribers. Figure 11 shows how a real-time connection is established with the messaging platform and how mutations trigger subscriptions.

Table 13 – List of subscription operations

Subscription	Mutation	Description
OnMessageReceived	SendMessage	Sends events when new messages are added to the system
OnMessageDeleted	DeleteMessage	Sends events when existing messages are deleted from the system
OnPresenceUpdate	UpdatePresence	Sends events when a group member presence status changes
OnTypingIndicator	UpdateTypingIndicator	Sends events when a group member typing status changes

6.3.2. Presence Indicators

Presence is a means for knowing, retrieving, and receiving real-time changes in the online and offline status information of other users. GraphQL resolvers and WebSocket routes are used to realize this functionality. User-defined presence statuses, e.g. *Busy, DND* can also be set apart from online and offline statuses. Figure 12 shows how presence functionality is realized with the help of an example group *Gr1* consisting of three users.

Typing indicator is a means to know the real-time typing status of other group members and is realized in a similar manner by only using GraphQL resolvers. It sets a flag to denote start of typing and resets the flag at the end of typing. The recommendation is to use the typing indicator mutation with a timeout period of 6-8 seconds in the client applications side before triggering the next mutation.

6.3.3. Security and Authentication

The requests to the messaging system are only served over HTTPS and WSS on port 443 in which the data is encrypted using Secure Sockets Layer (SSL). Requests via port 80 for HTTP and WS requests are simply rejected by the messaging platform. SSL protects the communications against man-in-the-middle attacks and data tampering.

Authentication to the messaging system is realized using a third-party HTTPS authentication service which generates and validates tokens. The service backend generates tokens valid for a certain interval specified by the application. The tokens used by front-end applications should be generated frequently as they are user facing while the frequency of generating tokens by backend service can be less frequent. Figure 13 shows the sequence diagram of the authentication process.

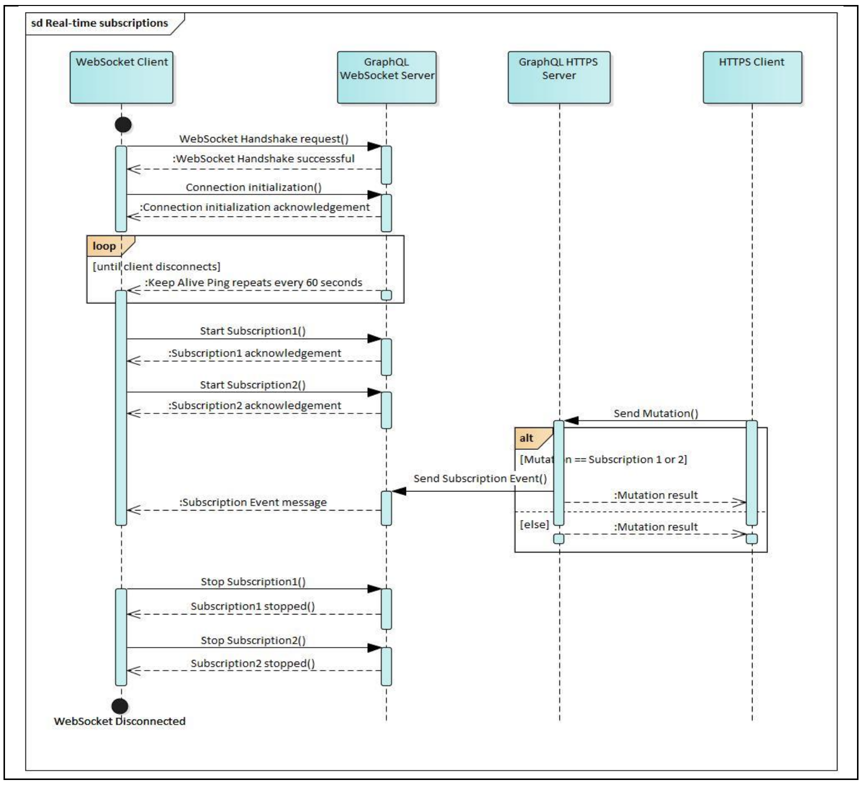

sd Real-time subscriptions
WebSocket Client
GraphQL WebSocket Server
GraphQL HTTPS Server
HTTPS Client
WebSocket Handshake request()
:WebSocket Handshake successsful
Connection initialization()
:Connection initialization acknowledgement
loop
[until client disconnects]
:Keep Alive Ping repeats every 60 seconds
Start Subscription1()
:Subscription1 acknowledgement
Start Subscription2()
:Subscription2 acknowledgement
Send Mutation()
alt
[Mutation == Subscription 1 or 2]
Send Subscription Event()
:Subscription Event message
:Mutation result
[else]
:Mutation result
Stop Subscription1()
Subscription1 stopped()
Stop Subscription2()
Subscription2 stopped()
WebSocket Disconnected

Figure 11 – GraphQL subscription sequence diagram

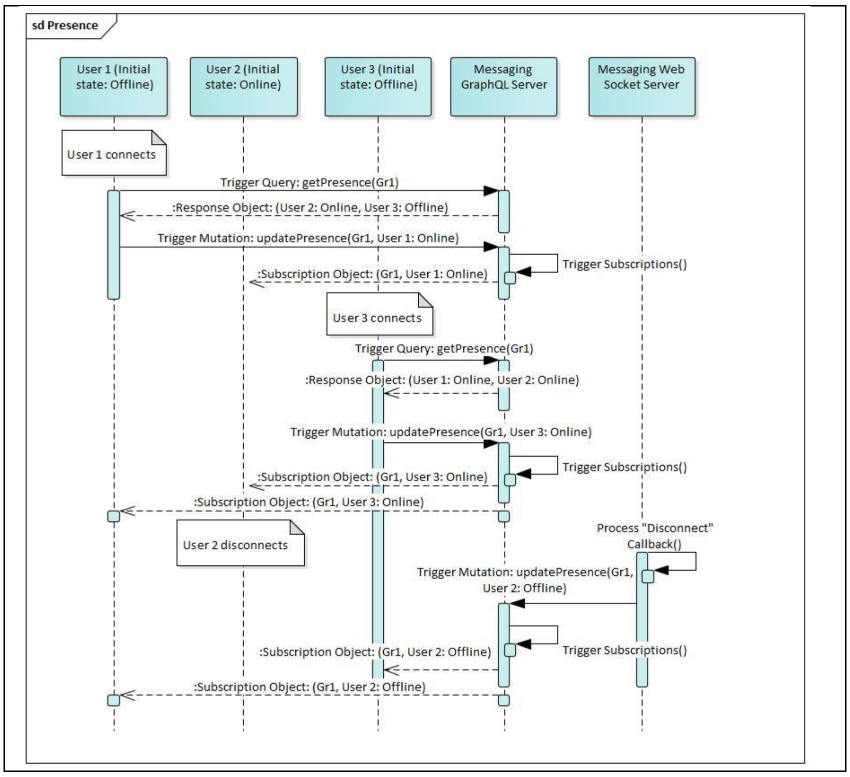

sd Presence
User 1 (Initial state: Offline)
User 2 (Initial state: Online)
User 3 (Initial state: Offline)
Messaging GraphQL Server
Messaging Web Socket Server
User 1 connects
Trigger Query: getPresence(Gr1)
:Response Object: (User 2: Online, User 3: Offline)
Trigger Mutation: updatePresence(Gr1, User 1: Online)
Trigger Subscriptions()
:Subscription Object: (Gr1, User 1: Online)
User 3 connects
Trigger Query: getPresence(Gr1)
:Response Object: (User 1: Online, User 2: Online)
Trigger Mutation: updatePresence(Gr1, User 3: Online)
Trigger Subscriptions()
:Subscription Object: (Gr1, User 3: Online)
:Subscription Object: (Gr1, User 3: Online)
Process "Disconnect" Callback()
User 2 disconnects
Trigger Mutation: updatePresence(Gr1, User 2: Offline)
Trigger Subscriptions()
:Subscription Object: (Gr1, User 2: Offline)
:Subscription Object: (Gr1, User 2: Offline)

Figure 12 – Real-time presence updates

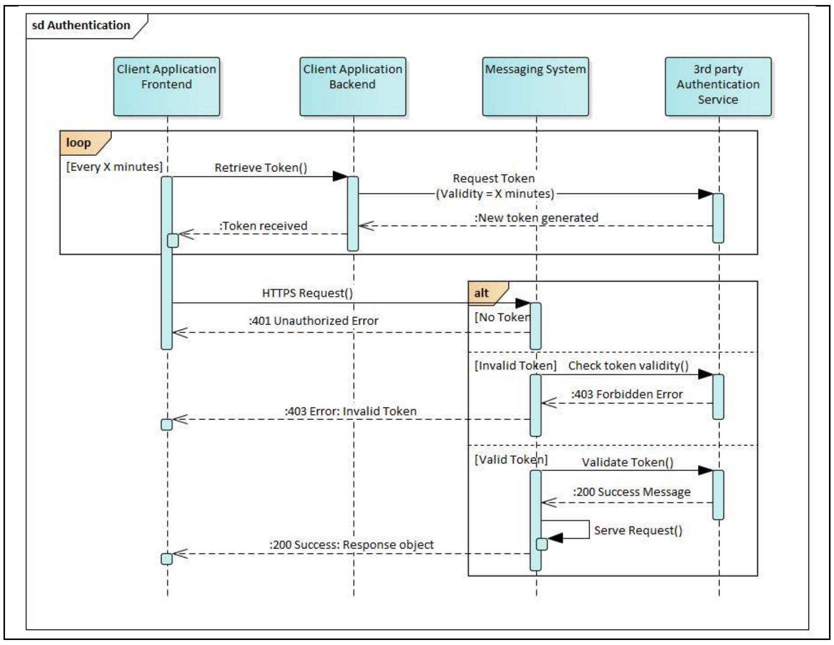


Figure 13 – Authentication sequence diagram

6.3.4. Multimedia Access

A Content Delivery Network (CDN) service is used to serve multimedia files. The files are retrieved by directly invoking the unique multimedia URL over the CDN service. A CDN is a geographically distributed network of data centers, which provide high availability and performance by serving static content from data centers located closest to the end users. The CDN service caches resources locally to serve the content faster for subsequent requests in order to facilitate low latency transfers. The activity flow for sending, uploading, and retrieving multimedia files is shown in Figure 14.

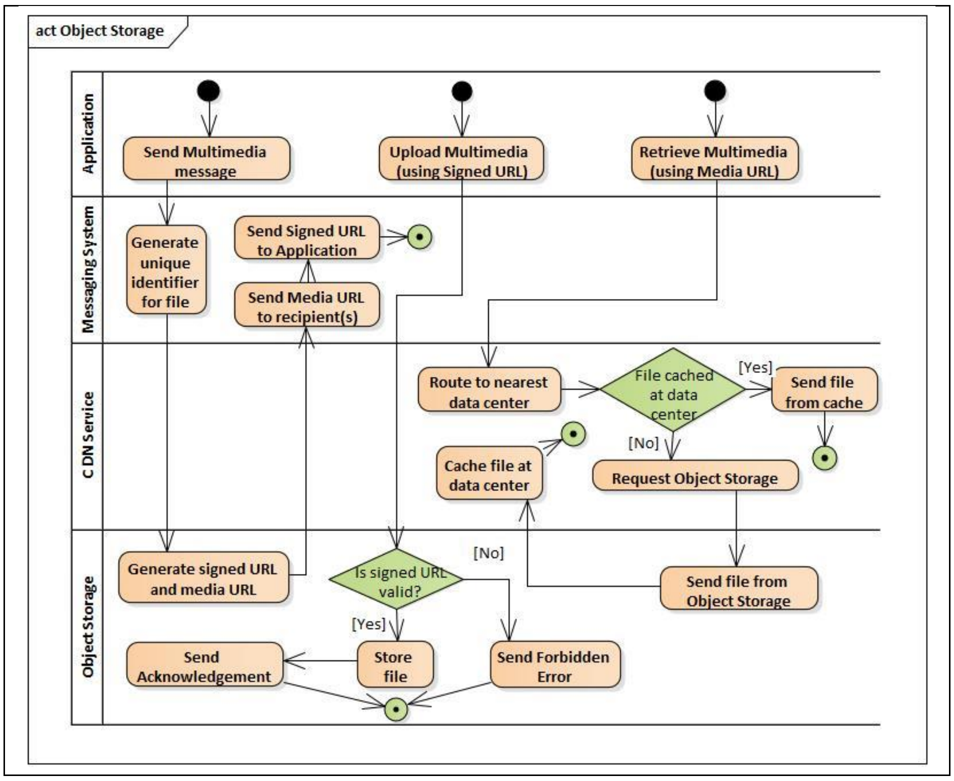

act Object Storage
Application
Messaging System
CDN Service
Object Storage
Send Multimedia message
Upload Multimedia (using Signed URL)
Retrieve Multimedia (using Media URL)
Generate unique identifier for file
Send Signed URL to Application
Send Media URL to recipient(s)
Route to nearest data center
File cached at data center
[Yes]
Send file from cache
[No]
Request Object Storage
Cache file at data center
Generate signed URL and media URL
Is signed URL valid?
[No]
[Yes]
Send file from Object Storage
Send Acknowledgement
Store file
Send Forbidden Error

Figure 14 – Activity diagram for sending, uploading, and retrieving multimedia files

6.3.5. Message Sequencing and Retrieval

Message Sequencing ensures that the messages are delivered to client applications in a particular order. The two timestamps captured by the messaging model are:

- **senderTimeStamp** – timestamp in milliseconds added by the client at the time when a message is sent. In cases when the client application is offline or when there are network inconsistencies, the senderTimeStamp can be different from the time when it was actually sent and the time when the messaging platform receives the messages and distributes to recipients.
- **serverTimeStamp** – timestamp in milliseconds added by the messaging platform when the message gets acknowledged by the backend.

The serverTimeStamp value acts as source of truth and is used to synchronize and sequence messages.

Retrieving messages from history or messages received on the real-time channel are always returned in the order sorted by *serverTimeStamp*. Some additional filters are added to filter messages based on certain criteria.

1) **Timestamps** – Retrieves messages between the specified start timestamp and the end timestamp. If only start timestamp is present in the request, messages between the specified time until the latest messages are returned. If only end timestamp is present, messages prior to the specified time are returned.
2) **Limit** – Limits the maximum number of messages which can be retrieved in one request. The maximum and the default value is set to 100 and can be reduced by client applications. If the number of messages corresponding to a request exceeds the limit, a key is sent along with the response for subsequent requests.
3) **Last Evaluated Key** – This property in returned by the messaging platform to indicate that additional records are present for the request. The subsequent request shall use this key to retrieve the additional paginated messages.

6.3.6. Data Synchronization

Every time the client application reconnects and requests for message history, it can be resource intensive to fetch the entire history data, especially for queries containing large number of records. This process is optimized by caching GraphQL responses locally in the browser cache of the web application. This allows clients to hydrate their local storage with results from one base query that might have a lot of records, and then receive only the data altered since their last connection, which are the delta updates. This is also efficient for client applications that frequently switch between online and offline states.

A Synchronization (Sync) Engine running at the client browser is responsible for managing the process by caching data in the browser local storage and performs two types of data fetching from the server, base hydration and delta hydration. This engine manages the invocations to the appropriate hydration.

- Base hydration refers to fetching the entire dataset of the in one batch and rehydrating the cache. The base hydration is triggered when client revisits the application any time after the base hydration time is lapsed (set to 24 hours). The *lastBaseSync* property is cached by the Sync engine to track the last base hydration time.
- Delta hydration refers to fetching only the incremental data (data altered since their last query) and updating the cache. The delta hydration is triggered when the client reconnects within 24 hours, irrespective of the number of reconnections. The *lastSync* property is cached and used by the Sync engine in order to track the client's last connected time.

The *Message (Base)* and *Message (Delta)* explained in Section 6.2.2 are used to handle the two types of hydrations. The delta table only keeps records for 24 hours after which the records are removed. When a GraphQL mutation is triggered, a record of that change is stored in the *Delta* table that is optimized for incremental updates. The delta query is more efficient than base query because the delta table is designed for timestamp-based queries which utilizes the database indexes while the base queries does a complete scan on the table. Items on the delta table are removed periodically to prevent from accumulating stale data and is in sync with the base table.

6.3.7. Offline Architecture

In scenarios when the network is not stable or when there is frequent switching between different networks, for example, hospital rooms protected against EM radiation, a synchronization (sync) engine running at the client's web browser provides a seamless experience to the clients. The sync engine interacts with the persistent storage system of the browser, which serves as a repository to store data locally. The Synchronization engine manages:

- Scanning the network connection status and synchronizing data with the servers when the network connection is established.
- Managing writes to the persistent storage of the browser when network is unstable or disconnected
- Maintaining a queue for pending requests and dequeuing when network connection is re-established.

Below section explains how different GraphQL operations are managed by the sync engine.
Query – The query results of the previous successful server operation are always in sync with the persistent storage data. So, when a query is requested in an offline state, the data from the cache is made available instead of fetching it from the network. The sync engine queues the query and waits for a change in network status. The query is automatically fired when the connection is restored, then the cache is updated, and additional data is synched with the cache.

Mutation – When a mutation is performed in an offline state, the mutation is cached and queued in the browser. The sync engine continuously checks the network connection status, and once the connection is restored, the mutation is fired by the sync engine without the client having to resend the message, which gives a seamless experience for users who frequently switch between different network error scenarios. The queue is emptied once the mutation is successful.

Subscription – The Synchronization engine also manages the subscription data by coordinating subscription reconnects and writes between offline to online transitions. The sync engine performs this by automatically resuming subscriptions and retrying through different network error scenarios and storing events in a queue. The appropriate delta or base query is then executed before merging any events from the queue and before finally processing subscriptions as normal.

The types of persistent storage systems [18] in browsers which are used.
- **Browser local storage** – It uses key value pairs to store data in the UTF-16 format and is suitable for storing simple data such as strings. Multimedia file references can be cached using the browser local storage by referencing system file paths, though it is not the recommended approach for multimedia files. The limits of local storage vary per browser but is usually in the range of 5-10MB.
- **IndexedDB** – It is available in browsers and provides a complete database system for storing complex data [19]. It can be used for caching raw binary files such as videos and images. The limit is upto 2GB, though it varies and is dynamically allocated based on the available disk space.

6.4 *Third-party Service Integrations*

Third-party services are used by the messaging platform in order to include additional features. Some of these services are authentication service, SMS, Email, and Web Push services. The platform also integrates AI services to add more insights to the data and add algorithms to the data. The idea is not exposing any third-party service dependencies on the client interfaces and have an abstraction layer for these services. This gives the flexibility to change the providers in the future without impacting end users.

6.4.1. Asynchronous Channel Services

The messaging system supports two inbuilt modes for sending messages which are peer-to-peer messaging and group messaging. Messages can be sent via three other asynchronous channels a) SMS message, b) Email message, and c) Web Push notification. The *from* and *to* field in the messaging data model denotes the sender, recipient, and channel information and varies per mode and channel type.
The *from* field contains the user, group, and application information in the format *chat:userId@groupId.appId*. The variations of the *to* field are mentioned in Table 14. The service providers for SMS, Email, and Web Push messages are abstracted as the model does not contain any provider specific information.

Table 14 – Messaging data model variations

Communication mode	Recipient variation	Description	Selected Provider
Peer-to-peer Message	*chat:userId@ groupId.appId*	Sends message to the specified user	-

Group Message	chat:*@*groupId.appId*	Sends message to all group members	-
SMS Message	sms:*phoneNumber*	Sends an SMS to the specified number. (*body* defines the SMS content)	SendGrid
Email Message	email:*emailId*	Sends Email to the specified address. (*bodyType* is email and *body* is JSON, containing Email subject and body.)	SendGrid
Web Push Notification	webpush:*registrationId*	Sends a web push notification. (*bodyType* is webpush and *body* is JSON, which defines web push registration.)	Google Firebase

6.4.2. Web Push Notifications

Web Push notifications are browser notifications which are pushed by a website to the client browser in response to a certain event. These notifications are clickable and can provide rich content messages. The web push protocol [20] enables communication between a user agent and a push service. The push service ensures reliable delivery of push messages while a user agent is actively using a web application or is in background window. There are two components of a push message, Push API and Notification API. The Push API is invoked when a server supplies information to a service worker. A Notification API is the action of a service worker or web page script showing information to a user. Web push notifications are supported by Chrome, Firefox, Safari, Opera, and Edge on Desktop and by Chrome, Firefox, Opera on Android Mobile. However, iOS does not support web push notifications yet.

In order for a Web Push notification to successfully sent, received, and displayed as browser notifications, the following components are involved and interactions between them are shown in Figure 15.

- Service worker – Tied to a specific website and processes the web push message based on the defined configuration. It can be configured to either display the notification or just persist in the browser storage.
- User agent – Identifies and activates the intended service worker and delivers the push message to it.
- Push service – Service that delivers the message to a specific user agent, identified by its push endpoint.
- Application server – Has the web push registration details of clients and invokes the messaging platform API in order to send a message to a specific user.
- Messaging system – Requests the push service to deliver a push message. This request uses the push endpoint included in the push subscription.

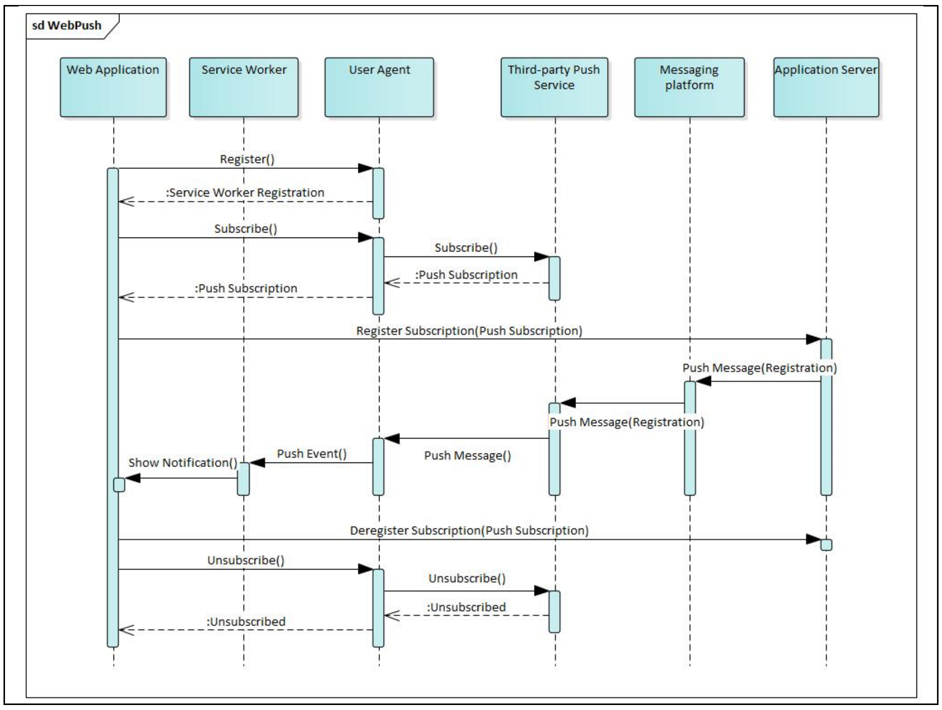


Figure 15 – Web push components and their interactions

6.5 Webhooks

The real-time behavior of the messaging system is facilitated via WebSocket using the GraphQL real-time end-point. GraphQL libraries are currently only available for front-end development frameworks, e.g., JavaScript, Angular. One of the system requirements is to facilitate sending and receiving real-time data with backend systems. In order to support real-time messaging with backend systems, Webhooks are part of the messaging system. Webhook is a way for the messaging platform to send real-time events to backend applications to their registered webhook URL. Backend applications can register their URL with the messaging platform and can add specific filters to restrict receiving only a subset of data. As soon as a webhook URL is registered, the backend application will start receiving events containing message data objects.

Database Streams are utilized to support real-time event generation. Database streams capture a time ordered sequence of every modification to the database table and is used to detect data changes. It tracks information about every modification to data items in the table, which can be addition of new data, modification of existing data, or deletion of data. The messaging platform listens for database streams objects, checks for all the registered webhooks, their filter conditions, and invokes the webhook URL by sending a HTTPS POST request. Figure 16 shows how Webhooks can be used to received real-time events.

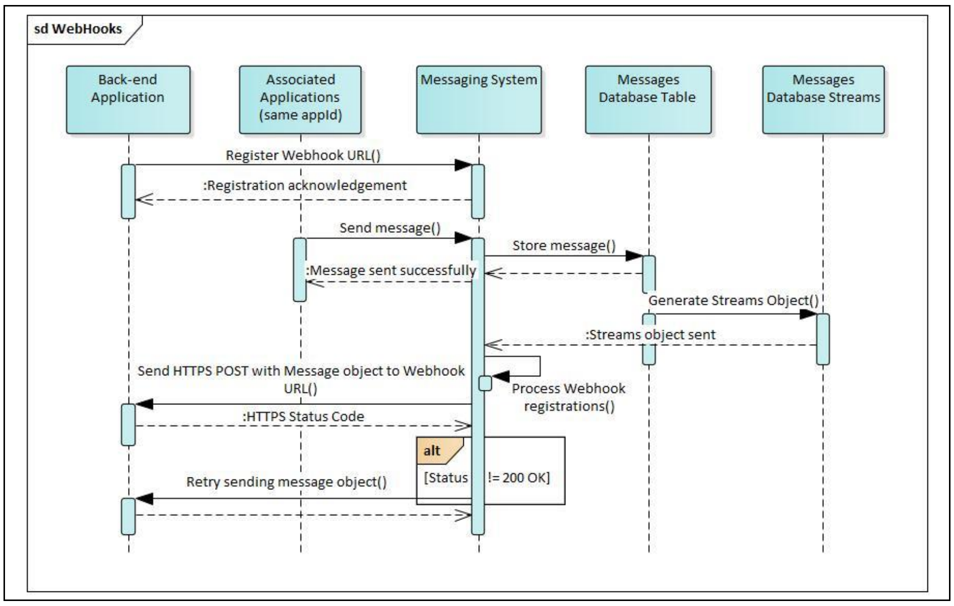

sd WebHooks
Back-end Application
Associated Applications (same appId)
Messaging System
Messages Database Table
Messages Database Streams
Register Webhook URL()
:Registration acknowledgement
Send message()
Store message()
:Message sent successfully
Generate Streams Object()
:Streams object sent
Process Webhook registrations()
Send HTTPS POST with Message object to Webhook URL()
:HTTPS Status Code
alt
[Status != 200 OK]
Retry sending message object()

Figure 16 – Webhook registration and callbacks using streams

6.6 Cloud and on-premise

The components and design explained in the previous sections of this chapter can be implemented and deployed on cloud or on premise or in a hybrid manner. The high-level concepts and design remain the same, but depending on the chosen technology stack, some aspects and configurations vary and may need tweaking. The on-premise implementation is kept out of scope for this project. Though the technology stack to realize this design can be quite diverse, an on-premise prototype using the stack listed in Table 15 was developed. The cloud implementation on AWS is explained in the next chapter.

Table 15 – List of potential cloud and on-premise technologies

	AWS Cloud	**On-Premise**
Database	DynamoDB	MongoDB (or any NoSQL DB)
Webhooks	DDB Streams	MongoDB streams
GraphQL API	AWS AppSync	Apollo GraphQL
REST API	REST API Gateway	Any REST library (e.g., Express.js in Node, Django in Python)
WebSocket API	WebSocket API Gateway	WebSocket libraries (e.g., Socket.IO in JavaScript)
Object Storage	AWS Simple Storage Service	Any on-premise object storage solution

■

7.Cloud Implementation

The previous chapter described the detailed design of the system and the final section listed the potential technology stacks that can be used for on-premise and cloud implementation. The scope of this project is limited to a cloud-based implementation. The messaging platform is implemented using Amazon Web Services (AWS) which is a platform by Amazon which operates globally providing technologies to develop applications on cloud. All the AWS resources and the third-party integration services that are used in realizing the platform are described. This chapter also explains some of the implementation choices such as the technologies and programming languages.

7.1 AWS Resources

The messaging platform is developed and hosted using a stack of AWS resources. AWS resources are entities which serve as building blocks to realize the end-to-end messaging system. Each resource is used for a certain functionality and the resources communicate with one another. For example, DynamoDB resource is used in the data layer to store text-based data and it communicates with lambda functions. Figure 17 shows the entire AWS stack used for the system implementation and their interactions. The resources are grouped as follows.

- Client-facing – The resources that are used to create and publish the platform APIs. These resources provide API endpoints, and client applications interact with the messaging platform via the APIs.
- Backend – The resources that are used to store and manage the data layer, platform logic, access management, and monitoring of the messaging platform.
- Software Development Kit – The AWS SDK that is used in order to support features like offline messaging and synchronization, which require interaction with the client browser side cache.

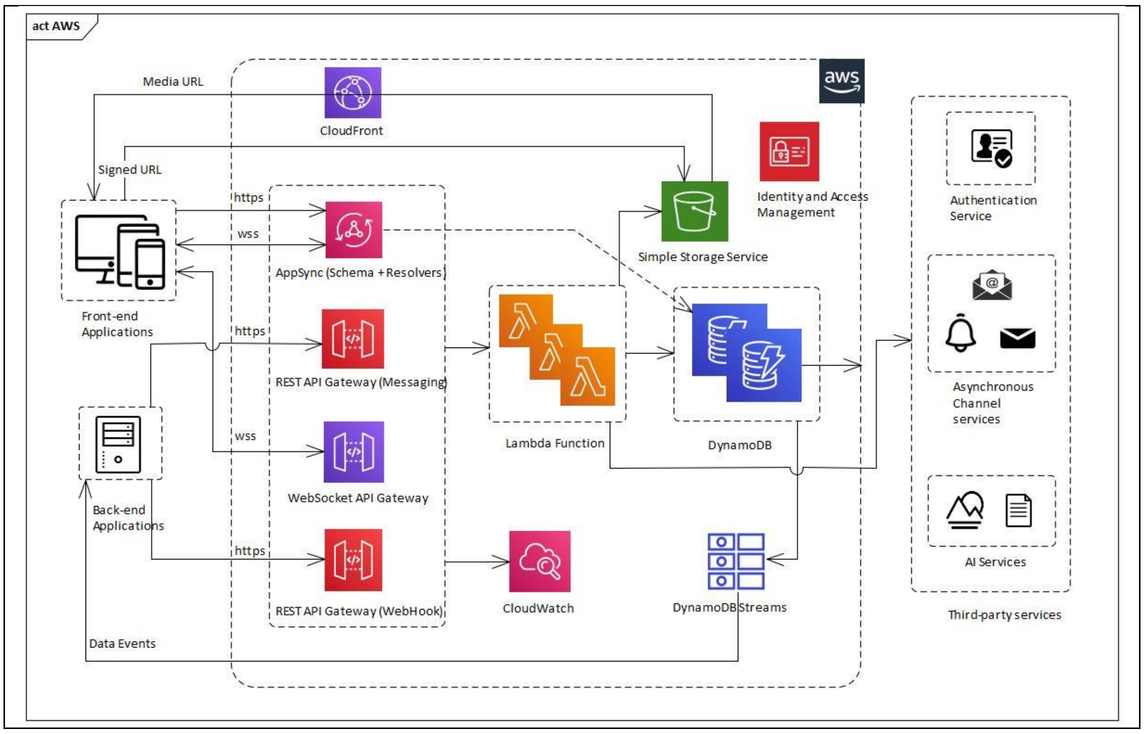


Figure 17 – Messaging platform AWS resources

7.1.1. Client-facing Resources

The messaging platform provides interfaces in the form of API endpoints which are of three types, GraphQL APIs, REST APIs, and WebSocket APIs. The details on the behavior and functionality of APIs explained in Chapter 6. Another client facing component is the Content Delivery Network service through which multimedia files are made available to the clients based on their geographical locations. Table 16 shows the list of client-facing AWS resources and their functionalities are explained in the subsequent section.

Table 16 – Client facing AWS resources

Type	AWS Resource
GraphQL API	AppSync
Messaging REST API	REST API Gateway
Webhook REST API	REST API Gateway
WebSocket API	WebSocket API Gateway
CDN	CloudFront

GraphQL API
AWS AppSync [21] is used as the GraphQL service which enables to create a flexible API to securely access, manipulate, and receive real-time updates from different data sources. It consists of two types of endpoints a) HTTPS endpoint manages *Query* and *Mutation* requests, b) WebSocket Endpoint manages *Subscription* requests and opens bi-directional communication channel with clients. Figure 18 shows the interaction between client applications and AppSync components.

1) **GraphQL Schema**: Describes the data objects and API input and output parameters. Refer Appendix A.
2) **Data Sources**: AppSync supports configuration of different data sources that are listed below.
   - DynamoDB Data Source – Connects DynamoDB tables to AppSync
   - HTTPS Data Source – Connects external HTTPS endpoints, e.g., Authentication API
   - Lambda Function Data Source – Connects AWS Lambda functions.
3) **Resolvers**: Resolvers serve as the link between APIs defined in the schema and the data sources. Each resolver is configured to one or more data sources and uses a request mapping and a response mapping template written in a DSL called Velocity Template Language (VTL). Resolvers transform the request into the language of the data source and transforms the response before sending it back to the client.

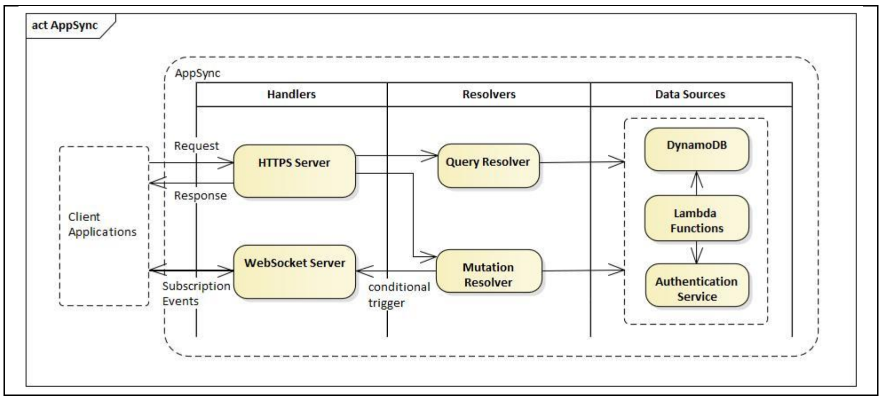


Figure 18 – Interaction between AppSync components

The GraphQL APIs of the messaging platform is described in Table 17.

Table 17 – Messaging platform GraphQL and REST APIs

Category	GraphQL Operation	REST Operation	API	Description	Data Source*
Application	Mutation	POST	RegisterApplication	Registers a new application with the messaging platform	Lambda
Identity	Mutation	POST	CreateIdentity	Creates a new identity specific to an application	Lambda
		PATCH	UpdateIdentity	Updates identity related data and metadata	Lambda
		DELETE	DeleteIdentity	Removes an application identity from the platform	Lambda
	Query	GET	GetIdentities	Retrieve all the identities registered with an application	Lambda

		GET	GetIdentity	Retrieve details of a specific identity	Lambda
Grouping	Mutation	POST	CreateGroup	Creates a new group specific to an application	Lambda
		PATCH	CloseGroup	Closes a group in order to stop sending and receiving messages from the group	Lambda
		PATCH	ReopenGroup	Reopens a closed group. Sending and receiving group messages are resumed	Lambda
		PATCH	AddIdentity	Adds a new identity to a group	Lambda
		PATCH	RemoveIdentity	Removes an existing identity from a group	Lambda
		DELETE	DeleteGroup	Removes an application group from the platform	Lambda
		PATCH	UpdateGroup	Updates group related data and metadata	Lambda
	Query	GET	GetGroups	Retrieve all the groups registered with an application	Lambda
		GET	GetGroup	Retrieve details of a specific group	Lambda
Messaging	Mutation	POST	SendMessage	Sends a message specifying the recipient, channel information, and other details.	HTTPS + DynamoDB
		DELETE	DeleteMessage	Soft deletes an existing message	Lambda
	Query	GET	GetMessages	Retrieves all messages from the groups based on filter conditions	HTTPS + DynamoDB
		GET	GetMessage	Retrieve details of a specific message	Lambda
	Subscription	*NA*	OnMessageReceived	Receive notifications when new messages are added to the subscribed groups	Lambda
		NA	OnMessageDeleted	Receive notifications when messages are deleted from the subscribed groups	Lambda
Presence and typing	Mutation	POST	UpdatePresence	Updates the online and offline status of a group member	Lambda
		POST	UpdateTypingIndicator	Updates the typing status of a group member	Lambda
	Query	GET	GetPresence	Retrieves presence information of group members	Lambda
	Subscription	*NA*	OnPresenceUpdate	Receive notifications when a group member changes presence state	Lambda
		NA	OnTypingIndicator	Receive notifications when a group member changes typing state	Lambda

* All the Lambda based data sources internally reference the HTTPS Authentication API

Messaging REST APIs

AWS API Gateway [22] REST is used to develop and publish the messaging REST APIs which use a request-response model to send data to clients synchronously and can manage a very high number of concurrent requests. Cross-Origin Resource Sharing (CORS) is enabled. All the *Query* and *Mutation* APIs of GraphQL are also exposed as REST APIs to support backend applications. Refer Table 17 for the list of REST APIs. The common APIs part of both GraphQL and REST share the same Lambda functions that contain the operation logic and data layer interactions.

The additional REST APIs for multimedia exchange and logging are explained below.

- GetMultimediaURL – Returns unique media URL from which the file can be accessed and a signed URL which is used to upload files. Validity of signed URL is one hour after which it needs to be regenerated.
- UploadMultimedia – Uploads the multimedia file to the object storage system using the signed URL.
- DeleteMultimedia – Deletes the multimedia files from the object storage system
- Logging – Generates a dump of DynamoDB tables into a read-only storage, which can be used by clients for use cases such as analytics.

WebSocket APIs
AWS API Gateway WebSocket is used to develop and publish the WebSocket APIs that enables real-time bidirectional communication in order to know users' connection status. Lambda functions are used to support the route operations. The three WebSocket routes are as follows.

- connect – Gets automatically invoked when a persistent connection between client and the messaging platform is initiated. Connections are tracked in the *Connections* table.
- disconnect – Gets automatically invoked when clients disconnect from the system. Disconnected client configurations are removed from the platform.
- updateConnection – A custom route that is used to track the currently active group of the client so that other group members are aware of the connection status.

Webhook REST APIs
AWS API Gateway REST is used to develop and publish the Webhook REST APIs. These APIs enable backend systems to receive real-time events from the messaging platform which lacks persistent connections. The Webhook APIs are as follows.

- RegisterWebHook – Enables clients to register their webhook URL with the messaging platform. The URL shall accept POST requests. The messaging platform pushes events to this URL when new information is available. An additional request is attempted if the first operation fails. DynamoDB Streams explained in Section 7.1.2 are used to generate the real-time events.
- UpdateWebHook – Updates webhook configurations details.
- GetWebHooks – Retrieve all webhook configurations tied to an application.
- DeleteWebHooks – Removes webhook configurations. Once removed, messaging system stops sending events.

Content Delivery Network
CloudFront is AWS CDN service that securely delivers multimedia files to clients globally with low latency and high transfer speeds. This is achieved by caching multimedia files to the nearest data centers and rendering files from these caches. The multimedia S3 URLs are accessible only via CloudFront as it enables faster file rendering based on the client geographical location. Lambda edge functions are added to authenticate client requests.

7.1.2. Backend Resources

The services explained in the previous section use different backend AWS resources which are the underlying resources in order to respond to API requests. These are database tables, database streams, object storage, roles and permissions managements and compute functions. Table 18 lists the AWS resource which are used as underlying components.

Table 18 – Backend AWS resources

Type	AWS Resource
Database	DynamoDB
Database Streams	DynamoDB Streams
Object Storage	Simple Storage Services (S3)
Compute Functions	Lambda functions
Access Management	Identity and Access Management (IAM)
Monitoring	CloudWatch

Database
AWS DynamoDB is a fast and flexible NoSQL database service that provides high performance and can serve high request traffic. The data layer of the messaging system is a set of DynamoDB tables which can store and retrieve large sets of data. There are two sets of tables instances, one for development and the other for the production environment. The tables are configured based on the database modelling explained in Section 6.2.2. The tables are configured to be on-demand which accommodates different levels of traffic workloads and makes it easy to balance cost and performance. The DynamoDB tables are: *Applications*, *Identities*, *Groups*, *Messages (Base)*, *Messages (Delta)*, *Notifications*, *Presence*, *WSConnections*, and *WebHooks*.

Database Streams
DynamoDB Streams captures time-ordered sequence of modifications in any DynamoDB table. Streams are enabled on the *Messages (Base)* table, which means any modifications to the table are tracked. A lambda function is initially configured and is triggered every time data is inserted, updated, or deleted from the table. The lambda function reads the stream, checks for registered webhook configurations, invokes the appropriate webhooks, and sends data to the webhook URL as events.

Object Storage
AWS Simple Storage Service (S3) is an object storage service that stores and protects large amount of data and offers high performance, scalability, and availability. The messaging platform uses S3 buckets to store and track the multimedia files. Each object in a S3 bucket is assigned a unique key-value pair where key is the unique file name and the value is the binary multimedia file. Individual objects are available as URLs in S3 identified by the key. The uniqueness of each file is guaranteed by concatenating a GUID, a milliseconds timestamp, and file name provided by the client. CORS is also enabled.

Compute Functions
AWS Lambda [23] are compute services that provides run time environments to source code written in a variety of programming languages and runs the code in a high-availability compute infrastructure and it also scales automatically. Lambda functions of the messaging platform are written in Node.js and Python 3 and are used as compute functions. These functions serve as the backbone of the platform as they contain the business logic and functionality. AWS Lambda communicates with other AWS Resources which are API Gateway, AppSync, CloudFront, DynamoDB Streams and S3.

Access Management
AWS Identity and Access Management (IAM) provides fine-grained access control to different AWS resources securely. It is achieved by using roles and policies. Communication between all AWS resources are managed using IAM. For example, in order for API Gateway to communicate with a lambda function, a policy is defined that specifies the access type, the allowed actions, and the resources for which those actions are applicable.

Monitoring
AWS CloudWatch provides data and insights in the form of logs, metrics, and events in order to monitor AWS resources in terms of performance and resource utilizations. All the AWS resources explained in the previous sections are enabled to provide logging via CloudWatch.

7.1.3. Software Development Kit

AWS AppSync provides a JavaScript Software Development Kit (SDK) [24] which is the GraphQL client library and has integrations with front-end frameworks such as React and Angular. It allows to easily develop UI components that send and retrieve data via GraphQL. The SDK supports queries, mutations, subscriptions and runs the caching and synchronization engine at the client browser. The SDK is therefore capable of providing offline features that uses the browser local cache as the storage engine and manages subscription handshaking. The SDK must be used in order to utilize the synchronization and offline features of the messaging platform.

7.2 Third-party Services

The messaging platform is integrated with several third-party services in order to facilitate some of the features and are available as part of the cloud implementation. These services are integrated in the Lambda functions and GraphQL resolvers. The different third-party services integrated with the messaging platform are as follows.

- SMS, Email, and Web Push channels – These services support messaging via different asynchronous channels. SendGrid provider is used for sending SMS and Email messages, and Google Firebase is used to send Web Push notifications.

- Authentication API – A token-based authentication service developed by Philips is integrated in order to authenticate each of the API requests.
- Analytics services – Few AI services are integrated as prototypes to demonstrate the potential value it can add to the multimedia data available as part of the messaging platform. Microsoft Azure Cognitive cloud services [25] are used for text analytics and multimedia analytics. Some of the integrated text analysis services are translation services (up to 80 languages), key phrases extraction, sentiment analysis, and named entity recognition.

7.3 Demo UI Application

A simple demo application is developed in Angular to demonstrate the features of the messaging platform. It uses the AppSync SDK to communicate with the GraphQL APIs. It also integrates with REST and WebSocket APIs. The features demonstrated by the application are grouping, sending and receiving messages, receiving presence and typing status, sequencing, synchronization and offline messaging. Web Push and text analytics services are also demonstrated. The Angular application is deployed on AWS S3 and is available via CloudFront.

■

8.Deployment and Pilot Execution

This chapter explains the deployment process and the Serverless Application Model which is used to configure, package source code, and deploy the messaging platform resources to AWS. An application named *Service Connect* was launched by Philips Research in line with the vision explained in Section 1.3.3. The messaging platform is one of the components being used in the pilot and this chapter explains how the platform fits into the big picture and how other components interface with the platform.

8.1 CloudFormation

AWS CloudFormation provides a way to create and manage a collection of AWS resources. The resources explained in Chapter 7 that constitutes the messaging platform are bundled together in a CloudFormation stack. It also enables to model and provision resources in an automated and secure way using templates. AWS Serverless Application Model (SAM) [26] is an open-source framework for building serverless applications. It provides resource specific syntax to express functions, APIs, databases, and event source mappings and is defined and modelled using YAML. During the deployment, SAM transforms this syntax into AWS CloudFormation syntax that creates and deploys the platform on AWS.

The YAML based SAM template consists of resource configurations and dynamic links to the source code. They are then packaged and uploaded to an S3 bucket. CloudFormation uses the YAML configurations and the source code located in S3 to create the stack of specified AWS resources. The stack of resources is then deployed, and they interface with one another based on the defined policies. Subsequent deployments to the same stack do not redeploy the resources and only updates the modified resources. Finally, the API endpoints are generated. The endpoints are created only when a new stack is deployed and subsequent deployments to the same stack references the existing endpoints. A snippet of the SAM template used to deploy the messaging system resources is available in Appendix B.

8.2 CI/CD

GitLab is used as the source code repository for the messaging platform. Gitlab CI/CD tool is used for Continuous Integration (CI), Continuous Delivery (CD), and Continuous Deployment (CD). The CI/CD configuration is managed by the gitlab-ci.yml file. This file creates a pipeline for changes in the code repository. Pipelines consists of stages and are executed in order and each stage consist of jobs that run simultaneously. These jobs are executed by Gitlab runners and executes inside a Docker container.

The messaging platform is deployed in three environments, testing, development, and production environments which maps to test, dev, and master branches in repository. The CI/CD pipeline consists of three stages.

- Validation – Runs unit tests and integration tests locally. Only if this validation stage succeeds, the next stage is triggered.
- Packaging – Packages the source code and uploads to an S3 bucket based on SAM template.
- Deployment – Pulls the source code from S3, creates AWS resources, and deploys the stack of messaging platform resources on AWS.

The CI/CD configuration is such that any changes committed to the test or dev branch creates a separate test or dev CloudFormation stack, and then packages and deploys the application in the respective environment. Any changes made to the master branch validates and packages the application and waits for a manual trigger to deploy the production version of the application. Redeployment of a stack only updates the resources that were changed compared to the previous deployment. If there are no modifications since the last commit, the stack is not updated. The GraphQL, REST, and WebSocket API endpoints are exposed as outputs after the deployment stage. There are three sets of messaging API endpoints that points to the three environments.

8.3 Service Connect Pilot

In line with the vision explained in Section 1.3.3, an application named *Service Connect* was launched for customer use at pilot sites in July 2020. The *Service Connect* application delivers a seamless end-to-end service experience by digital communication channels featuring tools improving remove communication workflow. Refer Appendix C for sample application screenshots. The pilot is scheduled to run for a couple of months by training pilot hospitals sites which results in receiving feedback on the digital support experience. The Service Connect application is a web application launched when a device QR code is scanned or when an invitation is sent to the user. The messaging platform developed as part of this project is one of the components being used in the pilot.
The messaging platform developed as part of this project is one of the components being used in the pilot.
The following hospitals in North America are the participating pilot sites.

- Duke University Hospital in North Carolina
- Southwest Regional Medical Center in Mississippi
- Wellstar Atlanta Medical Center in Georgia

The *Service Connect* application consists of the following components where each component is responsible for managing a set of functionalities and together shape the enhanced digital support experience. Figure 19 depicts the interactions between the pilot components.

1) **Service Connect (SC) Application –** The core application consists of a client-facing web application and a backend component. The backend is responsible for static and dynamic grouping, sending notifications, routing to Philips support, interfacing with WebRTC, and communication with other components.
2) **Messaging** – This component is the system developed during the project. This component acts as a central component to facilitate communication and provides interfaces to different components.
3) **Chatbot Orchestrator** – Manages different chatbot backends and merges the data into the messaging platform.
4) **Monitoring System** – Analytics and monitoring services using the messaging data.
5) **IPSA** – Contextual search engine and question answering system.

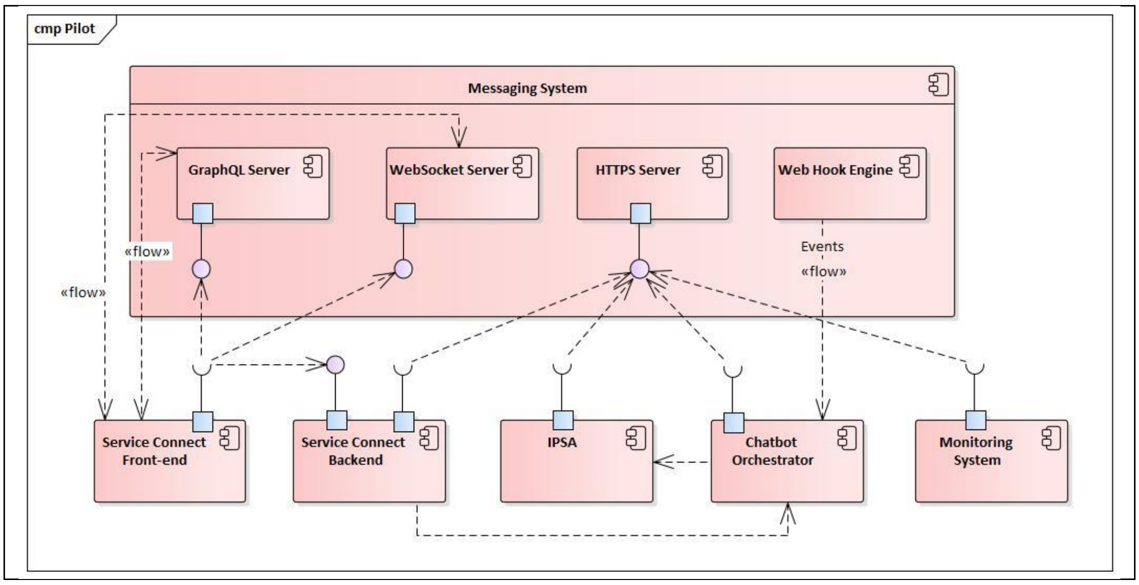


Figure 19 – Pilot components and interfaces with the messaging platform

9. Verification & Validation

The previous two chapters described the implementation and deployment of the platform. This chapter explains the process of verification and validation. This chapter revisits the requirements in Chapter 3 and lists the status of the requirements. They are categorized as: fully implemented, only the design is provided, and not implemented. As the platform was tested on pilot sites, metrics measuring performance over a span of three months were measured and are illustrated.

9.1 Verification

Verification is the process of checking whether the software meets the requirements. The developed platform is modular, hence components were developed and tested separately, and then collectively. Unit tests and integration tests are added to the deployment pipeline. Table 19 lists the status of the requirements.

Table 19 – Requirements status

Requirement Category	Completed	Design only	Not implemented
General System requirements	GR-001, GR-002, GR-003, GR-004, GR-005, GR-007, GR-008	GR-006	
FR - Registration	FR-001, FR-002, FR-003		
FR - Identity Management	FR-004, FR-005, FR-006, FR-007, FR-008, FR-009		
FR - Group Management	FR-010, FR-011, FR-012, FR-013, FR-014, FR-015, FR-018, FR-020	FR-017	FR-016, FR-019
FR - Sending messages	FR-021, FR-022, FR-023, FR-024, FR-025, FR-026, FR-027		
FR - Multimedia Support	FR-028, FR-029, FR-030, FR-031, FR-032, FR-033, FR-034, FR-035, FR-036		
FR - Receiving messages	FR-037, FR-038, FR-039, FR-040, FR-041, FR-042		
FR - Conversation History	FR-043, FR-044, FR-046, FR-047	FR-045	FR-048
FR - Delivery and Synchronization	FR-049, FR-050, FR-051, FR-052, FR-053, FR-054		
FR - Offline Messaging	FR-055, FR-056, FR-058	FR-057	
FR - Presence	FR-059, FR-060, FR-061, FR-062, FR-064, FR-065		FR-063
FR - Asynchronous channel integrations	FR-066, FR-067, FR-069	FR-068	
FR - Third-party integrations:	FR-070, FR-072, FR-073		FR-071

9.2 Validation

Validation of a software system is the process of checking whether the developed system satisfies the stakeholder needs. Validation was done in an iterative manner. In the progress and PSG meetings, the results were continuously validated by the stakeholders. The frequency of these meeting were every 2-3 weeks, and these meetings included demo of the current platform, and the future steps. Feedback was provided on the progress and the feedback was incorporated in the next design steps. The pilot customers also validated and provided inputs on the end-to-end system.

9.3 *API Metrics*

One of the important non-functional requirements of the system is time behavior. Latency of GraphQL APIs and REST APIs is measured and is explained below.

GraphQL APIs Latency – For a real-time system, the response time is one of the most important metrics to evaluate the performance. The GraphQL server is responsible for the real-time behavior. The time between the GraphQL server receiving a client request and returning a response to the client is the latency. This latency does not include the network latency to reach the end users. Figure 20 shows the latency in milliseconds between mid-June and end of August and is generated using AWS CloudWatch.

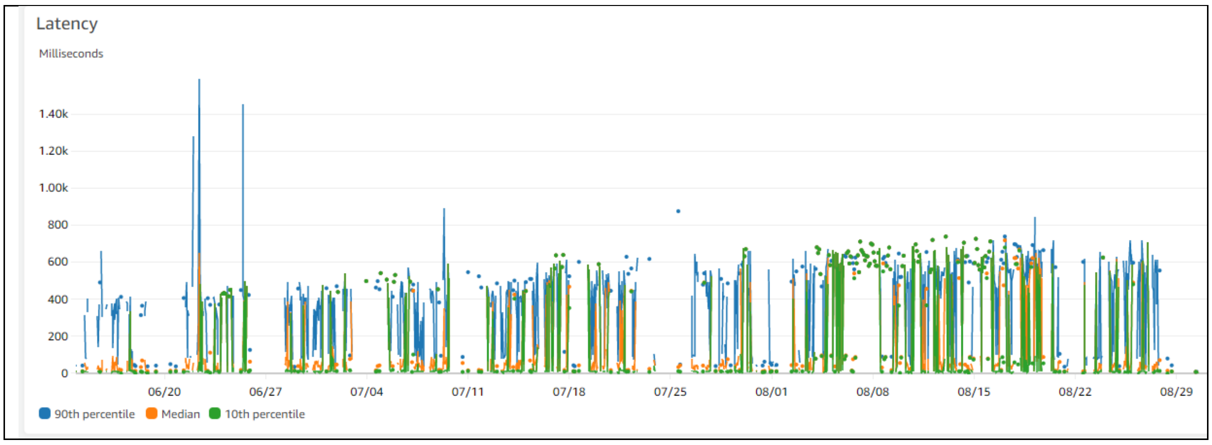


Figure 20 – GraphQL API latency graph

REST APIs Integration Latency – The time between the API Gateway relaying the request to the backend and receiving a response from the backend is Integration Latency. Figure 21 shows the average integration latency in milliseconds of the messaging REST APIs between June-July generated from AWS CloudWatch. Note that the response time is inclusive of a third-party HTTPS authentication API for which the average response time is one second.

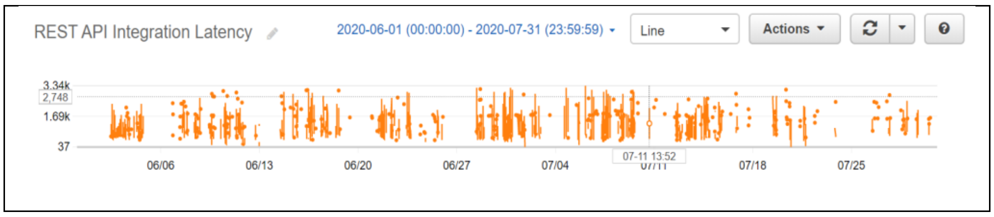


Figure 21 – REST API integration latency graph

REST APIs Latency – The time between the API Gateway receiving a request from the client and returning a response to the client is the overall latency. The latency is inclusive of the integration latency and other API gateway overheads. Figure 22 shows the overall latency in milliseconds of Messaging REST APIs between June-July generated from AWS CloudWatch.

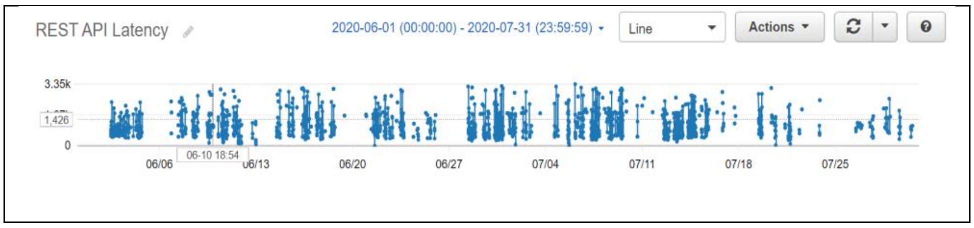


Figure 22 – REST API latency graph

■

10. Conclusion

This chapter discusses the results achieved with the project and the functionalities supported by the cloud messaging platform. It also summarizes the project deliverables, and recommendations for future work.

10.1 Results

The high-level objectives of the project are explained in Section 1.3.3. The design and implementation conform to the objectives and requirements. This section describes the results achieved from this project and assesses the results at different abstractions.

A unified messaging platform is implemented, offering core messaging functionalities and messaging via external channels. The core messaging functionalities are multimedia group messaging, optimized data synchronization, real-time presence updates, and offline messaging. The platform was tested in a pilot, launched by Philips Research in order to digitize the existing remote customer support experience. Refer Appendix C for details. The messaging platform offers a set of APIs which can be used by client applications in order to add features to their system, which are GraphQL APIs, REST APIs, and WebSocket APIs.

The proposed architecture and design of the platform is explained in Chapter 6. The cloud implementation is based on the proposed design. The interfaces are at a generic abstraction level so that various client applications can integrate the platform features for their use cases.

WebSocket is used as the underlying protocol. It inherently eliminates request-response connection overheads; hence the least amount of latency is observed when compared to other methods. It also facilitates full-duplex communication. It is standardized across all major browsers and is compatible with HTTP ports 80 and 443. Network firewalls are usually configured to not block HTTP(S) traffic; therefore, browser-based client applications can integrate with the messaging platform and add real-time capabilities to their systems.

Client applications not supporting WebSocket (e.g., backend applications or non-browser client applications) can also achieve near real-time capabilities by registering their webhook URL with the messaging platform. As and when new information is available, corresponding webhook URLs are triggered. The platform uses HTTPS to post data to the webhook and the latency is slightly higher compared to WebSocket.

Communication via SMS and Email channels are supported. Web Push notifications can also be sent via the platform. Communication via these channels are facilitated by third-party services and serve as integration points to the platform. The platform also demonstrates prototypes by application of third-party AI services to the text and multimedia data.

10.2 Project Deliverables

The tangible results delivered to the company in alignment with the project goals and requirements are as follows.

- AWS *Serverless Application Model* template, the configurations, and the source code of the messaging platform is available on Philips GitLab.
- Messaging platform APIs are hosted on AWS in two separate environments, development and production.
- Architecture diagrams modeled in Enterprise Architect.
- Analysis, comparison, and design documentation.
- API documentation and the source code of a demo front-end application.
- The PDEng graduation final project report is delivered both to the company and TU/e.

10.3 Future work

This section elaborates on potential additions and improvement points which can be made to the platform in the future. Some of the requirements that were not realized during the project or were kept at a design-only state due to time constraints, requirements prioritization, and other factors are also included in the recommendations for future work.

- Currently, only text-based offline messaging is supported. Multimedia offline messaging as explained in Chapter 6 using IndexedDB can be added to the platform.
- Currently, only three external channels are supported by the platform. Channels such as WhatsApp or other social media channels, e.g., Facebook Messenger can be integrated.
- Implementation of an on-premise solution based on the design and recommended technologies in Chapter 6.
- The platform APIs can be implemented as language specific SDKs and provided to clients for easier integrations.
- The platform was tested in the remote customer support environment. Exploring more use cases and using the system in different use cases and contexts.
- Integrations with external AI services were developed as prototypes. This extension can be evolved, and more insightful services can be added.

■

11. Project Management

This chapter elaborates on the project management process which was carried out during the lifetime of the project.

11.1 Work Breakdown Structure

A Work Breakdown Structure (WBS) was created during the first few weeks of the project that consisted of high-level work packages. The work packages and sub-tasks are as follows.

- Initiation – Project planning, risk assessment, and scheduling activities.
- Analysis – Problem analysis, stakeholder analysis, capturing requirements and domain exploration.
- Implementation – Developing system architecture and detailed design, implementation, verification, and deployment.
- Conclusion – Testing at pilot sites, validation, and documentation of the graduation report

11.2 Project Plan

The project plan is in line with the WBS. A project plan was drafted during the initial phase of the project based on the initial overview of project goals. Modifications to the initial project plan were done in an iterative manner when actions started becoming more concrete which accounted for sub-tasks in the plan. Minor modifications were also done to accommodate change in priorities. In the bi-weekly progress update meetings and monthly Project Steering Group (PSG) meetings with the supervisors, the ongoing progress and the next steps were discussed. These meetings also acted as alignment meetings to discuss prioritization in requirements and subsequent activities.

The project plan was tracked using Microsoft Project and was modified iteratively. The final project plan contains the activities that were actually carried out during the project. Figure 24 represents the Gantt chart which contains the high-level tasks, sub-tasks, and milestones. The figure also contains a visual representation of tasks scheduled over time. The high-level activities in the final plan are in sync with the initial plan. Figure 23 shows a burndown chart which visualizes how the tasks were completed over time.

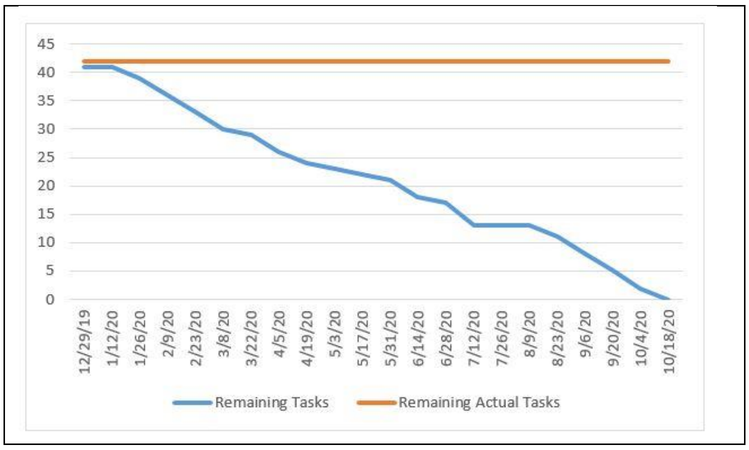
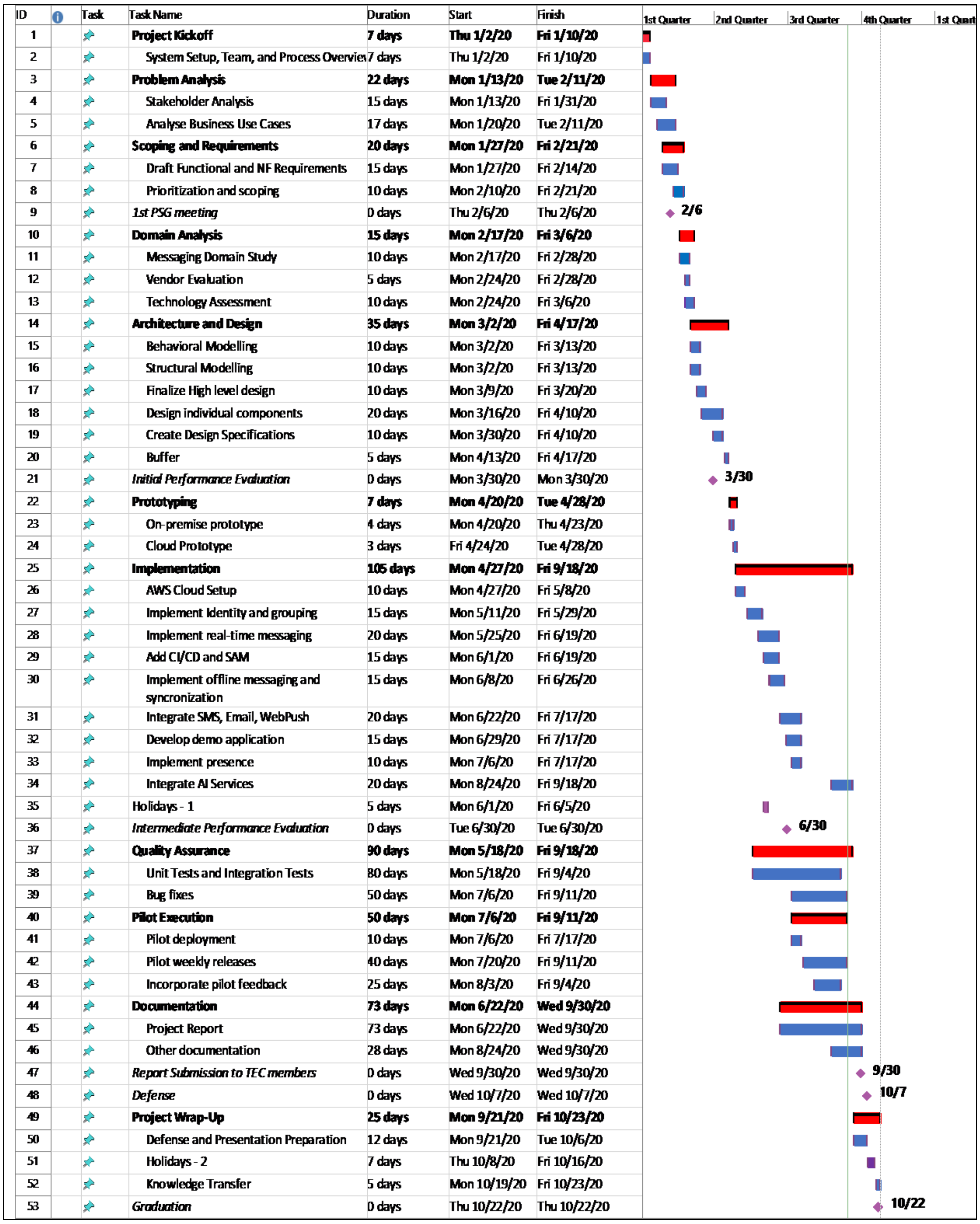


Figure 23 – Tasks burndown chart



ID	Task Name	Duration	Start	Finish
1	**Project Kickoff**	**7 days**	**Thu 1/2/20**	**Fri 1/10/20**
2	System Setup, Team, and Process Overvie	7 days	Thu 1/2/20	Fri 1/10/20
3	**Problem Analysis**	**22 days**	**Mon 1/13/20**	**Tue 2/11/20**
4	Stakeholder Analysis	15 days	Mon 1/13/20	Fri 1/31/20
5	Analyse Business Use Cases	17 days	Mon 1/20/20	Tue 2/11/20
6	**Scoping and Requirements**	**20 days**	**Mon 1/27/20**	**Fri 2/21/20**
7	Draft Functional and NF Requirements	15 days	Mon 1/27/20	Fri 2/14/20
8	Prioritization and scoping	10 days	Mon 2/10/20	Fri 2/21/20
9	*1st PSG meeting*	0 days	Thu 2/6/20	Thu 2/6/20
10	**Domain Analysis**	**15 days**	**Mon 2/17/20**	**Fri 3/6/20**
11	Messaging Domain Study	10 days	Mon 2/17/20	Fri 2/28/20
12	Vendor Evaluation	5 days	Mon 2/24/20	Fri 2/28/20
13	Technology Assessment	10 days	Mon 2/24/20	Fri 3/6/20
14	**Architecture and Design**	**35 days**	**Mon 3/2/20**	**Fri 4/17/20**
15	Behavioral Modelling	10 days	Mon 3/2/20	Fri 3/13/20
16	Structural Modelling	10 days	Mon 3/2/20	Fri 3/13/20
17	Finalize High level design	10 days	Mon 3/9/20	Fri 3/20/20
18	Design individual components	20 days	Mon 3/16/20	Fri 4/10/20
19	Create Design Specifications	10 days	Mon 3/30/20	Fri 4/10/20
20	Buffer	5 days	Mon 4/13/20	Fri 4/17/20
21	*Initial Performance Evaluation*	0 days	Mon 3/30/20	Mon 3/30/20
22	**Prototyping**	**7 days**	**Mon 4/20/20**	**Tue 4/28/20**
23	On-premise prototype	4 days	Mon 4/20/20	Thu 4/23/20
24	Cloud Prototype	3 days	Fri 4/24/20	Tue 4/28/20
25	**Implementation**	**105 days**	**Mon 4/27/20**	**Fri 9/18/20**
26	AWS Cloud Setup	10 days	Mon 4/27/20	Fri 5/8/20
27	Implement Identity and grouping	15 days	Mon 5/11/20	Fri 5/29/20
28	Implement real-time messaging	20 days	Mon 5/25/20	Fri 6/19/20
29	Add CI/CD and SAM	15 days	Mon 6/1/20	Fri 6/19/20
30	Implement offline messaging and syncronization	15 days	Mon 6/8/20	Fri 6/26/20
31	Integrate SMS, Email, WebPush	20 days	Mon 6/22/20	Fri 7/17/20
32	Develop demo application	15 days	Mon 6/29/20	Fri 7/17/20
33	Implement presence	10 days	Mon 7/6/20	Fri 7/17/20
34	Integrate AI Services	20 days	Mon 8/24/20	Fri 9/18/20
35	Holidays - 1	5 days	Mon 6/1/20	Fri 6/5/20
36	*Intermediate Performance Evaluation*	0 days	Tue 6/30/20	Tue 6/30/20
37	**Quality Assurance**	**90 days**	**Mon 5/18/20**	**Fri 9/18/20**
38	Unit Tests and Integration Tests	80 days	Mon 5/18/20	Fri 9/4/20
39	Bug fixes	50 days	Mon 7/6/20	Fri 9/11/20
40	**Pilot Execution**	**50 days**	**Mon 7/6/20**	**Fri 9/11/20**
41	Pilot deployment	10 days	Mon 7/6/20	Fri 7/17/20
42	Pilot weekly releases	40 days	Mon 7/20/20	Fri 9/11/20
43	Incorporate pilot feedback	25 days	Mon 8/3/20	Fri 9/4/20
44	**Documentation**	**73 days**	**Mon 6/22/20**	**Wed 9/30/20**
45	Project Report	73 days	Mon 6/22/20	Wed 9/30/20
46	Other documentation	28 days	Mon 8/24/20	Wed 9/30/20
47	*Report Submission to TEC members*	0 days	Wed 9/30/20	Wed 9/30/20
48	*Defense*	0 days	Wed 10/7/20	Wed 10/7/20
49	**Project Wrap-Up**	**25 days**	**Mon 9/21/20**	**Fri 10/23/20**
50	Defense and Presentation Preparation	12 days	Mon 9/21/20	Tue 10/6/20
51	Holidays - 2	7 days	Thu 10/8/20	Fri 10/16/20
52	Knowledge Transfer	5 days	Mon 10/19/20	Fri 10/23/20
53	*Graduation*	0 days	Thu 10/22/20	Thu 10/22/20

Figure 24 – Project Gantt chart

11.3 *Risk Analysis*

This section describes the risks that were identified during the initial phase of the project. A risk table was formulated containing descriptions of the risk items. For each risk item, the probability of occurrence and impact levels were analyzed and were assigned either High, Medium, or Low values. Mitigation strategies were also proposed and were accordingly prioritized and implemented to reduce adverse effects on the project goals. Table 20 shows the risk analysis table.

Table 20 – Risk assessment

Risk Type	Risk Item	Impact	Proba-bility	Mitigation
Process	Unscheduled holidays of supervisors	Medium	Low	Try to have planned work items for the next two to three weeks. If something is blocking due of absence of one supervisor, focus on other project related tasks in the time being.
Process	Trainee ill for a long duration	High	Low	Negotiate and prioritize requirements and have some planned buffer periods during the project.
Technical	Lack of domain knowledge and technologies causing potential delays	High	Medium	Start exploration of the domain and technologies early in project. Contact supervisors for assistance or ask for domain knowledge experts.
Technical	Lack of experience with developing cloud solutions	Medium	Medium	Be pro-active in learning. Consult supervisor for assistance, refine skills by tutorials, and learn from open source projects.
Technical	Issues with integration of third-party services	Medium	Low	Choose services which have customer support available or an active support community. Contact them and request for early ETA fix. Consult alternative solutions and prepare backups for the worst case.
Technical	Feasibility of the proposed solution	Medium	Medium	Try to develop prototypes (one on-premise and one cloud solution) to detect possible problems so that corrective actions can be taken.
Technical	Unable to fulfil all the low-level requirements.	Medium	Low	Prioritize the requirements. Start with high priority items, complete them, and only then move to lower priority items. Signal to supervisor if a high priority item is blocked and what actions can be taken.
Technical	Not having the platform ready before the scheduled pilot launch	High	Medium	Plan the implementation in steps so that at least an MVP is ready. Next, plan early integrations with other sub-systems in the pilot to make sure that there is enough time to integrate and resolve issues.

■

12. Project Retrospective

This chapter finalizes the report by providing a reflection on the project and the lessons learnt during the project, from the author's perspective.

12.1 Project Reflection

Working at Philips for a duration of 10 months as part of my PDEng graduation project was a valuable and enriching experience. I got introduced to a research environment which was a rather new type of setting that stimulates thinking ahead of the curve by spotting ideas and developing proof of concepts. Working in such an environment was challenging and equally exciting because ideas are initially vague, and they need to be realized in a streamlined way and also gave flexibility to discuss and innovate new ideas.

During the initial phases of the project, I was trying to gain more and more insights to the problems faced in the present customer support experience and the potential ways in which this experience could be digitized and improved. The project goal was to also consider different scenarios and think in the right direction to come up with a solution to incorporate all of these use case scenarios. I had experience working with some networking protocols in the past, but the messaging related protocols were new, and it was an interesting experience to learn about these protocols and methods that enables real-time communication.

During the design phase, I was continuously sharing ideas and prototypes in order to realize the end-to-end system. I divided the system into building blocks and worked on detailed design of individual blocks. The design decisions were discussed, analyzed, and were finalized together with the company supervisors. Somewhere towards the end of the first quarter, I was told that the system I was developing would be part of the pilot execution with hospitals in North America. This was an opportunity for me to collaborate with more people and to see the features of the developed system being used by actual end users.

It was also my first-time hands-on experience developing a cloud-based application and was a steep learning curve in terms of understanding the nitty-gritty details of cloud resources and how they function and interact. The project had activities in all the stages of the software development life cycle process starting from planning till deployment. To conclude, it was a great experience and I am quite satisfied with the way the project was carried out and the end results. The challenges and learnings will be valuable lessons for the upcoming projects.

12.2 Learnings

This section lists some new skills and experiences I gained and refinement of familiar skills in improving my technical, organizational, and personal skills.

Technical skills – From a technical viewpoint, the project had both familiar and unknown domains. I got introduced to the latest messaging protocols, real-time systems, GraphQL query language and APIs, and also got introduced to developing cloud-based applications. I have new stacks added to my technical software development experience.

Organizational and personal skills – I have improved on project planning and management skills as this was an individual project spanning across the entire software development life cycle. I have improved communicating at different abstraction levels based on stakeholders and audience. I have also improved time management and priorities management skills.

Glossary

API	Application Programming Interface is a set of definitions and protocols for developing and integrating software applications
AWS	Amazon Web Services
CDN	Content Delivery Network
CORS	Cross-origin Resource Sharing is a mechanism that allows applications running at one origin to access resources from a different origin
GraphQL	A data query language for APIs and a runtime for executing queries
HTTP	Hypertext Transfer Protocol is stateless application-layer protocol working on the client-server model and is the foundation of World Wide Web
JSON	JavaScript Object Notation
On-premise	Having full control of the infrastructure which is hosted and maintained at company premises
Pilot	An experimental release and testing with a limited set of customers before being introduced more widely
PSG	Project Steering Group
Pub-Sub	Publish-Subscribe
Real-time	Communication type in which users interact almost instantly with negligible latency
REST	Representational State Transfer is an architectural style used for development of web services
SAM	Serverless Application Model
SDK	Software Development Kit is a collection of software tools and services developed for specific platforms
SSL	Secure Sockets Layer adds encryption algorithms and provides communication security during transit
TCP/IP	Transmission Control Protocol and Internet Protocol is a protocol suite that facilitate network communications
Webhook	User defined HTTP callbacks triggered on specific events
WebSocket	Communication protocol that facilitate persistent bi-directional communication
XMPP	Extensible Messaging and Presence Protocol

References

[1] "PDENG PROGRAM, PDEng Software Technology," 2020. [Online]. Available: https://www.tue.nl/en/education/graduate-school/pdeng-software-technology/. [Accessed March 2020].

[2] Philips, "Company," 2020. [Online]. Available: https://www.philips.com/a-w/about/company.html. [Accessed August 2020].

[3] Philips, "About Philips Research," 2020. [Online]. Available: https://www.philips.com/a-w/research/about-philips-research.html. [Accessed August 2020].

[4] Philips, *Scalable Services Delivery, Service Blueprint for Issue Resolution. [Confidential],* 2020.

[5] Philips, *Service Experience Map, Call Handling (Benelux) [Confidential],* 2020.

[6] Scalable Service Delivery, *5-02-2020 lores [Internal],* 2020.

[7] International Organization for Standardization, "Systems and Software Engineering: Systems and Software Quality Requirements and Evaluation (SQuaRE): Guide to SQuaRE. ISO/IEC," 2014.

[8] T. Berners-Lee, R. Fielding and H. Frystyk, "Hypertext Transfer Protocol -- HTTP/1.0," *RFC1945,* 1996.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1," *RFC2616,* 1999.

[10] MDN Contributors, "Connection management in HTTP/1.x," 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Connection_management_in_HTTP_1.x. [Accessed August 2020].

[11] I. Fette and A. Melnikov, "The WebSocket Protocol," *RFC6455,* 2011.

[12] S. Loreto, P. Saint-Andre, S. Salsano and G. Wilkins, "Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP," *RFC6202,* 2011.

[13] P. Saint-Andre, "Extensible Messaging and Presence Protocol (XMPP): Core," *RFC3920,* 2004.

[14] Frozen Mountain, "Do more than just signalling," 2020. [Online]. Available: https://www.frozenmountain.com/products-services/websync/. [Accessed March 2020].

[15] Twilio, "Programmable Chat REST API," 2020. [Online]. Available: https://www.twilio.com/docs/chat/rest. [Accessed March 2020].

[16] PubNub, "PubNub Chat Overview," 2020. [Online]. Available: https://www.pubnub.com/docs/chat/overview. [Accessed March 2020].

[17] Pusher, "Channels overview," 2020. [Online]. Available: https://pusher.com/docs/channels. [Accessed March 2020].

[18] MDN Contributors, "Client-side storage," 2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Client-side_storage. [Accessed August 2020].

[19] MDN Contributors, "IndexedDB API," 2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API. [Accessed August 2020].

[20] M. Thomson, E. Damaggio and B. Raymor, "Generic Event Delivery Using HTTP Push," *RFC8030,* 2016.

[21] AWS, "AWS AppSync," [Online]. Available: https://aws.amazon.com/appsync/.

[22] AWS, "Amazon API Gateway," 2020. [Online]. Available: https://aws.amazon.com/api-gateway/. [Accessed May 2020].

[23] Amazon, "AWS Lambda," 2020. [Online]. Available: https://aws.amazon.com/lambda/. [Accessed July 2020].

[24] "AWS AppSync JavaScript SDK," 2018. [Online]. Available: https://github.com/awslabs/aws-mobile-appsync-sdk-js. [Accessed May 2020].

[25] "Azure Cognitive Services," 2020. [Online]. Available: https://azure.microsoft.com/en-us/services/cognitive-services/. [Accessed August 2020].

[26] GitHub, "AWS Serverless Application Model (AWS SAM)," 2020. [Online]. Available: https://github.com/aws/serverless-application-model. [Accessed July 2020].

Appendix A. GraphQL Schema

```
type Application {
      appId: String!
      appDescription: String!
}

type Group {
      groupId: String!
      appId: String!
      createdAt: AWSTimestamp!
      groupName: String!
      groupDescription: String
      members: [String]!
      state: String!
      members_data: AWSJSON
}

type Identity {
      userId: String!
      appId: String!
      createdAt: AWSTimestamp!
      displayName: String
      userType: String!
}

type Message {
      id: ID
      senderTimeStamp: AWSTimestamp
      serverTimeStamp: AWSTimestamp
      from: String
      to: String
      bodyType: String
      body: String
      numMedia: Int
      mediaTypes: [String]
      mediaUrls: [String]
      numChoices: Int
      choices: [String]
      replyTo: ID
      version: String
      fromChannel: String
      toChannel: String
      _deleted: Boolean
}

type Presence {
      userId: String
      userName: String
      groupId: String
      appId: String
      lastSeenOnline: AWSTimestamp
      userStatus: String
}

type PresenceSet {
      appId: String
      groupId: String
      groupPresence: [Presence]
}

type TypingIndicator {
      groupId: String
      appId: String
      userId: String
```

```
        userName: String
        isTyping: Boolean
}

type Mutation {

        RegisterApplication(appId: String!, appDescription: String!): Application

        CreateIdentity(userId: String!, appId: String!, displayName: String, userType: 
String): Identity

        UpdateIdentity(userId: String!, appId: String!, displayName: String, userType: 
String): Identity

        DeleteIdentity(userId: String!, appId: String!): Identity

        CreateGroup(groupId: String!, appId: String!, groupName: String!): Group

        CloseGroup(groupId: String!, appId: String!): Group

        ReopenGroup(groupId: String!, appId: String!): Group

        AddIdentityToGroup(groupId: String!, appId: String!, identity: String!): Group

        RemoveIdentityFromGroup(groupId: String!, appId: String!, identity: String!): 
Group

        DeleteGroup(groupId: String!, appId: String!): Group

        UpdateGroup(groupId: String!, appId: String!, groupName: String, groupDescrip-
tion: String): Group

        SendMessage(
                id: ID, senderTimeStamp: AWSTimestamp, from: String, to: String, body: 
String, bodyType: String,
                numMedia: Int, mediaTypes: [String], mediaUrls: [String], numChoices: 
Int, choices: [String], replyTo: ID
        ): Message

        DeleteMessage(id: ID, serverTimeStamp: AWSTimestamp, from: String, to: String, 
deletedBy: String): Message

        UpdatePresence(userId: String, appId: String, groupId: String, userName: 
String, userStatus: String): Presence

        UpdateTypingIndicator(userId: String, groupId: String, appId: String, 
userName: String, isTyping: Boolean): TypingIndicator

}

type Query {

        GetIdentities(appId: String!): [Identity]

        GetIdentity(appId: String!, userId: String!): Identity

        GetGroup(groupId: String!, appId: String!): Group

        GetGroups(userId: String!, appId: String!, state: String): [Group]

        GetMessage(id: ID!): Message

        GetMessages(groupId: String, appId: String, startTimeStamp: AWSTimestamp, end-
TimeStamp: AWSTimestamp, lastSync: AWSTimestamp): [Message]

        GetPresence(groupId: String, appId: String): PresenceSet

}
```

```
type Subscription {

       OnMessageReceived(toChannel: String): Message
              @aws_subscribe(mutations: ["SendMessage"])

       OnMessageDeleted(toChannel: String): Message
              @aws_subscribe(mutations: ["DeleteMessage"])

       OnPresenceUpdate(appId: String, groupId: String): Presence
              @aws_subscribe(mutations: ["UpdatePresence"])

       OnTypingIndicator(appId: String, groupId: String): TypingIndicator
              @aws_subscribe(mutations: ["UpdateTypingIndicator"])

}
```

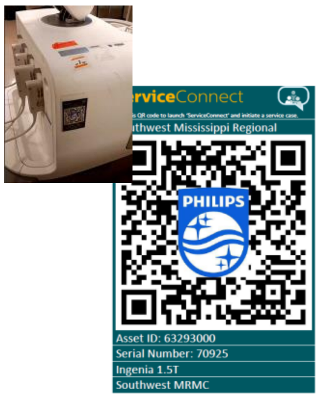
Appendix B. Sample SAM Template

The Serverless Application Model (SAM) is used to create and deploy the messaging platform on AWS. The SAM template is written in YAML and is ~2500 lines containing resource configurations and external references to the source code. The template contains input configurations, 8 DynamoDB references, 3 IAM roles, 32 Lambda permissions, Messaging REST API with 23 endpoints, Webhook REST API with 4 endpoints, WebSocket API with 3 routes, 35 Lambda functions, GraphQL API with GraphQL schema, 19 data sources and 21 resolvers, and stages and deployment configurations. This section contains a snippet of the SAM template depicting one configuration per resource.

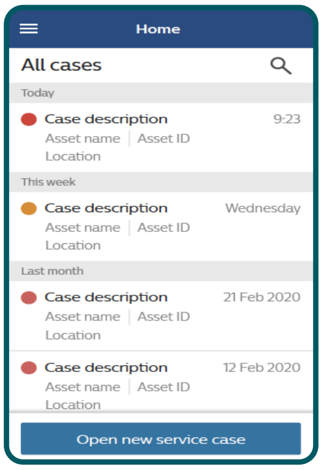
```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Messaging Stack - API GW, Lambda, AppSync Resources

Globals:
  Function:
    Runtime: nodejs12.x
    Timeout: 5
    Handler: index.handler
    Layers:
      - !Ref MessagingLambdaLayerDep
    Environment:
      Variables:
        REGION : !Ref Region

Parameters:
  Env:
    Type: String
    Default: dev
    AllowedValues:
      - dev
      - test
      - prod
    Description: Environment for deployment, test, dev or prod. Default is dev.

  Region:
    Type: String
    Default: us-east-2
    AllowedValues:
      - us-east-2
    Description: Region for AWS Resources deployment

  MessagesTable:
    Type: String
    Default: MSG_BE_Messages-Dev
    AllowedValues:
      - MSG_BE_Messages-Dev
      - MSG_BE_Messages
    Description: Messages DynamoDB tables based on env.

  WSConnectionsTable:
    Type: String
    Default: MSG_BE_WSConnections-Dev
    AllowedValues:
      - MSG_BE_WSConnections-Dev
      - MSG_BE_WSConnections
    Description: Web Socket Connections DynamoDB tables based on env.

  GraphQLKey:
    Type: String
    Description: GraphQL Api Key based on env.

Resources:
  MessagingAppSyncRole:
    Type: "AWS::IAM::Role"
    Properties:
        Description: Role to provide AppSync access to DynamoDB
        Path: "/"
        RoleName: !Sub 'Messaging-AppSync-Role-${Env}'
        ManagedPolicyArns:
          - "arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess"
          - "arn:aws:iam::aws:policy/AWSLambdaInvocation-DynamoDB"
        AssumeRolePolicyDocument:
          Version: "2012-10-17"
          Statement:
            -
              Effect: "Allow"
              Action:
                - "sts:AssumeRole"
              Principal:
                Service:
                  - "appsync.amazonaws.com"

  GetMessagesLambdaPermission:
    Type: "AWS::Lambda::Permission"
    Properties:
      Action: "lambda:InvokeFunction"
      Principal: "apigateway.amazonaws.com"
      FunctionName: !Ref GetMessagesLambda

  SendMessageLambdaPermission:
    Type: "AWS::Lambda::Permission"
    Properties:
      Action: "lambda:InvokeFunction"
      Principal: "apigateway.amazonaws.com"
      FunctionName: !Ref SendMessageLambda

  MessagingRestApi:
    Type: AWS::Serverless::Api
    Properties:
      EndpointConfiguration: REGIONAL
      StageName: !Ref Env
      Name: !Sub 'Messaging-${Env}'
      OpenApiVersion: 3.0.1
      Cors:
        AllowMethods: "'POST, GET, PATCH, DELETE'"
        AllowHeaders:  "'Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Secu-
rity-Token'"
        AllowOrigin: "'*'"
      DefinitionBody:
        swagger: 2.0
        basePath: /prod
        info:
          title: Messaging-Swagger
        schemes:
        - https
        paths:
          /GetMessages:
            get:
              produces:
              - application/json
              responses:
                '200':
                  description: 200 response
                  schema:
                    $ref: "#/definitions/Empty"
                  headers:
                      Access-Control-Allow-Origin:
                        type: string

              x-amazon-apigateway-integration:
                responses:
                  default:
                    statusCode: "200"
                    responseParameters:
                      method.response.header.Access-Control-Allow-Origin: "'*'"
                    responseTemplates:
                      application/json: ""
                uri:
                  Fn::Sub:       "arn:aws:apigateway:${Region}:lambda:path/2015-03-
31/functions/${GetMessagesLambda.Arn}/invocations"

                passthroughBehavior: when_no_templates
                httpMethod: POST
                type: aws
                requestTemplates:
                  application/json: "{\n\
                      \ \"body\" : $input.json('$'),\n\n\
                      \ \"headers\": {\n\
                      \          \"Messaging-Token\":  \"$util.escapeJavaScript($in-
put.params().header.get('Messaging-Token'))\"\n\
                      \   }\n\
                      \ }"

        definitions:
          Empty:
            type: object
            title: Empty Schema

  MessagingWebHookApi:
    Type: AWS::Serverless::Api
    Properties:
      EndpointConfiguration: REGIONAL
      StageName: !Ref Env
      Name: !Sub 'MessagingWebHook-${Env}'
      OpenApiVersion: 3.0.1
      Cors:
        AllowMethods: "'POST, GET, PATCH, DELETE'"
        AllowHeaders:  "'Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz-Secu-
rity-Token'"
        AllowOrigin: "'*'"
      DefinitionBody:
        swagger: 2.0
        basePath: /prod
        info:
          title: WebHook-Swagger
        schemes:
        - https
        paths:
          /RegisterWebHook:
            post:
              produces:
              - application/json
              responses:
                '200':
                  description: 200 response
                  schema:
                    $ref: "#/definitions/Empty"
                  headers:
                      Access-Control-Allow-Origin:
                        type: string

              x-amazon-apigateway-integration:
                responses:
                  default:
                    statusCode: "200"
                    responseParameters:
                      method.response.header.Access-Control-Allow-Origin: "'*'"
                    responseTemplates:
                      application/json: ""
                uri:
                  Fn::Sub:       "arn:aws:apigateway:${Region}:lambda:path/2015-03-
31/functions/${RegisterWebHookLambda.Arn}/invocations"

                passthroughBehavior: when_no_templates
                httpMethod: POST
                type: aws
                requestTemplates:
                  application/json: "{\n\
                      \ \"body\" : $input.json('$'),\n\n\
                      \ \"headers\": {\n\
                      \          \"Messaging-Token\":  \"$util.escapeJavaScript($in-
put.params().header.get('Messaging-Token'))\"\n\
                      \   }\n\
                      \ }"

  GetMessagesLambda:
    Type: AWS::Serverless::Function
    Properties:
      Description: Messaging - Function to retrive all group messages
      FunctionName: !Sub 'GetMessages-${Env}'
      CodeUri: lambdas/GetMessages/
      Role: !GetAtt [MessagingLambdaRole, Arn]
      Environment:
        Variables:
          MESSAGE_DB : !Ref MessagesTable

  SendMessageLambda:
    Type: AWS::Serverless::Function
    Properties:
      Description: Messaging - Function to send a group message
      FunctionName: !Sub 'SendMessage-${Env}'
      CodeUri: lambdas/SendMessage/
      Role: !GetAtt [MessagingLambdaRole, Arn]
      Environment:
        Variables:
          NOTIFICATION_DB: !Ref NotificationsTable
          GRAPHQL_URL: !GetAtt [ MessagingGraphqlApi, GraphQLUrl ]
          GRAPHQL_KEY: !Ref GraphQLKey

  RegisterWebHookLambda:
    Type: AWS::Serverless::Function
    Properties:
      Description: Messaging - Function to register a new web hook
      FunctionName: !Sub 'RegisterWebHook-${Env}'
      CodeUri: lambdas/RegisterWebHook/
      Role: !GetAtt [MessagingLambdaRole, Arn]
      Environment:
        Variables:
          WEBHOOK_DB : !Ref WebHooksTable

  MessagingLambdaLayerDep:
    Type: AWS::Serverless::LayerVersion
    Properties:
        LayerName: messaging-dependencies
        Description: Auth dependencies and helper functions for messaging
        ContentUri: dependencies/
        RetentionPolicy: Retain
        # RetentionPolicy: Delete

  MessagingGraphqlApi:
    Type: AWS::AppSync::GraphQLApi
    Properties:
      AuthenticationType: API_KEY
      Name: !Sub 'Messaging-${Env}'

  MessagingGraphqlSchema:
    Type: AWS::AppSync::GraphQLSchema
    Properties:
      ApiId: !GetAtt [MessagingGraphqlApi, ApiId]
      DefinitionS3Location: graphql/schema.graphql

  MessagingDSAddIdentityToGroup:
    Type: AWS::AppSync::DataSource
    Properties:
      ApiId: !GetAtt [MessagingGraphqlApi, ApiId]
      LambdaConfig:
        LambdaFunctionArn: !GetAtt [ AddIdentityToGroupLambda, Arn ]
      Name: AddIdentityToGroupSource
      ServiceRoleArn: !GetAtt [MessagingAppSyncRole, Arn]
      Type: AWS_LAMBDA

  MessagingDSMessagesTable:
    Type: AWS::AppSync::DataSource
    Properties:
      ApiId: !GetAtt [MessagingGraphqlApi, ApiId]
      DynamoDBConfig:
        AwsRegion: !Ref Region
        DeltaSyncConfig:
          BaseTableTTL: 43200
          DeltaSyncTableName: !Ref DeltaMessagesTable
          DeltaSyncTableTTL: 1440
        TableName: !Ref MessagesTable
        # UseCallerCredentials: Boolean
        Versioned: TRUE
      Name: MessagesTableSource
      ServiceRoleArn: !GetAtt [MessagingAppSyncRole, Arn]
      Type: AMAZON_DYNAMODB

  MessagingRslAddIdentityToGroup:
    Type: AWS::AppSync::Resolver
    Properties:
      ApiId: !GetAtt [MessagingGraphqlApi, ApiId]
      DataSourceName: !GetAtt [ MessagingDSAddIdentityToGroup, Name ]
      FieldName: AddIdentityToGroup
      RequestMappingTemplate:  '{"version":  "2017-02-28",  "operation":  "Invoke",
"payload":    {"Source":    $utils.toJson("AWSAppSync"),    "body":    {"appId":
$utils.toJson($ctx.args.appId), "groupId": $utils.toJson($ctx.args.groupId), "iden-
tity": $utils.toJson($ctx.args.identity)}, "headers": {"Messaging-Token": ""} }}'
      ResponseMappingTemplate: $util.toJson($ctx.result)
      TypeName: Mutation

  MessagingRslGetMessages:
    Type: AWS::AppSync::Resolver
    Properties:
      ApiId: !GetAtt [MessagingGraphqlApi, ApiId]
      DataSourceName: !GetAtt [ MessagingDSMessagesTable, Name ]
      FieldName: GetMessages
      RequestMappingTemplateS3Location: graphql/GetMessagesReqResolver
      ResponseMappingTemplate: $util.toJson($ctx.result.items)
      TypeName: Query

  MessagingRslSendMessage:
    Type: AWS::AppSync::Resolver
    Properties:
      ApiId: !GetAtt [MessagingGraphqlApi, ApiId]
      DataSourceName: !GetAtt [ MessagingDSMessagesTable, Name ]
      FieldName: SendMessage
      RequestMappingTemplateS3Location: graphql/SendMessageReqResolver
      ResponseMappingTemplate: $util.toJson($ctx.result)
      TypeName: Mutation
      SyncConfig:
        ConflictDetection: VERSION
        ConflictHandler: OPTIMISTIC_CONCURRENCY

Outputs:
  RESTApi:
    Description: REST URL for Messaging APIs
    Value:      !Sub      "https://${MessagingRestApi}.execute-api.${Region}.amazo-
naws.com/${Env}"

  WebHooksApi:
    Description: REST URL for Web Hooks APIs
    Value:     !Sub     "https://${MessagingWebHookApi}.execute-api.${Region}.amazo-
naws.com/${Env}"

  GraphQLApi:
    Description: GraphQL URL for Messaging APIs
    Value: !GetAtt [ MessagingGraphqlApi, GraphQLUrl ]
```

Appendix C. *Service Connect* Screenshots

Service Connect is a web application developed for pilot sites in order to demonstrate digitization of remote communication capabilities. The application is launched by scanning a device QR code or when an invitation link is opened. This application uses the messaging platform for text and multimedia communication. A few screenshots of the application running in a mobile browser are shown below.



Scan device QR code (launches the application)



Home

All cases

Today

Case description 9:23
Asset name | Asset ID
Location

This week

Case description Wednesday
Asset name | Asset ID
Location

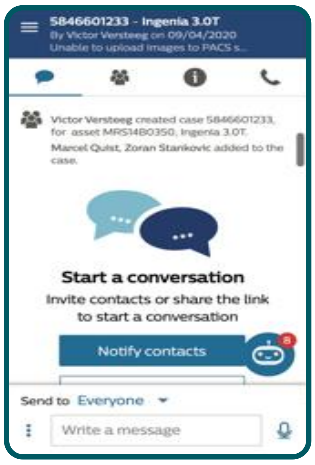
Last month

Case description 21 Feb 2020
Asset name | Asset ID
Location

Case description 12 Feb 2020
Asset name | Asset ID
Location

Open new service case

Retrieve previous cases and conversation history



5846601233 - Ingenia 3.0T
By Victor Versteeg on 09/04/2020
Unable to upload images to PACS s...

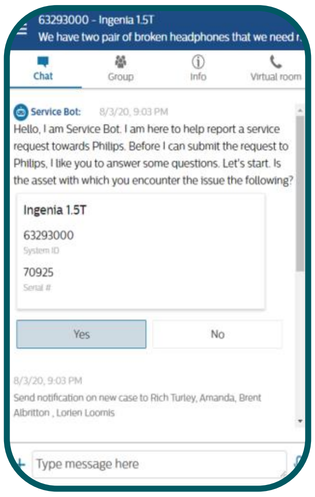
Victor Versteeg created case 5846601233, for asset MRS14B0350, Ingenia 3.0T. Marcel Quist, Zoran Stankovic added to the case.

Start a conversation
Invite contacts or share the link to start a conversation

Notify contacts

Send to Everyone
Write a message

Create a new service case. SMS or Email notification is sent to associated members



63293000 - Ingenia 1.5T
We have two pair of broken headphones that we need r...

Chat Group Info Virtual room

Service Bot: 8/3/20, 9:03 PM
Hello, I am Service Bot. I am here to help report a service request towards Philips. Before I can submit the request to Philips, I like you to answer some questions. Let's start. Is the asset with which you encounter the issue the following?

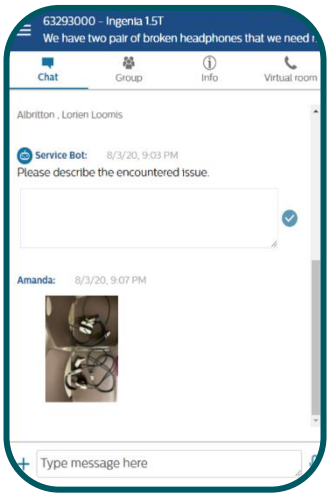
Ingenia 1.5T
63293000
System ID
70925
Serial #

Yes No

8/3/20, 9:03 PM
Send notification on new case to Rich Turley, Amanda, Brent Albritton , Lorien Loomis

Type message here

Conversation with chabot



63293000 - Ingenia 1.5T
We have two pair of broken headphones that we need r...

Chat Group Info Virtual room

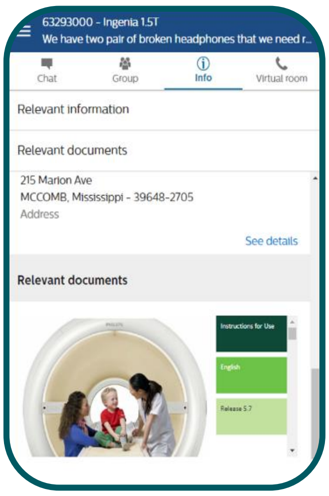
Albritton , Lorien Loomis

Service Bot: 8/3/20, 9:03 PM
Please describe the encountered issue.

Amanda: 8/3/20, 9:07 PM

Type message here

Text and Multimedia Messaging



63293000 - Ingenia 1.5T
We have two pair of broken headphones that we need r...

Chat Group Info Virtual room

Relevant information

Relevant documents

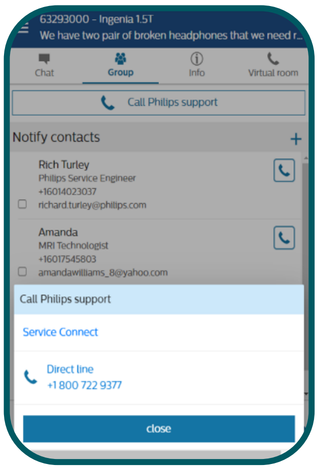
215 Marion Ave
MCCOMB, Mississippi - 39648-2705
Address

See details

Relevant documents

Instructions for Use
English
Release 5.7

Access to relevant documents



63293000 - Ingenia 1.5T
We have two pair of broken headphones that we need r...

Chat Group Info Virtual room

Call Philips support

Notify contacts

Rich Turley
Philips Service Engineer
+16014023037
richard.turley@philips.com

Amanda
MRI Technologist
+16017545803
amandawilliams_8@yahoo.com

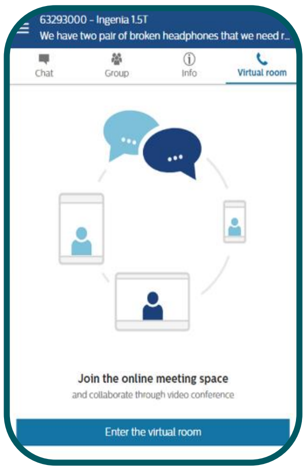
Call Philips support

Service Connect

Direct line
+1 800 722 9377

close

Request for direct call



63293000 - Ingenia 1.5T
We have two pair of broken headphones that we need r...

Chat Group Info Virtual room

Join the online meeting space
and collaborate through video conference

Enter the virtual room

Request for video conferencing

About the Author

Priyanka Patel received her bachelor's degree in Electrical and Electronics Engineering in 2014 from R.V. College of Engineering, India. Her thesis was in the networking research domain in collaboration with Cisco Systems. After graduation, she worked at Cisco Systems as a full-stack software engineer and a backend engineer between 2014 and 2018. She was part of the Software Technology PDEng program, 2018-2020, at Eindhoven University of Technology. During her graduation project she worked at Philips Research.