Examensarbete

HÖGSKOLAN
I SKÖVDE

DATA COMMUNICATION FOR THE AUTOMOTIVE INDUSTRY: REST API VS MESSAGE QUEUE

Bachelor's degree in informatics
30 ECTS
William Aspensten

Supervisor: Yacine Atif
Examiner: Henrik Gustavsson

Summary

Sending messages between systems is today a necessity in the production industry. The message communication needs to be designed and managed. Different architectural designs are to be considered for a communication system. Designing and implementing a communication system can be heavy for both time and cost. It is therefore important that the right architecture is selected early in development.

Two of the aspects that are important to consider are latency and throughput. The goal of this thesis is to give insight into these aspects using an experiment that will measure transfer speed and message loss. The conclusion of the thesis is that Swagger API has faster transfer speed but also saw more message loss than RabbitMQ. An Anova test proves that the difference is statistically significant. This thesis only experiments with the tools Rabbit MQ and Swagger Rest API; future work could expand on this work by changing tools.

Nyckelord: Data communication, Message Queue, Rest API, Performance, RabbitMQ, Swagger

Table of Contents

1 Introduction 1

2 Background 2

2.1 REST API 2

2.1.1 Swagger 2

2.2 Message Queue 3

2.2.1 RabbitMQ 4

2.3 Grafana K6 4

2.4 Dotnet and Razor Pages 5

2.5 The automotive industry 6

3 Problem 7

3.1 Problem statement 8

3.2 Research questions 8

3.3 Hypothesis 8

4 Method 9

4.1 Alternative Methods 9

4.1.1 Case Study 9

4.1.2 Survey 10

4.1.3 Conclusion 10

4.2 Threats to validity 10

4.2.1 Construct validity 10

4.2.2 External validity 11

4.2.3 Internal validity 11

4.2.4 Conclusion validity 11

5 Implementation 12

5.1 Literature Study 12

5.2 K6 13

5.2.1 Progression 17

5.3 Swagger - Rest API 18

5.3.1 Progression 22

5.4 RabbitMQ - Message Queue 22

5.4.1 Progression 24

5.5 Razor dotnet Application 24

5.5.1 Progression 27

6 Pilot study 28

6.1 Tooling 28

6.2 Scripts 28

6.3 Results 29

6.4 Pilot study analysis 31

6.5 Conclusion 32

7 Evaluation 34

7.1 Preparation 34

7.2 Results 35

7.2.1 Latency 35

7.2.2 Throughput 43
7.3 Analysis 43
7.3.1 Latency 44
7.3.2 Throughput 45
7.4 Conclusions 45
8 Concluding Remarks 47
8.1 Summary 47
8.2 Discussion 47
8.2.1 Ethics 48
8.3 Future work 49
9 References 51

1 Introduction

Communication between systems when transferring data can be complex to maintain and manage. Data needs to be reliable, have scalability and have as low latency as possible. This creates challenges for the automotive industry where communication with different systems is highly dependent on being able to send messages in real-time. The choice of data transfer architecture becomes an important factor and can affect both the performance and maintainability. Changing communication architecture for industrial systems can require long redevelopment and restructuring of existing applications. That is why having the right communication architecture early in development has a high importance. Many different factors can have an impact on what communication architecture is most suitable for a production system.

Commonly used solutions for data transferring are Message Queues and REST API:s. Message queue systems provide decoupled and asynchronous communication. This allows the systems to continue operating and sending messages when parts or one of the systems are unavailable. Messages are queued and processed independently between services and can be retrieved when a system is available. REST API:s instead uses synchronous request-response communication where data is transferred directly between systems over HTTP protocols. Communication is direct between systems where a client sends a request and waits for a response before continuing execution. Message queues are built on reliability and scalability while RESP API:s is simple to implement and may provide faster direct communication. Both communication architectures have some clear advantages and disadvantages in how they function. What is not clear from the architectural standpoint is how well they perform.

Comparison between communication architectures have been done previously, often focusing on scalability and reliability using throughput and latency as benchmark to create measurement series. Common research has looked at different REST API:s versus each other and in some cases a Message Queue tool versus another Message Queue tool. There are few studies looking at the two communication paradigms versus each other and even fewer within an industrial automotive environment. There is some prior research like the one made by (Hong et al., 2018) that benchmarks and compares Message queue versus Rest API. The research in this thesis aims to expand on their research and have the automotive industry in focus. This study seeks to contribute to the message communication area by examining the performance and stability between the Message Queue RabbitMQ and the REST API Swagger. The goal of this thesis is to provide insight into what implications Message Queue solutions and REST API solutions have on the performance of data transferring. The study is based on the hypothesis that switching to REST API from Message Queue should improve performance in terms of transferring speed. An empirical study is conducted in the form of an experiment. This experiment will benchmark two communication solutions by using measurement points from a measurement series. The goal of the study is to help developers in the selection of a communication architecture for a system. This will not give a definitive answer in what communication architecture is the best but will instead give insight into how the two architectures affect the latency and throughput for communication messages.

2 Background

Message management and data handling involves the methods and technologies used to transfer, process, and store information between systems. Modern systems, applications and services need to exchange data continuously. The data exchange defines the communication between systems. Managing communication between systems has high importance, badly managed communication can lead to bad reliability and performance. To handle system communication there are several approaches, two of the most common ones are synchronous communication through REST API:s and asynchronous communication through Message Queues. The two approaches differ in how they function and exchange data. This could lead to differences in performance for the communication systems regarding latency and throughput.

2.1 REST API

A REST API is a communication architecture used for transferring data between systems over HTTP protocols. The data is transferred using predefined endpoints where clients can send requests and responses. API protocols explained by (Niswar et al., 2024) are a set of rules that facilitate the communication and interaction between systems. REST API:s uses synchronous communication where a client sends a request and waits for a response from a server, if the server does not answer the execution disrupts. Because of the synchronous model REST API:s is relatively simple to implement. The communication architecture REST API is commonly used in web applications and cloud services as it is a secure and simple way of transferring data.

REST (Representational State Transfer) API uses HTTP operations which (Poulter et al., 2015) explain to be POST, GET, PUT and DELETE. By using these operations clients can consume messages. The client can be a web application receiving a message from a service. The REST API architecture and synchronous communication have some drawbacks. Because of the synchronous communication method, performance can be affected in the case a server is overloaded or if the network latency is high. REST API also has a tighter coupling between systems as both the client and the server need to be available for the communication to be able to work.

2.1.1 Swagger

Is a framework used for designing, testing and documenting REST API:s. Swagger uses the OpenAPI specification to describe how to handle endpoints, requests and responses. Swagger as a framework simplifies the development of REST API architectures and improves the communication for developers to understand the system. Swagger makes it easier to see the result by incorporating interfaces that the developers can interact with. The interface is one of the main features with Swagger and allows developers to view available API endpoints and test requests can be done directly through a web browser.

(Dos Santos et al., 2020) explains how Swagger was renamed to OpenAPI specification in 2017 with version 3.0. They also explain how Swagger is a framework for documenting REST API:s and can be written JSON or YAML. Swagger acts as a layer for documenting and managing API communication but does not define how the systems should communicate over HTTP. That is done by the OpenAPI specification. Because Swagger only works as a layer for the REST API it should not have any impact on the performance. Swagger will instead be used to

understand the endpoints and test communication both from a browser but also from an external tool.

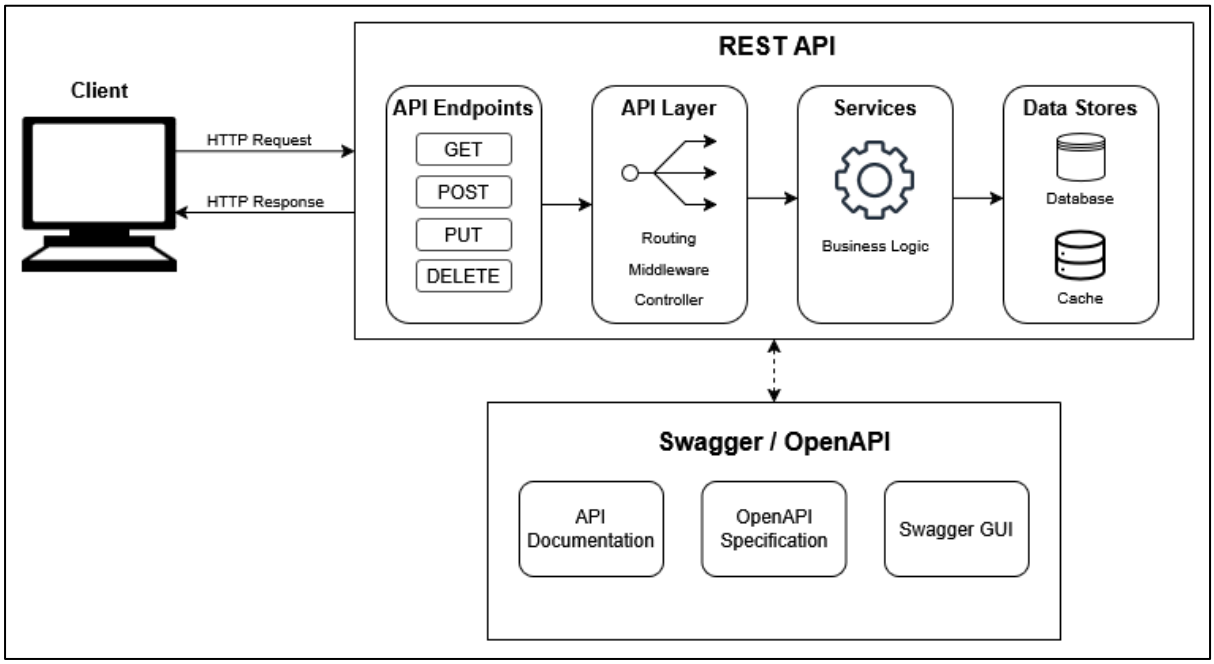

Figure 1 Showing how a Rest API works and the connection to Swagger

A client sends a request to the REST API through HTTP. This request is handled by the REST API using Endpoints, routing, middleware and controllers. A service layer implements business logic and communicates with storage like a database or internal memory. Swagger/OpenAPI provides API documentation, specification, and an interactive Swagger GUI. (Figure 1)

2.2 Message Queue

A Message Queue is a communication method where messages are stored in a queue before being processed by another system or service. Message queues use asynchronous communication where the sender and receiver do not interact with each other. The messages are temporarily stored inside the Message queue until the receiving system is ready to process them. This communication method decouples systems from each other and lets operations continue even if parts of an application are currently unavailable. This means that outside forces have a low impact on the Message Queues performance. Two of the strengths of using message queues for a production system according to (Ionescu, 2015) is the asynchronous architecture where operation can continue even if a failure occurs and the message queue ability to scale.

An advantage of Message Queues is how they reduce direct dependencies between systems, while this is great asynchronous communications also have some drawbacks. The system design has a higher complexity than REST API and can be harder to monitor and debug. Message Queues may also introduce latency since messages need to be processed through queues before reaching the receiving system.

2.2.1 RabbitMQ

There exist many different versions of Message Queues in the world, and for this thesis the Message Queue tool RabbitMQ will be used. RabbitMQ receives a data message from one system and adds the message into an internal queue. A queue can store multiple messages and can use certain rules like priority prioritization. RabbitMQ holds the messages until another system sends a retrieval request, when the request comes in RabbitMQ will route the message to the receiving system. When RabbitMQ sends the messages to the receiving system the messages are also removed from the queue. It is important that the receiving system can consume the messages from RabbitMQ, this is one of the reasons that makes implementing a Message Queue more complex than a REST API solution.

RabbitMQ uses the protocol AMQP (advanced message queuing protocol). According to (Dobbelaere and Sheykh Esmaili, 2017) AMQP was created to gain *interoperability of different asynchronous messaging middlewares*. The approach becomes modular where brokering tasks can be divided between message queues andGr exchanges with systems. AMQP defines the rules for how systems can communicate with the Message Queue. The rules specify how messages should be formatted and how the queues and routing should work. AMQP originated from JPMorgan in 2023 as stated by (Dobbelaere and Sheykh Esmaili, 2017) who also explains that RabbitMQ was developed in Erlang.

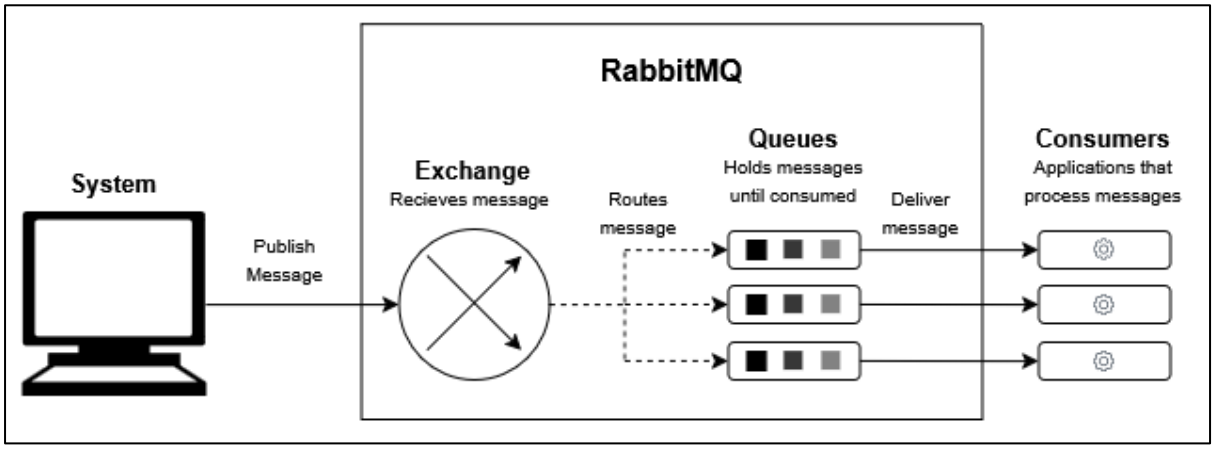

Figure 2 Showing a message exchange using RabbitMQ

A system publishes a message to an exchange, which routes them to one or multiple queues. The queues temporarily hold messages until they are to be. Consumers then retrieve and process the messages independently. (Figure 2)

2.3 Grafana K6

Message Queue and Rest API:s does not generate their own messages but instead receive them from somewhere. The sender of the message can for example be a system or a service. Instead of creating an entire new system to send messages for testing and evaluating Message Queues and Rest API:s one can use already existing tools like Grafana K6. This tool is open-source and is designed to generate and execute scripts. The tool was originally developed as K6 and later was incorporated into the Grafana ecosystem. Grafana K6 uses JavaScript or TypeScript and allows a user to define request rates, payload contents, execution duration and can also be

used to log message transactions. Grafana K6 can send JSON objects using HTTP requests to an endpoint which a Message Queue or a Rest API can listen to.

One of the key characteristics of Grafana K6 is its ability to generate large volumes of HTTP requests while also maintaining a low resource consumption. Like Grafana K6 other tools like Apache JMeter or Gatling could be used to send data messages but according to (Pinyagin and Sadovykh, 2025) Grafana K6 consistently outperforms the other available performance testing tools. In this thesis Grafana K6 will be used to send messages to both RabbitMQ but also Swagger API, the messages sent will be over HTTP and will involve large chunks of JSON objects. Because of this Grafana K6 is the most appropriate for the experiment that will be performed as it will have the lowest impact on the performance result for the two communication architectures. Recent research like the one made by (Sedaghatbaf et al. 2011) discusses the advantages of using automated workload generation. Using a tool like Grafana K6 can improve the reproducibility of the experiment and removes the need of creating test scenarios.

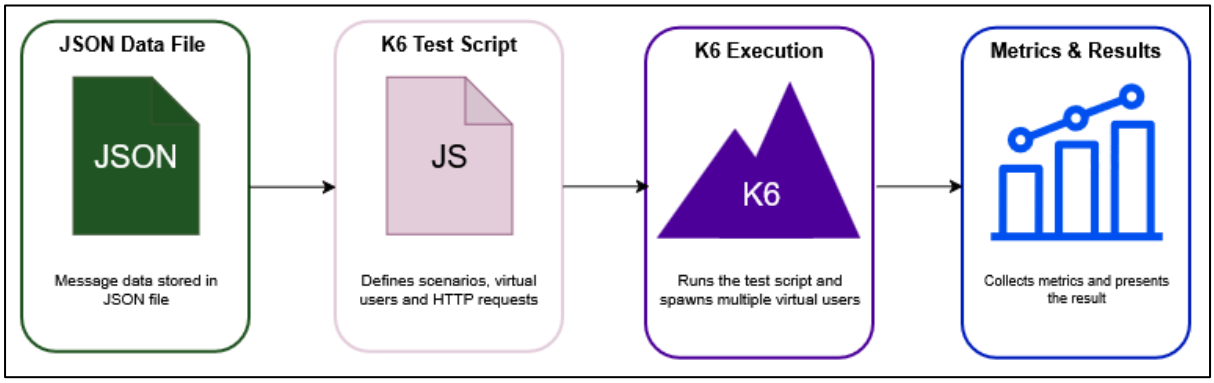

Figure 3 A K6 transaction from a JSON file to a metric result

Message data is stored in a JSON file and used by K6 test script. The script defines test scenarios; how many virtual users should be used and how HTTP request should work. K6 runs the test and collects metric data to present how the execution went. (Figure 3)

2.4 Dotnet and Razor Pages

Dotnet, also known as ASP.NET Core, is a web application framework developed by Microsoft. Dotnet is used to build web applications and services, the primarily programming language for Dotnet is C#. Within Dotnet Razor Pages provides a development model that combines user interface with server logic. This framework was introduced to simplify web development by reducing the complexity for a controller-based architecture while also separating presentation and business logic. Razor pages use a page focused architecture where each page has a Page Model class that is responsible for the application logic and a markup class that uses HTML elements to create the interface. In this thesis a Dotnet application will be used to receive messages from RabbitMQ and Swagger API.

One of the advantages with Dotnet is its performance and scalability. The framework was designed to be lightweight, modular and can handle large amounts of requests while still having a low resource consumption. This makes Dotnet a valuable option for systems and web services where responsiveness and efficient resource utilization are important aspects. In this thesis the performance of the two communication architectures Message Queue and Rest API will be tested, with this in mind the tools sending and receiving the messages should have as

low impact on the result as possible. Therefore, it is important to select an appropriate framework to receive the messages, this is demonstrated in the research by (Stojanovski et al., 2012). Dotnet also has a large ecosystem with libraries, development tools and community resources. The framework can easily be integrated with messaging technologies like Message Queues and Rest API:s. All these aspects make Dotnet a suitable option for this thesis.

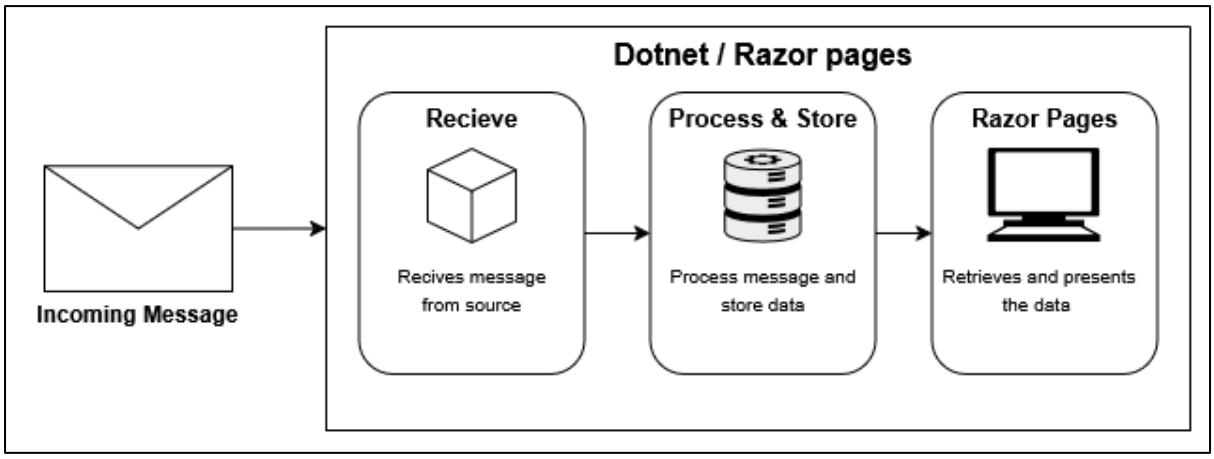

Figure 4 Shows how dotnet handles a message

An incoming message is received by the dotnet application from a Message Queue or a REST API. The message data is being processed and stored in some type of storage like a database or internal memory. The Razor page retrieves the data and presents it. (Figure 4)

2.5 The automotive industry

Having efficient data handling is important for the industrial environments where systems process large amounts of information in real time. If communication has a failure when transferring data, the production may be impacted. When a production relies on data transferring being reliable and fast the cost for failure can be expansive. It is important to choose an appropriate communication architecture that can solve the needs for the production, both the desired functionalities and the performance.

To be market leading an industry needs to be innovating and embracing new technologies. This philosophy is true both for the products being created and the production process that creates the product. Customers today expect new vehicles to have robust technology with multiple functionalities. For the production process IT systems serve an important factor in how well the production can run. A firm according to (Svahn et al., 2017) should focus on innovation and development of new technologies to expand their companies. To grow an automotive company and keep up with future demands the IT infrastructure constantly must evolve.

In the automotive industry today IT systems are used for the entire production, all from inventory balance to transportation of material. If one system handles the material storage while another handles the transportation the two systems need to communicate with each other when material should be transferred from one location to another. If the communication system has bad latency the package could arrive at its delivery location too late, and if the throughput is bad the message data may not go through and production stands still without material to continue their work.

3 Problem

The problem is a lack of data and therefore knowledge when it comes to understanding what communication paradigm should be used when designing an architectural message structure between two systems. This creates a problem for developers and architects in the automotive industry when multiple different systems in their factories are used and data needs to be transferred from one system to another. The two most common communication paradigms used in the automotive industry are asynchronous message brokers and synchronous HTTP request/response using REST. Previous research like the one made by (Fernandes et al., 2013) has investigated the performance differences between Web services using REST API and Message queues using AMQP. In their research they found that Message queues performed better when large number of messages were exchanged. This indicates that the performance between the communication architectures can depend on the amount of data and messages being sent to a system rather than on architecture consistently outperforming the other. Making the choice of what type of communication architecture should be used comes down to what is of importance when sending data and what resources and tools are available for data transferring.

Speed in the form of low latency has high importance in an industry, the longer the data takes to go from one system to another the more potential loss can occur in production. The same is true for message loss, the communication needs to be reliable and have a good throughput, so the correct data gets through. If data is missing or a message does not get through, production may slow down while waiting for material to get to the correct place. Research around communication for latency, throughput and reliability has been done by (Sarasola et al., 2024) where they evaluated protocols under different message sizes. In their result they discovered that performance could differ for different communication system depending on the communication workload and system conditions.

In the research done by (Hong et al., 2018) the message queue tool RabbitMQ is compared to the communication type REST API. In their research they explain the differences between HTTP communication using REST and Advanced Message Queuing Protocol (AMQP) and how they work. Their experiment tests response speed by checking the time it takes for a user to receive a response message after sending a request. They repeat this experiment six times using different numbers of users that send requests. The conclusion is that the REST API is faster when there are a lower number of users sending requests but slower when the number of users exceeds 250. They also mention in the result that the error information rate increases for REST API up to 28,41% when there are 300 users. A problem with this is that they never describe what they mean by error information rate or how they measure this. The study tries to answer what communication paradigm is faster but never specifies what type of REST API is being used. The message queue is mentioned to be RabbitMQ but there is no transparency for the REST API, did they create their own or use one of the famous tools like Express or Swagger. The method also introduces an Event API between the Message Queue/REST API and the database. This could potentially create internal validity problems as the cause of change can be because of the interaction with the Event API.

This thesis has full transparency for what communication tools will be used. The measurement types for latency and throughput are also described and explained to avoid confusion and show what is measured and why. The method for this thesis will not use any extra communication processes like the Event API that (Hong et al., 2018) is using instead of creating a simple

method so the communication paradigms will be in focus. The goal of this thesis and its research will be to show what communication paradigm has the lowest latency in the form of speed it takes to send and retrieve a message while also look at the throughput to see if any of the communication paradigms has a higher chance of losing data when sending many messages. Both measurement types will then be considered when discussing the positives and negatives of changing the communication architecture for a system from Message Queue to Rest API.

3.1 Problem statement

What is the difference in performance between queue-based communication using IBM WebSphere MQ and API-based communication using Swagger?

3.2 Research questions

RQ1: How does the time performance differ between REST API and Message Queue when retrieving and sending a message?

RQ2 :How does the reliability in message loss differ between REST API and Message Queue when sending multiple messages?

3.3 Hypothesis

H1: By changing solutions for data transferring from Message queues to REST API the performance should improve in transferring speed.

H0: No performance boost comes by changing from Message queue to REST API.

4 Method

The research will try to solve a knowledge problem for different communication architectures in Information technology. A mix of multiple research types will be used, such as for example descriptive, applied and empirical research. The research method will fall under controlled experiment with an experimental research approach. The experiment will use two different communication paradigms to send messages of data from a measurement tool to a logistic application. In the chapter Microservice web application by (Hong et al., 2018) they describe a method where a message is being sent using either a message queue in the form of RabbitMQ or a REST API. In their method the message is being sent through a web browser to one of the communication tools and then stored into a database. The method in this thesis experiment will be of some similarity. A message will be handled by either a REST API using swagger or by the message queue RabbitMQ. Instead of using a browser to send the message the measurement tool K6 will send the message to the communication tools. The message will then be retrieved by a Razor dotnet application.

This experiment will test and benchmark latency. This will be done by measuring the time a message is received by one of the communication tools and then sent to an application. This measures the system's responsiveness and is instrumental in how both a user can perceive a system but also in how effective the system is. The experiment will also check the throughput of the communication tools by looking at loss of data. By sending a large chunk of messages in a short time will messages get stuck or data in the messages be missing. This will be the same measure stocks that (Maharjan et al., 2023) describes in their methodology chapter. They describe how high latency can feel sluggish while low latency can enhance a user's experience and throughput is about how many messages that a system can handle in a specified period.

The method chosen for the experiment lies mostly in the available tools and data that is accessed through the company the thesis is collaborated with. A message process exists involving messages with material transport orders, same process will be used for the thesis experiment. The things that will be built are the K6 measurement tool, a REST API using Swagger as a framework that will be connected to the Razor application. All the chosen tools are open source; this improves the repeatability of the experiment. As this experiment needs to benchmark some type of measurement, latency is a good choice to get measure points that can compare the two different communication paradigms. Throughput also shows how stable the systems are, this will not be benchmarked in the same way as the latency but will be used in the analyzing as a counterpart to the efficiency of the communication paradigms. Things that will not be measured or considered for the experiment are CPU utilization, memory usage or server capacity. The experiment does not consider aspects like architectural advantages but only focuses on the latency and throughput. This experiment will only test the communication tools RabbitMQ and Swagger Rest API and do not cover the entire extent of the two communication paradigms. Therefore, more research needs to be done to get a definitive answer on what communication paradigm has the best efficiency and stability.

4.1 Alternative Methods

4.1.1 Case Study

As an alternative to the experimental study, a case study could be conducted. In the case study two artifacts are implemented and evaluated against each other in an automotive industry environment. One would use a Message Queue while the other would use REST API. Data could be collected through developer feedback, user observations and performance measurements for latency and throughput. The study would cover both technical and practical

performance. A case study could provide us with more data than an experimental study and would give insight into how a user perceives the communication systems. A case study has some limitations; the collected data will be subjective and more difficult to analyze compared to experimental measurements. If the study was made in a realistic environment external factor may influence the results and potentially damage the validity of the study. For this thesis it would be difficult to gather user observations as the communication tools run in the background and no user interaction is necessarily needed.

4.1.2 Survey

Another alternative would be doing a survey. Two artifacts are created and distributed to survey contenders. They would get to interact with the artifacts and submit a form with questions detailing how they perceived the artifacts. Using a survey could lead to insight regarding how a user perceives the responsiveness, ease of integration and if latency is noticeable during data transfers. A survey can have some drawbacks. It can be difficult to gather enough human subjects to get a big dataset that can be analyzed. The results are also subjective and may be influenced by external factors like participating in experience and how the survey questions are formulated. As described in the case study section the communication tools work as background processes and adding user interactions could potentially change the result.

4.1.3 Conclusion

Both a case study and a survey would be interesting approaches to evaluate the user experience. An experimental method was chosen because it has the highest potential to answer the hypothesis and the research questions. What the thesis tries to answer is the latency in raw speed for the two communication systems. By using an experimental method, we can gather large amounts of data that can be analyzed with statistical significance. The experiment can be controlled, and the result will be both objective and clear.

4.2 Threats to validity

The handling of validity threats is connected to research ethics and good scientific practice. If threats to validity are not addressed properly the experiment risks producing a false result and conclusion regarding latency and throughput for the communication architectures. It is important to have transparency for the experiment, implementation and the analysis to ensure credibility and ethical responsibility. To achieve reproducibility and enable external reviews all source code developed for this experiment will be published on GitHub and all the tools used are open source. This will allow other researchers to inspect the implementation, build and test it and use it as inspiration and a basis for future work.

4.2.1 Construct validity

It is important that the measurement represents the actual measure concept. This thesis presents two research questions that need to be answered by the experiment. The measure for latency must represent the concept of end-to-end system latency. This is done by using timestamps for when the message is sent and a response has been received. The throughput represents the concept of message loss. All messages must be checked so the same data that was sent is the same that is retrieved by the logistic application. To make sure construct validity is not broken the same message data will be used for both treatments and the

definition of latency and throughput need to be consistent in the thesis all the way from the background to the analysis.

4.2.2 External validity

This thesis will measure two different communication paradigms, one using the message queue IBM WebSphere and another using the REST API swagger. External validity refers to how well a result can be generalized beyond the specific experiment and its test environment. Can the result be applied to real production usage and what happens when components or tools are changed. There is many different Message queues and REST API: s and changing them could influence the result for both latency and throughput. To make sure external validity is kept it is important to be transparent with the design method and what tools this experiment is applied to. This experiment will also be done using production systems like the IBM WebSphere MQ and the logistic application.

4.2.3 Internal validity

For the experiment it is important that we measure the treatments correctly. That the cause construct and in this case the independent variable is what is causing a difference in latency and throughput, not any other known or unknown variable. Factors that could lead to problems with internal validity would be CPU load, network conditions, different hardware and message sizes. The artifacts for the message queue and REST API need to be similar in their functionality to make sure the measurements have no impact from different workloads because of faulty code and message handling. The same hardware and messages need to be used for both treatments in the experiment, and the CPU load and network need to be monitored during the trials.

4.2.4 Conclusion validity

To make sure a correct conclusion from an experiment can be taken, statistical significance is crucial. Without any statistics for the result, both the analysis and conclusion can be faulty and have biases. For this thesis the lower latency could not be proven without any statistical significance. Conclusion validity will be ensured by doing statistical tests on the data.

5 Implementation

To conduct this experiment four artifacts have been created. Two of the artifacts use Grafana K6 to send messages, one to the API endpoint and one to the MQ endpoint. One artifact using Swagger API, the API receives and sends messages and another artifact using Razor dotnet that receives the messages from both the API and the MQ.

All the artifacts have been published on GitHub and are open to the public (Thesis GitHub, 2026)[1].

5.1 Literature Study

For this thesis a variant of different literature studies and documentations have been used, both in setting up the method but also how to create the implementation. The method for this thesis is based upon the method used by (Hong et al., 2018). Some differences exist between the two methods like the use of an extra API before RabbitMQ and the use of a database. Both methods follow a similar data flow and benchmarks RabbitMQ versus a REST API. The same dependent variable in the form of latency will be measured. For most of the systems used some prior knowledge exists which has been used in creating the implementation. Most of the knowledge gaps have been filled in by documentation from the web. The different tools used all have their own documentation page but also communities that update the information. As all tools used in this thesis are open source, the community keeps the documentation updated. Both the tools and the people at Volvo have also contributed to the success of the implementation.

K6: To use the K6 library it needs to be installed on a client, this is explained in the documentation at (Grafana 2026). The client used for the implementation uses the operating system windows. In their documentation they also describe how to execute a script and how to write one. For example, in the section HTTP Request they explain how to create a basic HTTP request script. This was used as the basis for the first K6 script built. In the section API load testing they explain how to post from K6 to an endpoint.

Razor dotnet application: To use dotnet it first needs to be installed on a client. Downloading and installation guide can be found at (Microsoft, 2026) download section. On (Microsoft, 2026) learn page one can find instructions and documentation for how dotnet works and how to create a razor page. The YouTube page owned by dotnet (Dotnet YouTube, 2026) has been useful to start the creation of the Razor artifact. The readings by (Price, 2025) have been helpful to get a deeper understanding of dotnet, MVC structure and Web API:s.

Swagger - REST API: As Swagger is a framework using OpenApI more research has gone into the OpenAPI specification than Swagger. To get a deeper understanding of how Swagger works and how to use it, the documentation at Swagger's official webpage (Swagger io, 2026) was valuable. Here they explain how to both setup Swagger but also how to use it to create API requests from the browser. Documentation for Swagger can also be found on their GitHub page (Swagger GitHub, 2026). The book by (Price, 2025) was also used to get a deeper understanding about Web API:s and the OpenAPI specification. Explanation for the OpenAPI specification can also be found on their GitHub page (OpenAPI GitHub, 2026).
RabbitMQ: No coding is necessary to set up RabbitMQ. To download and set up RabbitMQ the documentation from RabbitMQ's official web page (RabbitMQ Page, 2026) was used. From this web page the Open-Source RabbitMQ Server was downloaded and installed onto a

[1] a22wilas/Final-Exam

client. Instead of coding RabbitMQ, a K6 script to send messages and a consumer in the Razor application to receive the messages had to be developed. The K6 script was like the API script but needs to use authorization when connecting to RabbitMQ. The information for authentication and authorization was found in (RabbitMQ Page, 2026) section How to Manage RabbitMQ. K6can not connect directly to RabbitMQ but instead needs an HTTP connection between the systems. On (RabbitMQ Page, 2026) plugin section they explain a built in HTTP API and how to activate it for RabbitMQ. One of the harder parts to solve was the consumer in the Razor app. Some insight into how to create one was found at (RabbitMQ Stack overflow, 2026)

5.2 K6

To be able to test the thesis messages that need to be sent to an API and Message Queue, this will be done using Grafana K6. With K6 JSON data can be sent to a selected endpoint. Grafana K6 is an open-source software, documentation for how it works and how to install it is available on Grafana's webpage. K6 uses JavaScript to handle all the logic, both to open a file but also to send the data to the endpoint. In the JavaScript additional logic can be added to handle how many simulated machines will send data and how the data will be iterated through. Logic can be added to iterate through the data in sequence or using randomization, and if the data should be iterated through once or use a timer for how long data should be iterated through. The K6 scripts can also be modified to send additional data, for example a timestamp when every JSON object was being sent.

Installation: The installation process can vary between the different operating systems. For this thesis the artifacts have been installed on windows. The installation on windows can either be done using Choco or the windows package manager. For this thesis the windows package manager has been used. To install the commando *winget install k6 --source winget* is executed in the Command Prompt (CMD).

Api script: The API script imports K6 and SharedArray to be able to use the JSON file. The data from the JSON file gets parsed into the variable data. (Figure 5)

```
import http from 'k6/http';
import { SharedArray } from 'k6/data';
const data = new SharedArray('data', function () {
  return JSON.parse(open('../JSON/data.json'));
});
```

Figure 5 Part of the K6/API script, handles the JSON file

```
export const options = {
  vus: 1,
  iterations: data.length,
  insecureSkipTLSVerify: true,
};
```

Figure 6 Part of the K6/API script, handles virtual users and iteration

In export const options we can decide how many users will send data while also handling the logic for how many times the JSON objects should be iterated through. From here we could change the logic, so data is being sent for a specific period. (Figure 6)

```
export default function () {
  const index = (__ITER % data.length);
  const item = data[index];
  const id = __ITER;
  const sentAt = new Date().toISOString();
  const payloadObj = {
    id: id,
    ...item,
    SentAt: sentAt
  };
  const payload = JSON.stringify(payloadObj);
  const res = http.post('http://localhost:5178/messages', payload, {
    headers: { 'Content-Type': 'application/json' },
  });
```

Figure 7 Part of the K6/API script, handles the message load

We create a payload that will be sent to the API. In the payload we add a unique id for every JSON object iteration, the JSON data as an item and a variable holding the timestamp for when the message object is being sent. To make the Swagger API able to read this we convert the payload back into a JSON. The new JSON data is being sent to the endpoint located at localhost:5178/messages where the swagger API will later retrieve it. (Figure 7)

```
console.log(JSON.stringify({
    id: id,
    sentAt: sentAt,
    status: res.status,
    receivedAt: responseBody.receivedAt || null,
    payload: payloadObj
  }));
```

Figure 8 Part of the K6/API script, handles the logging

The console.log does not handle the logic or sending of data to the Swagger API but is instead a means to validate what data is being sent from the K6. From here we can log the data and in the command prompt write it out to a txt file[2]. (Figure 8)

MQ script: Most of the MQ script is the same as the API script with some small changes, so to not repeat the same code sections with the same explanation we will only explain the differences between the two scripts in this section.

```
import encoding from 'k6/encoding';
…
const credentials = encoding.b64encode('guest:guest');
```

Figure 9 Part of the K6/MQ script, handles credentials

When connecting to RabbitMQ we need to use credentials for authorization. The user that is used for this thesis is "guest" with the same password as username. To do this we encode the credentials using base64 and add the value to the variable credentials. (Figure 9)

[2] https://github.com/a22wilas/Final-Exam/commit/28c15bb

```
export default function () {

…

    const payload = JSON.stringify({

       properties: {},

       routing_key: "logistic_mq_messages",

       payload: encoding.b64encode(payloadObj),

       payload_encoding: "base64"

    });

    Const res =
http.post(      'http://localhost:15672/api/exchanges/%2F/amq.default/publish',

    payload,

    {

      headers: {

        'Content-Type': 'application/json',

        'Authorization': `Basic ${credentials}`

      }

    }

  );
```

Figure 10 Part of the K6/MQ script, handles the connection to RabbitMQ

From here one can see the difference between sending a message directly to an endpoint versus sending it to one of RabbitMQ message queues. We need to encode the payload using base64 and route it to the message queue, the message queue for this thesis is logistic_mq_messages. By using the HTTP plugin, we can send this to the endpoint at localhost:15672 that RabbitMQ listens to. In the headers we also need to include the credentials to be able to connect to the RabbitMQ user. (Figure 10)

JSON: To hold the data that will be sent through the API and MQ a JSON file has been created. The object structure is like what is used by the real-life logistic application at Volvo but uses dummy data in every string element.

```
{
        "AuthorityCodeField": "add",
        "AssignmentIdField": "1",
        "AssignmentReferenceIdField": "0",
        "Package": "3001",
        "PickupLocationField": "Loc_001",
        "DeliveryLocationField": "Loc_002",
        "PriorityField": "8"
}
```

Figure 11 The JSON object used for the experiment

This represents one JSON data object. Every object uses the same elements but with different values. For real life production the AssignmentIdField and Package should be unique and not be able to be received by the Razor application if they already exist. For this thesis and the artifacts this is not the case but instead a unique ID is being sent from the K6 artifacts for every message object[3]. (Figure 11)

5.2.1 Progression

To start with a simple K6 artifact was created, this was one of the first things that were created for this thesis. The first artifact worked as a prototype and did not send the message to Swagger API or RabbitMQ but instead were only made to try out K6 as a tool. To see if the K6 artifact worked a logger was added, through this one could see timestamps and the messages that were sent through the command prompt. The first artifact was called FirstTest.js[4]. After successfully being able to send messages from the K6 artifact a JSON was created. In the JSON the messages were constructed, at this time with only one object. The K6 artifacts were modified to read the JSON and send the data to an endpoint.

After having a working K6 artifact the Swagger API artifact was created. The old K6 artifact was not compatible with the Swagger API and thus a second K6 artifact was created with the name SecondTest.js[5]. By modifying SecondTest the data could now be sent from the artifact to the Swagger API. Many different iterations were tried with how the data should be iterated through. Multiple tests were done with how the data should be sent, every JSON object iterated through once or using a timer for how long the data should be iterated through.

[3] https://github.com/a22wilas/Final-Exam/commit/753fe1d

[4] https://github.com/a22wilas/Final-Exam/commit/0b266d7

[5] https://github.com/a22wilas/Final-Exam/commit/5a4a22f

Testing was also done by increasing the number of users sending data at the same time[6]. At a later stage another K6 artifact was created, MQTest.js. This was like SecondTest but sent the data to the endpoint that RabbitMQ listens to. When both artifacts were working SecondTest was scrapped and APITest.js was instead created to have a better naming.

5.3 Swagger - Rest API

To receive and handle the JSON message we create a Swagger API that uses the endpoint address localhost:5178. The swagger API will be connected directly to the Razor application.

Installation: Swagger API uses dotnet logic and therefore dotnet needs to be installed on the computer, more about this in the Razor dotnet application section. To create an API the command dotnet new webapi -n CreateTransportAssignment was executed in command prompt. Swagger API uses Swashbuckle, to install this one needs to be in the Swagger API directory for CreateTransportAssignment. To install swashbuckle run the command dotnet add package Swashbuckle.AspNetCore. To start CreateTransportAssignment run the command dotnet run, important that the command prompt is in the right directory. After installation a Swagger API template will have been created with a complete map structure. This template creates multiple files that are necessary to run the Swagger API in a dotnet environment. Most of these files will not be modified or touched for the artifact creation. Two files will be added, Message.cs and MessageStore.cs while the main Program.cs file will be modified.

Message.cs: This is used to handle the JSON objects sent from K6. This file is the data contract and defines what a Message is and what should be included in it. Ensures a consistent structure between the API request and the API response.

[6] https://github.com/a22wilas/Final-Exam/commit/96ceb0e

```
using System.Text.Json.Serialization;

namespace CreateTransportAssignment.Models;

public record Message(

    int Id,

    [property: JsonPropertyName("AuthorityCodeField")]

    string AuthorityCodeField,

    [property: JsonPropertyName("AssignmentIdField")]

    string AssignmentIdField,

    [property: JsonPropertyName("AssignmentReferenceIdField")]

    string AssignmentReferenceIdField,

    [property: JsonPropertyName("Package")]

    string Package,

    [property: JsonPropertyName("PickupLocationField")]

    string PickupLocationField,

    [property: JsonPropertyName("DeliveryLocationField")]

    string DeliveryLocationField,

    [property: JsonPropertyName("PriorityField")]

    string PriorityField,

    DateTime? SentAt,

    DateTime? ReceivedAt

);
```

Figure 12 Part of the Swagger artifact, handles message values

We use a record to define the data model to make sure the values are not changed after its creation. In the message we use JsonPropertyName to map each of the JSON elements and add the values to variables that will be used in the Razor app. We use DateTime? to make sure the message goes through even if those values would be NULL. (Figure 12)

MessageStore.cs: This file works as a service and holds the messages retrieved from the Message file. Without this the API would receive the data, process it and then lose it. Every message gets added into a list by using POST and later retrieved using GET. By using POST and GET the content in the list can't be modified, instead messages can only be added or

removed from the list. The negative part with using a service like this is that when the Swagger API is turned off all data is removed and can't be accessed by the Razor application.

```
using CreateTransportAssignment.Models;

namespace CreateTransportAssignment.Services;

public class MessageStore

{

    public List<Message> Messages { get; } = new();

}
```

Figure 13 Part of the Swagger artifact, handles a list that holds messages

List<Message> is a collection that can hold message objects and new() initializes the list to make it ready to use. The { get; } part is what makes the list readable from the Program.cs file. (Figure 13)

Program.cs: This file works as the entry point and configuration for the Swagger API. It sets up the MessageStore service, enables Swagger for testing and defines the API endpoint that will handle incoming HTTP requests. It creates a web API that will allow POST and GET requests while also storing the messages in a memory.

```
using CreateTransportAssignment.Models;

using CreateTransportAssignment.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton<MessageStore>();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

app.UseSwagger();

app.UseSwaggerUI();
```

Figure 14 Part of the Swagger artifact, activates swagger framework

The "using" code rows makes sure that we can access the files Message and MessageStore. The builder section initializes the application, registers MessageStore as a singleton to make it into one shared instance for the entire application, enables Swagger and builds the application.

The last two swagger rows activate swagger and the webpage interface to make it possible to see messages and the structure in a UI. (Figure 14)

```
app.MapPost("/messages", (Message message, MessageStore store) =>

{

    var receivedAt = DateTime.UtcNow;

    var stamped = message with

    {

        Id = store.Messages.Count + 1,

        ReceivedAt = receivedAt

    };

    store.Messages.Add(stamped);
```

Figure 15 Part of the Swagger artifact, convert JSON data into message objects

This code defines an endpoint to handle HTTP POST requests. From here the Swagger API automatically converts the JSON data into Message objects. Into the message we also add an Id that is based on how many messages exist in MessageStore and a timestamp for when we receive the data[7]. (Figure 15)

```
Console.WriteLine($"[RECEIVED]              Id={stamped.Id}              |
SentAt={stamped.SentAt:O} | ReceivedAt={receivedAt:O}");

    return Results.Ok(new

    {

        status = "received",

        id = message.Id,

        sentAt = stamped.SentAt,

        receivedAt = receivedAt

    });

});
```

Figure 16 Part of the Swagger artifact, handles logging

[7] https://github.com/a22wilas/Final-Exam/tree/8bc52b6

The first line Console.Writeline is used to log and print id, sentAt and ReceivedAt into a console. A response is returned using HTTP 200 OK to confirm receipt with message data. Without this the endpoint would not compile. (Figure 16)

```
app.MapGet("/messages", (MessageStore store) => store.Messages);

app.Run();
```

Figure 17 Part of the Swagger artifact, handles endpoint and activation

Defines the HTTP Get endpoint and returns all messages from MessageStore. The last line activates the web server and the listening to HTTP requests. (Figure 17)

5.3.1 Progression

The installation for Swagger using Swashbuckle was installed on a local machine, and a web API was created using the command prompt. This was a pre-built weather template. The API at this stage was not connected to either the K6 artifact or the Razor application[8]. At this moment the Swagger API was modified to use the data sent by K6 SecondTest and log it locally. This was done to see so the endpoint was working and be able to get a visual through the command prompt to see that all the messages were retrieved[9]. The last thing to do was connect the Swagger API to the Razor application. Most of this modification was done in the artifact for the Razor application as the logic to send the data to an endpoint already existed in the artifact[10].

5.4 RabbitMQ - Message Queue

RabbitMQ is an open-source message queue. No artifact has been created for RabbitMQ, instead the RabbitMQ server has been downloaded and installed on the local test computer. RabbitMQ can be installed in two different ways, it can either use Choco or Erlang. The documentation and installation process can be read on RabbitMQ official webpage. For this thesis the Erlang route has been used. Erlang can be built from source or use the pre-built package. To use the pre-built package the windows installer package can be downloaded from Erlang's official webpage.

Installation: Download both Erlang and RabbitMQ from their official webpages and install the packages on the local machine. RabbitMQ has an inbuilt dashboard; to activate the

[8] https://github.com/a22wilas/Final-Exam/tree/4d47e4e

[9] https://github.com/a22wilas/Final-Exam/tree/28c15bb

[10] https://github.com/a22wilas/Final-Exam/tree/e8f8f61

dashboard the command enable “rabbitMQ_management” needs to be installed into the file RabbitMQ Server\rabbitmq_server-3.x.x\sbin\rabbitmq-plugins.bat using CMD[11].

The default web address for the RabbitMQ Management Plugin web UI is http://localhost:15672. To access the web UI the localhost:15672 needs to be entered into a web browser.

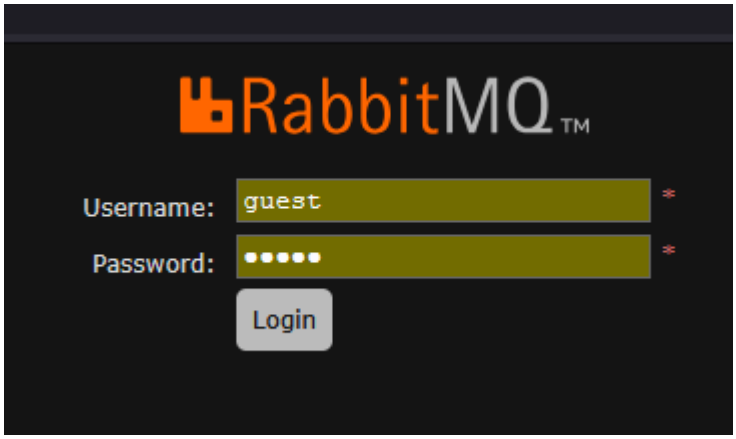

Figure 18 Picture showing the login page to RabbitMQ

When accessing the RabbitMQ web UI a username and password is needed, the default user for RabbitMQ is guest with the same password. Username: guest, Password: guest. Additional users can be created with different permissions. For this thesis the default user guest has been used. (Figure 18)

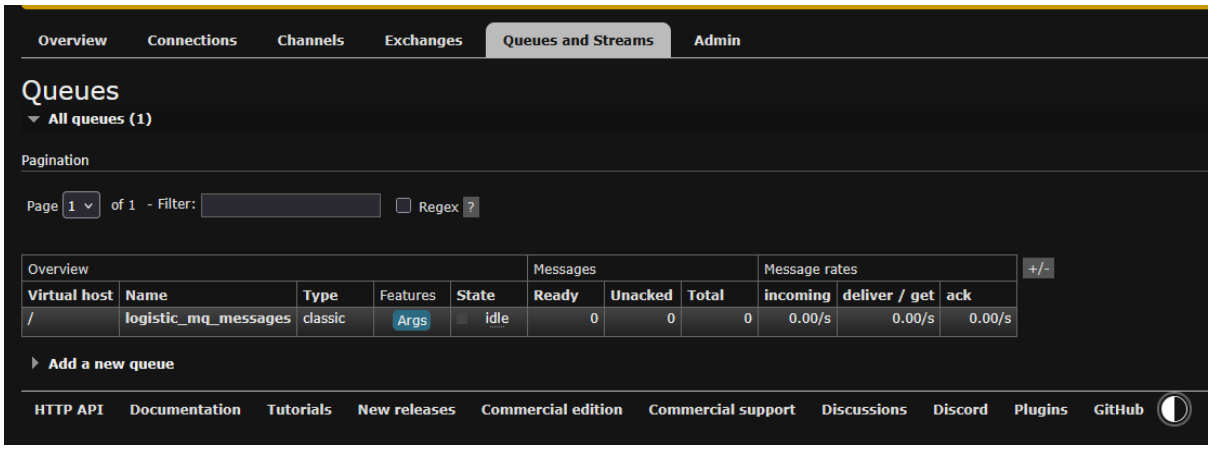

Figure 19 Picture showing the queue used for the experiment

In RabbitMQ the message queue logistic_mq_messages has been created, this queue receives the messages from K6 and are used to communicate with the consumer in the dotnet application. (Figure 19)

[11] https://github.com/a22wilas/Final-Exam/tree/cfbfead

RabbitMQ cannot communicate directly with K6 but instead needs some sort of communicator between them. One way to do this is to create an API that receives the message from K6 and connects it to RabbitMQ. This could potentially create overhead and slow down the speed for the messages and therefore increase the latency. Another way to solve this issue is to use the inbuilt HTTP plugin. To enable this the command, enable "rabbtimq_web_dispatch" also must be installed into the file RabbitMQ Server\rabbitmq_server-3.x.x\sbin\rabbitmq-plugins.bat using CMD. By doing this RabbitMQ can listen to an endpoint like how the API works and receives the messages directly from the K6 artifact. This thesis uses the HTTP plugin.

5.4.1 Progression

From the start IBM WebSphere provided by Volvo was going to be used, because of certain complexity with their Razor application, a new method had to be designed. In this new method the Message Queue tool was changed from IBM WebSphere to RabbitMQ. The progression for RabbitMQ lies more in the K6 and Razor artifacts. RabbitMQ was installed on the local machine and started using the command net start RabbitMQ in CMD. The K6 script MQTest.js was created and the Razor app was updated with a consumer and its own Razor page[12].

5.5 Razor dotnet Application

This artifact is the largest of the ones created and has multiple files that have both been created and modified. In this section we will go through the most important parts of the artifact and how it works. The entire artifact with all code files is available publicly in the GitHub repository (GitHub).

The Razor dotnet application will retrieve the data from the Swagger API and RabbitMQ and visualize it into a table. Two different pages on the Razor app have been created, each to hold a table for the respective communication system. A JavaScript is used to download the table data into a CSV file when a button is pressed. The structure for retrieving data is a bit different depending on the communication system. The retrieving logic for the Swagger API is done solely in the Index.cshtml.cs file while the retrieving from RabbitMQ first goes through a consumer and uses a MessageStore and MQ.cshtm.cs to handle the logic.

To access the API page: http://localhost:5004/

To access the MQ page: http://localhost:5004/MQ

Installation: Before being able to create a Razor application dotnet needs to be downloaded and installed on the device. This can be downloaded from Microsoft's official dotnet webpage. After dotnet has been installed, run the command line dotnet new webapp -n LogisticApp. To make RabbitMQ available for the Razor app we also need to run the command dotnet add package RabbitMQ.Client. The next three command lines should be executed in order, dotnet clean, dotnet build and dotnet run. This will compile, build and start the Razor application.

[12] https://github.com/a22wilas/Final-Exam/tree/585bbcd

Program.cs: This is the entry point and configuration hub for the Razor application. It configures how http requests are processed, sets up services such as Razor pages, the in-memory message store and the RabbitMQ consumer running in the background.

Item.cs: Holds the structure for the received messages. Uses a get/set to receive the message data from Swagger API or RabbitMQ and assign the value to Razor app variables.

downloadButton.js: This JavaScript takes the data from the tables and creates a CSV file with the table data. The script gets activated when a user presses the download button on the pages and the file is automatically downloaded on the user's device. The same script is used for both the API and MQ pages but uses different functions. The function downloadAPICSV() is used to download from the API page while the function downloadMQCSV() is used for the MQ page.

Index.cshtml.cs: In this file we have the logic for retrieving the Swagger API messages and adding them into a list.

```
public List<Item> Items { get; set; } = new();

…

var response = await
client.GetStringAsync("http://localhost:5178/messages");

…

Items = JsonSerializer.Deserialize<List<Item>>(response, options)

                    ?? new List<Item>();
```

Figure 20 Part of the Razor artifact, handles the retrieving of messages

We first initialize a new List using the Item class from Item.cs. We listen to the Swagger API endpoint at localhost:5178/messages to retrieve the message data. We then add the response into our list. (Figure 20)[13]

Index.cshtml: This file handles the creation of the API Razor page; this is done by using HTML to make the button and the table.

[13] https://github.com/a22wilas/Final-Exam/tree/e8f8f61

```
<button onclick="downloadMQCSV()">⬇ Download as CSV</button>


<table id="MQdataTable" border="1" cellpadding="6"cellspacing="0">

@foreach (var item in Model.Items){

 var latency = (item.SentAt.HasValue && item.ReceivedAt.HasValue)

    ? (item.ReceivedAt.Value -

    item.SentAt.Value).TotalMilliseconds.ToString("F2")
    : "N/A";

 <tr>

    <td>@item.Id</td>

    <td>@latency</td>
```

Figure 21 Part of the Razor artifact, creates a button and a table

We create a button with the onclick function to start the function downloadAPICSV() in downloadButton.JS. After that we create a table that loops through all the objects in our item List. We use the values from SentAt and ReceivedAt to calculate the difference for the latency column. We then map the table columns to the values from the Item list. (Figure 21)

RabbitMQConsumer.cs: To be able to receive the message data from RabbitMQ a consumer is necessary. From here we can connect to the RabbitMQ user "guest" and select the message queue logistic_mq_messages. We put all the data into an array that we parse into a JSON. Like for the API we create a list from the Item and add the JSON elements into our variables. We do this so we can use them for the table on the MQ.cshtml razor page.

MQMessageStore.cs: This works as a service in the same way as MessageStore in the Swagger API artifact. This makes sure that we can POST and GET items from the list and makes sure that we don't throw away the messages after we have retrieved them. A difference between the API:s service and this service is that the data is removed if the Razor page is restarted while for the API the data only gets removed if the Swagger API is restarted.

MQ.cshtml.cs: Here we handle the logic for retrieving the data for the Razor page, works like the Index.cshtml.cs file. This connects the item list to the Razor page MQModel that will be used in the MQ.cshtml file.

MQ.cshtml: From this file the creation of the MQ Razor page and the HTML elements are created. This is almost identical to the Index.cshtml with a difference to which function is activated when the downloadButton is pressed and has another ID for the table.

5.5.1 Progression

The progression for the Razor application artifact has been the longest of all the artifacts created. The Razor application has gone through the same process as the Message Queue. At first a simple artifact was created only to see that the Swagger API was working[14]. This artifact was not originally meant to be used for this thesis but instead an already existing Razor logistic application provided by Volvo was going to be used. After having created the K6 and Swagger API artifacts connecting them to the Logistic application was more complex than anticipated. Because of time restrictions a decision was made to instead change the method for this thesis. The new method followed the same design with a message being sent from K6 to Swagger API which was connected to a Razor application, but instead of using the Logistic application from Volvo the development of the Razor artifact was continued[15].

First came fixing the handling of the Swagger API to make sure the data could be visualized on the Razor webpage. This was done by creating a table where every message got presented in its own row. Logic for the latency calculation was also added to use SentAt and RecievedAt to see how long time every message took from being sent to retrieved. As the data also had to be used a JavaScript called downloadButton.js was created to be able to download the table data into a CSV file[16]. When the Razor artifact could receive the Swagger API data and use it the development shifted to make the same functionality for RabbitMQ. This turned out to be a bit trickier to design than the handling of the Swagger API data. An API is directly connected to an application and can handle the HTTP logic and service like a MessageStore. This is not the same for a message queue, instead of a consumer and a service needs to be added to the Razor application. Because of this, four new files were created: RabbitMQService, MqMessageStore, MQ.cshtml.cs and MQ.cshtml. With the addition of these files the messages sent from RabbitMQ could now be used in the Razor application and be visualized on its own table[17].

[14] https://github.com/a22wilas/Final-Exam/tree/c0ed201

[15] https://github.com/a22wilas/Final-Exam/tree/f7fd71e

[16] https://github.com/a22wilas/Final-Exam/tree/7634cfb

[17] https://github.com/a22wilas/Final-Exam/tree/585bbcd

6 Pilot study

To evaluate the tools and the methods we do a pilot study. The pilot study will show us if the method can be used to get reliable and useful data to make an analysis. The analysis shall be able to answer the research questions and either show if the hypothesis is correct or false. The pilot study will also show if the tools are working and we can use them to get measure points for the analysis. The pilot study was done using commit 753fe1d[18]. For both artifacts 300 measurement points were taken and used for the analysis.

6.1 Tooling

Multiple command prompts are used locally to run the API, dotnet application and the K6 scripts. A RabbitMQ server is started locally. A Firefox web browser is used to visualize the dotnet application and the table that holds the received data. From the browser the table data can be downloaded into a CSV file. The JSON data used for the pilot study has been manually created. All the artifacts are running using localhost.

Software versions used for the pilot study:

- Firefox - Version 149.0.2 (64 bit)
- RabbitMQ server - Version 4.2.5
- Command Prompt (CMD) - Version 10.0.26200.8246
- K6 - Version 1.6.1
- dotnet - Version 10.0.201

Hardware used for the pilot study:

- CPU - AMD Ryzen 7 78003XD 8-Core
- Memory - 64GB at 5200 MT/s
- Disc - Samsung SSD 990 EVO 2TB

6.2 Scripts

The dotnet and Swagger API artifacts are being run in the background and only need to be compiled and started for the Pilot study while the RabbitMQ server only needs to be activated.

[18] https://github.com/a22wilas/Final-Exam/tree/753fe1d

The same is not true for the K6 scripts, these need to be executed manually to be able to send data into the Swagger API and the RabbitMQ. To run the scripts the command prompt first needs to have the directory where the scripts lay, or the location needs to be added into the command. For this thesis and Pilot study the scripts are in Final-Exam/K6 measurement tool/tests.

Run the K6 script for Swagger API: K6 run APITest.js

Run the K6 script for RabbitMQ: K6 run MQTest.js

6.3 Results

The results from the pilot study show that both the tools and the method are working and can be used to create a measure series with mostly consistent measure points. For the Pilot study 300 JSON objects were sent through the Swagger API and the RabbitMQ. This gave us a result of 300 measure points. As said the measure points are mostly consistent with some small spikes, except for one big spike for the Swagger at the start and one big for RabbitMQ at the end. It's unknown why these spikes happened and is something that needs to be monitored when multiple trial runs are made for the experiment.

Execution: First the trial run for the Swagger API was executed, this was done by running the APITest script. By updating the dotnet application webpage the table had now been updated with one row for every JSON object sent. This was then downloaded into a CSV file using the in-built download button on the webpage. After this the API was turned off, the dotnet application restarted with cache cleared and the RabbitMQ server activated. Then the script for RabbitMQ MQTest was executed and like Swagger API the data from the dotnet webpage was downloaded into a CSV file. Both tables have a column for latency that is calculated by the dotnet application using the sent and received variables. This latency column was used to create a line diagram to show the difference between the message speed for Swagger API and RabbitMQ.

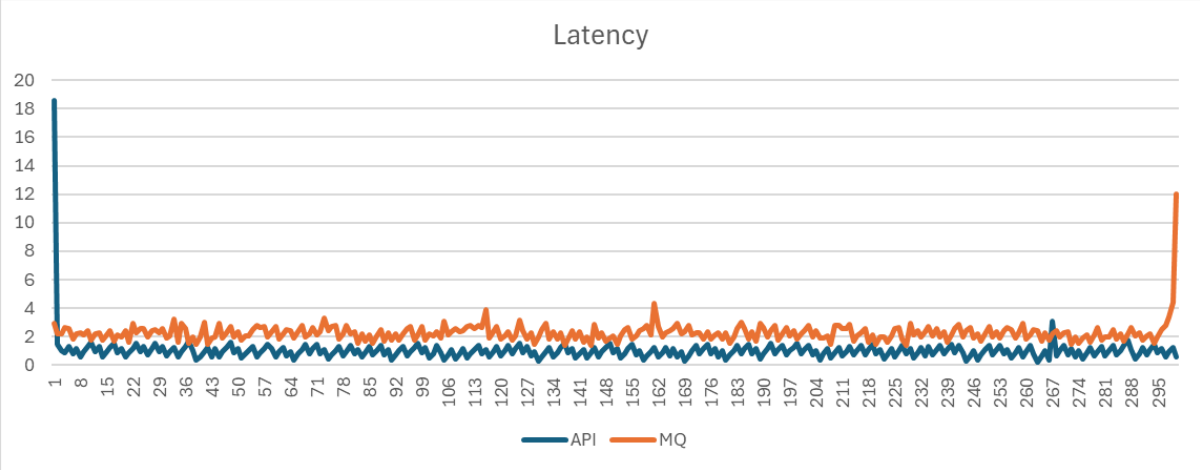

Figure 22 Line diagram for the pilot study measure series

For the Pilot study only one measure series was done for each of the treatments. Only one machine sent the messages, and all the JSON objects were iterated through once. Both Swagger API and RabbitMQ send the messages in the order of the JSON file. The order shown in the tables on the dotnet application are not the same for the different communication systems. For Swagger API the order is the same as the JSON file but for RabbitMQ it is the reverse. In Figure this has not been considered and the order for the measure points for RabbitMQ should be reversed (Figure 22). When doing the experiment this needs to either be changed for the artifact or be considered when creating diagrams and doing the analysis. This also shows us that the big spike at the start for Swagger API and at the end for RabbitMQ is happening at the start for both communication systems.

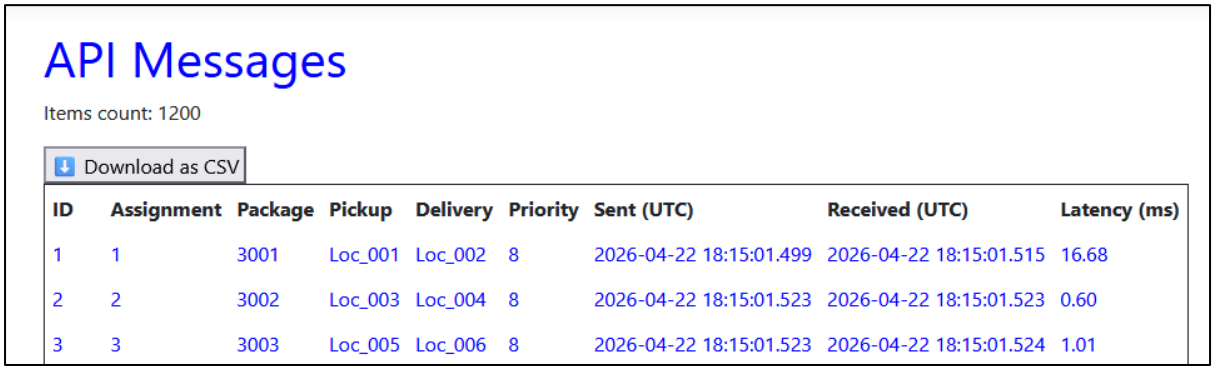

Figure 23 Showing the table order for Swagger API

A table holds and visualizes the message values from Swagger API (Figure 23)

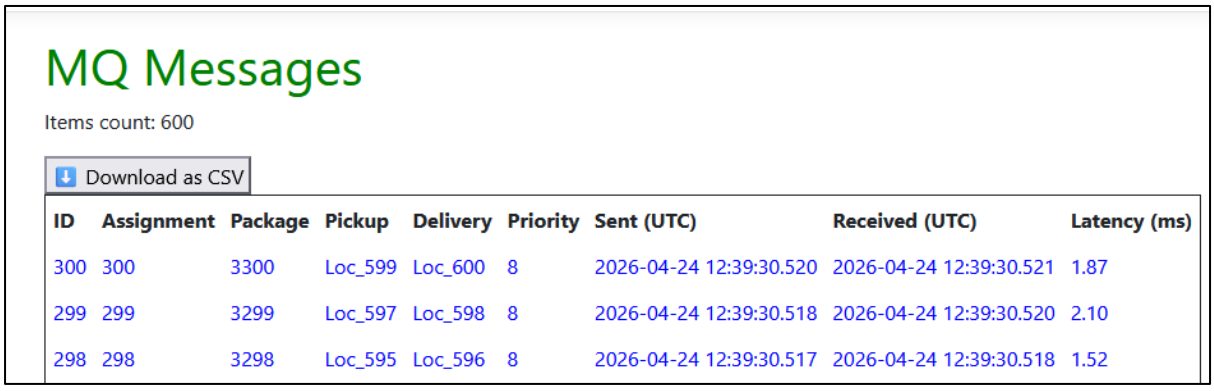

Figure 24 Showing the table order for RabbitMQ

A table holds and visualizes the message values from RabbitMQ (Figure 24)

System:	Swagger API	RabbitMQ
Mean:	1,038133333	2,280466667
Std dev:	1,071747723	0,719461743
Std error:	0,061774512	0,041538143
Min:	0,21	1,37
Max:	18,58	11,98

Table 1 Shows statical results from the Pilot Study

Table **s**hows the mean, min, max values and statistics like standard error and standard deviation for the communication systems. (Table 1)

From the measure series the result shows that RabbitMQ had a mean time that was over double that of Swagger API. Swagger API also had a much lower minimum speed but because of the big spike in the beginning it also had the highest max speed.

6.4 Pilot study analysis

The pilot proves that the method and the tools can be used to get a functional result but has some problems that need to be addressed and fixed for the final experiment. The measure points for Swagger API and RabbitMQ are not in the same order, either this needs to be fixed in the artifacts or considered when creating diagrams. This does not change any of the values from table 1 and is therefore not a threat to any validity but does make the visualization of diagrams a bit confusing and does not show the exact correlation of the measure points against each other.

Two big measure point spikes happen at the beginning of both the Swagger API and the RabbitMQ. For the Pilot study only one trial run was executed per communication system, for the final experiment multiple trial runs will be executed. The multiple trial runs will show if these spikes were a one-time thing or if it happens every time a K6 script is executed. Depending on the result, action needs to be taken, if the same spike happens every time for both communication systems these measure points could potentially be erased from the result, diagrams and statistics. From the pilot study we only see the result of the speed it takes to send a message, with other words the latency. From the research questions we also want to see the throughput of the communication systems. When sending 300 JSON objects no missing data was seen, instead both communication systems managed to get all 300 JSON objects to the

dotnet application without losing any data. To better see throughput either the number of objects being sent must be increased or increase the number of users that send data. Even if both communication systems manage to handle all objects with increased load this should be documented into some sort of diagram for the final experiment.

The result is being downloaded from a table on the dotnet webpage. Depending on how many JSON objects are being sent the table can grow very large. With 300 rows the table loaded in quickly and for the Pilot study this was not a problem but when sending a much larger chunk of objects the load time increases drastically. For the final experiment we want to send up to 10000 JSON objects. Instead of showing all the data in the table and using a download button the dotnet artifact should be changed so it creates a file automatically with the trial result and not show all the rows at the same time on the webpage. In this pilot study we send the JSON objects by iterating them once, we do this in the order of the JSON file. A better way to do this is to let K6 randomize the order in which it sends the object as well as being able to send the same object more than once. Important here is to make sure that the randomization is seeded to make sure the same order is sent for both communication systems.

6.5 Conclusion

The point of the Pilot study is to show if the tools work and the method can answer the research questions. For this thesis two research questions are asked:

RQ1: How does the time performance differ between REST API and Message Queue when retrieving and sending a message?

RQ2: How does the reliability in message loss differ between REST API and Message Queue when sending multiple messages?

The pilot study uses K6 artifacts to send the messages, a Swagger API artifact and RabbitMQ server to handle the messages and a dotnet artifact to retrieve them. In the message from the K6 a timestamp is included for when it sends the message object while the dotnet application handles the timestamp for when it retrieves the message. The latency speed has been calculated by the dotnet application by calculating the retrieved timestamp by subtracting the sent timestamp. From the result we can draw conclusions about RQ1 and what the answer could be. We can see that latency can be tested and that there is a difference between the two communication systems.

For RQ2 no conclusion can be made as both communication systems had no memory loss and from the Pilot study the two systems do not differ when it comes to reliability. As this was only 300 measure points with one user sending them this result could change for the final experiment. For the experiment we will have to increase the workload, both to see the differences in transfer time but also to check if the communication systems will show any message loss. The experiment will be divided into five trials per system. The same JSON objects will be iterated through as for the Pilot study but the number of virtual users sending them will increase. The trials will be 1, 5, 10, 20 and 50 virtual users. By making an incremental increase the experiment may provide insight into how different workloads impact the transfer speed and message loss. The pilot study's goal is not to make a conclusion about the research questions or the hypothesis but from the result one can already get an impression of what the

result could look like. By looking at the results from the latency speed one can see that Swagger API has a faster time than RabbitMQ.

7 Evaluation

In the Evaluation section we will go through the experiment process and the result from the experiment. The result will be presented using both text and diagrams. The result will also be analyzed to make a conclusion regarding the hypothesis and the research questions. The experiment will use a total of 25800 measure points per communication system. Both RabbitMQ and Swagger REST API will have five trials with different amounts of virtual users sending messages.

7.1 Preparation

From the Pilot Study some issues were encountered. These issues involved both the process of doing the experiment and how the result materialized. The K6 scripts were not easy to modify to increase the number of virtual users sending messages. A small change had to be made to the scripts to make it easier to increase the number of messages sent. The Pilot Study showed that the throughput for both communication systems was stable when sending 300 messages on one virtual user and no message loss occurred.

```
const VUS = 50;

export const options = {

  vus: VUS,

  iterations: data.length * VUS,

  insecureSkipTLSVerify: true,

};
```

Figure 25 Part of the K6 scripts, changing how virtual users are handled

The table shows how the variable VUS changes the number of users, simplifying the process to test trials. (Figure 25)

At the time of the pilot study the Razor artifact used tables to visualize all the retrieved messages. This table was also used to download the result into a CSV file. When sending too many messages, the Razor page starts to slow down. For the experiment we want to be able to send thousands of messages to make sure we can achieve statistical difference between RabbitMQ and Swagger REST API. The code had to be updated, which was done by removing the table and saving the message data internally in the Razor applications memory. The messages from RabbitMQ were already stored in the memory in the form of a MessageStore but were only utilized for the creation of the table. Instead of requesting data from Swagger when creating the table, the request goes out when the button is pressed. For both communication systems the data is downloaded into a CSV file when the download button is pressed.

The process to run a trial run is like the one in the pilot study. Command prompts are used to start the Razor application and the Swagger API. RabbitMQ service is started locally on the client, and a command prompt is used to execute the K6 scripts. The pilot study only did one trial run per each communication system with a set amount of 300 JSON objects. In this experiment a total of 25800 JSON objects will be sent from each of the K6 scripts, this will be performed with five trial runs. The number of messages being sent will differ between trials by increasing the number of virtual users. Every virtual user will iterate through the same 300 JSON objects. The tools used for the experiment are the same as for the pilot study, both in the software and hardware department.

7.2 Results

In this section the results from the experiment will be presented and visualized. The experiment consists of ten measure series with five per communication system. The number of messages sent per virtual user will be 300 and the number of virtual users will be 1, 5, 10, 20 and 50. The experiment is conducted on commit: 0d8d24a[19].

The experiment will benchmark the speed of every message going to the communication system. The same number of virtual users will be measured and presented against each other. There will also be one result where the combined measure points from all series are presented. Throughput will also be measured, but not by measuring points but instead by checking if any message rows are missing from the CSV files and presenting the findings.

When measuring latency, we will remove the first message sent by every virtual user, the reason for this is that a spike consistently occurs for those messages for every measure series. This is true for both RabbitMQ and Swagger API. The reason for the spike probably has to do with the process power when starting the K6 scripts. This could be relevant if one is interested in seeing which communication system is fastest with sending one message, in this experiment we are more curious to see the differences for a large chunk of messages.

7.2.1 Latency

Virtual Users: 1

The pilot study sent 300 measure points by using one virtual user. In the experiment the same treatment is tested once. By running the same treatments again, we can see if there is any big difference from the Pilot study.

[19] https://github.com/a22wilas/Final-Exam/commit/0d8d24a

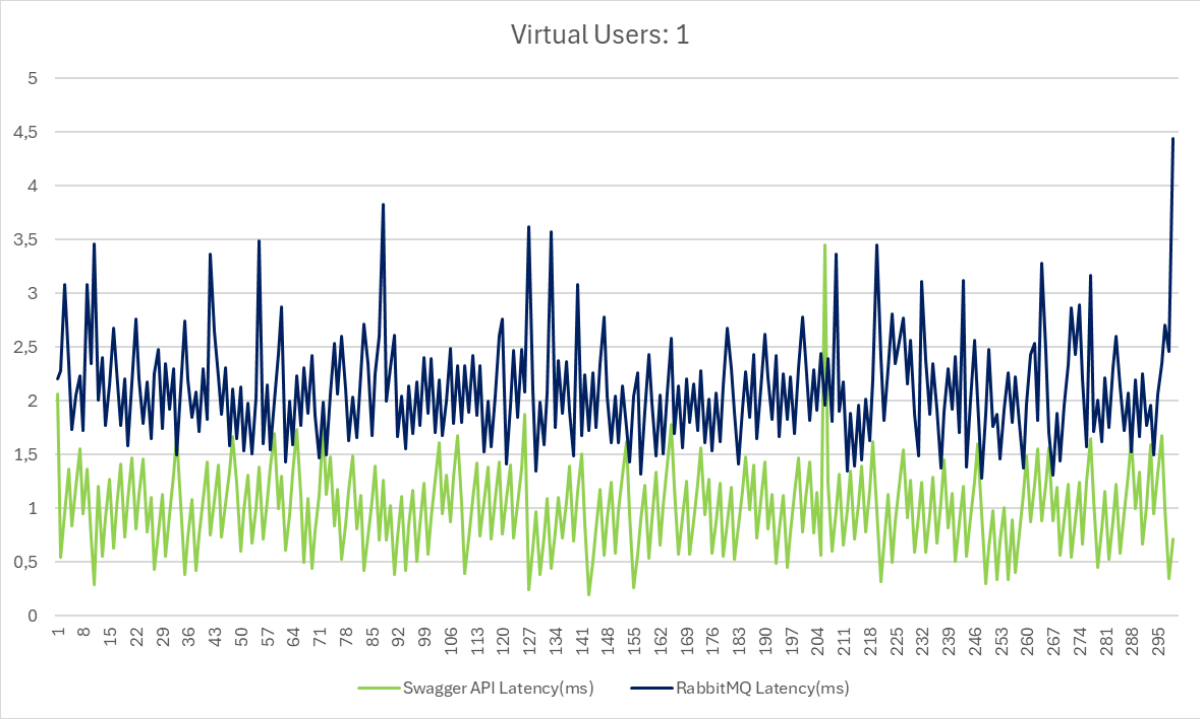

Figure 26 Line diagram presenting the 1 virtual user measure series

By comparing the diagram from the experiment to the one from the Pilot study we can see that the differences are minimal and can confirm that the tools and scripts are still working as intended. (Figure 26)

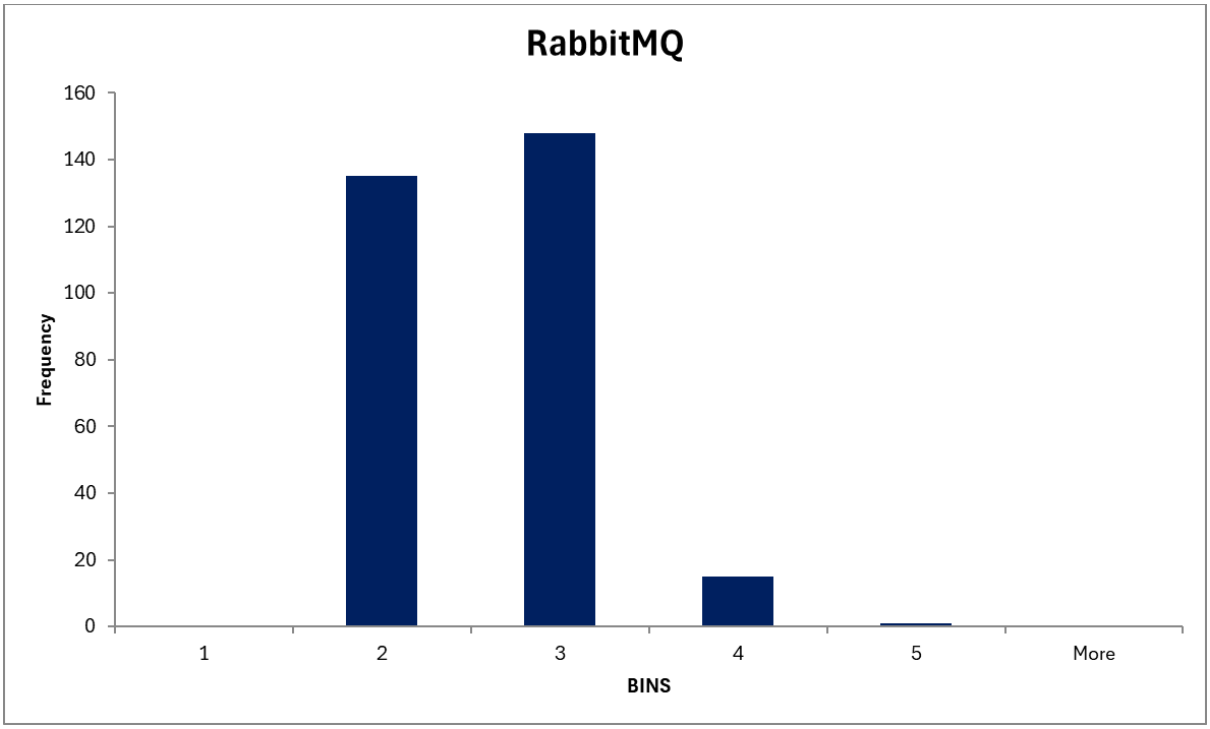

Figure 27 Histogram for RabbitMQ – 1 virtual user series

For RabbitMQ most measure points were between one and three milliseconds. Not a single measure point was under one millisecond, and a few were over three. (Figure 27)

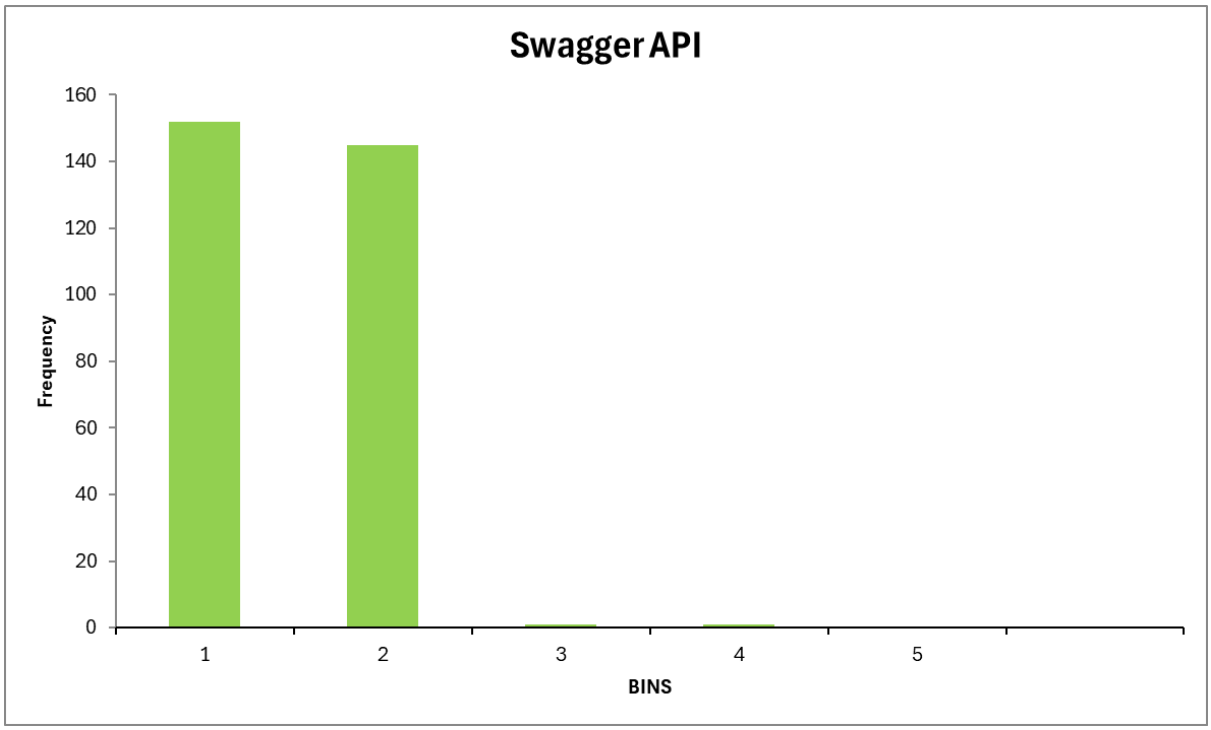

Figure 28 Histogram for Swagger API – 1 virtual user series

Where RabbitMQ had zero measure points under one millisecond most of Swaggers measure points are in that area. From the histogram we can also see that around half of the measure points happened between one and two milliseconds. Only two points were over two milliseconds. (Figure 28)

BINS (ms)	RabbitMQ	Swagger API
1	0	152
2	135	145
3	148	1
4	15	1
5	1	0
More	0	0

Table 2 Shows exact numbers for the histogram bins – 1 virtual user series

From the table we can see the differences between the two communication systems. Swagger API has a significant amount of measure points that are lower than RabbitMQ and has no points that are over four with RabbitMQ having one. (Table 2)

Virtual Users: 50

From here we will now scale up the experiment and present the results from the measure series using 50 virtual users. This will give us a clearer representation of how the communication systems work under high conditions. By increasing the load message loss has a higher chance of occurring, and with a larger sample the statistical analysis has a higher chance to show statistical difference.

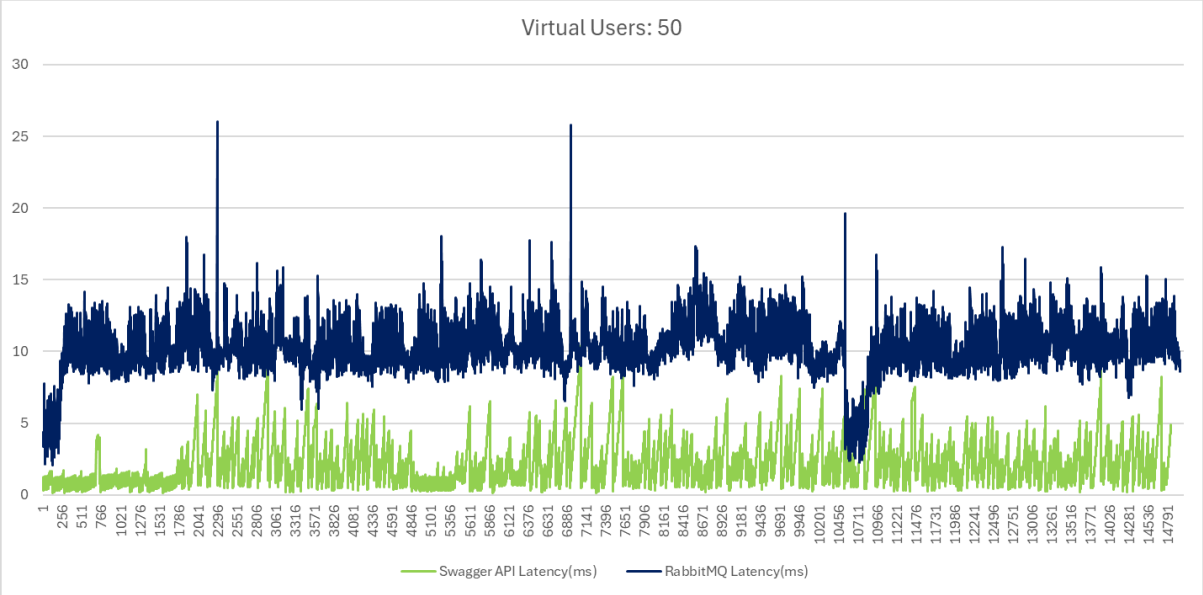

Figure 29 Line diagram presenting the 50 virtual user measure series

From the line diagram we can see that RabbitMQ lies a steep way above Swagger API. RabbitMQ also has some irregularities with two massive spikes but also two areas where the message speed was faster. Swagger API instead is more consistent for message speed with spikes that occur more frequently. Most of the spikes from Swagger API are under RabbitMQ:s speed and only in a few areas does RabbitMQ outperform Swagger API. (Figure 29)

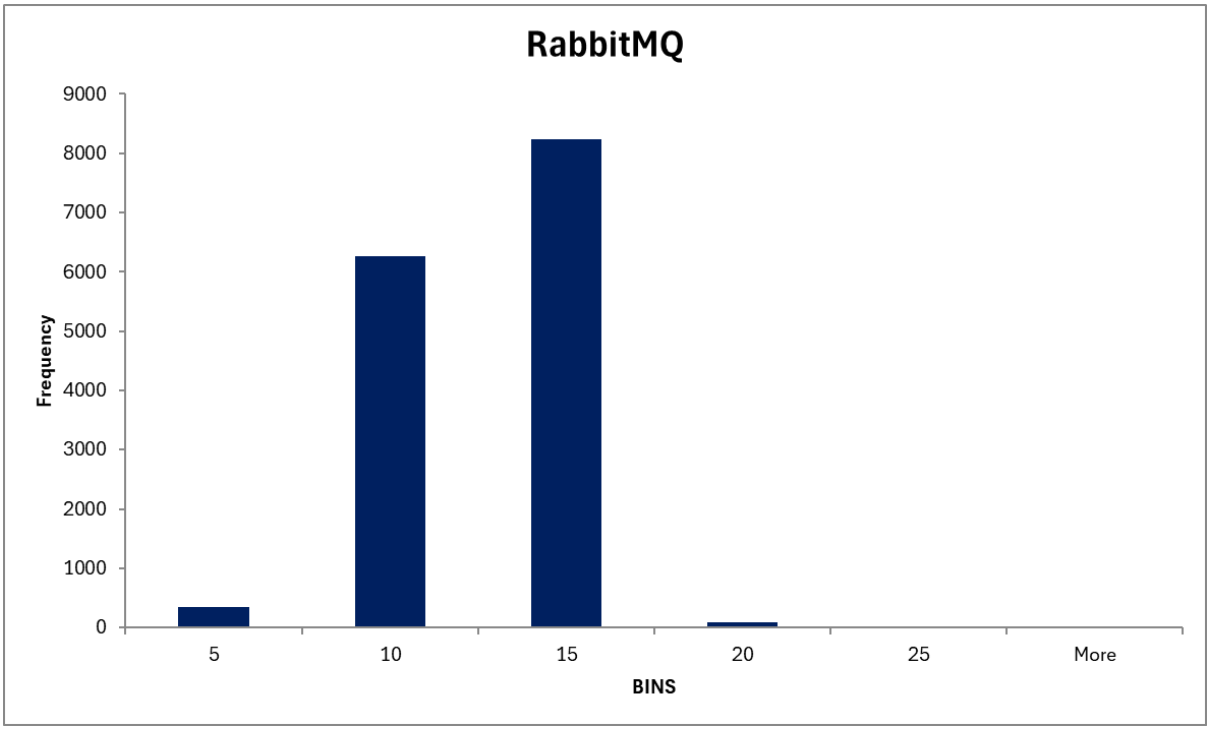

Figure 30 Histogram for RabbitMQ – 50 virtual user series

Most of RabbitMQ:s sample points are in the five to fifteen areas. Only 340 sample points were under five milliseconds. By comparing this result to the result from the experiment using one virtual user we can see that the maximum has increased significantly. This trial run also had a larger sample range which can be seen by the range of the bins. By increasing the treatment to 50 virtual users the speed has slowed down significantly from one virtual user. (Figure 30)

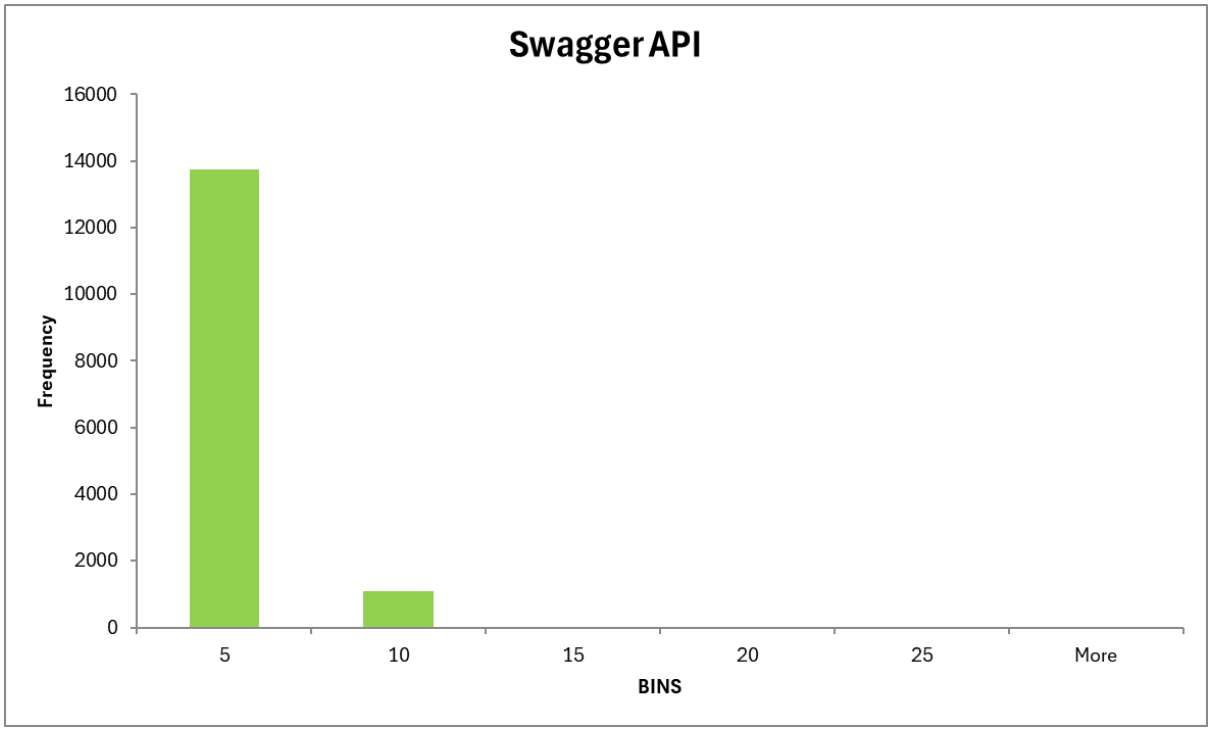

Figure 31 Histogram for Swagger API – 50 virtual user series

For Swagger API it is a completely different story than RabbitMQ. Here a vast majority of the sample points are under five milliseconds. The range of bins could have gone up to 15 but to keep consistency in the diagrams the same number of bins are used for both treatments. It is harder to make a good comparison for this treatment vs the one using one virtual user as we have no bins under five. Even if comparison cannot be made on the lower values the histograms show that the max value for a sample point has increased. (Figure 31)

BINS (ms)	RabbitMQ	Swagger API
5	340	13751
10	6271	1078
15	8243	7
20	96	0
25	0	0
More	3	0

Table 3 Shows exact numbers for the histogram bins – 50 virtual user series

This table presents the actual values for the treatment for all the bins. Swagger API has most points between zero and five while RabbitMQ has most between 10 and 15. The bins range cannot cover the entire extent of RabbitMQ's range and has three values that exceed 25. Swagger API instead has no sample points with values that exceed 15 and only seven that is higher than ten.

All measure series

In this section we will present the values for all the measure series.

RabbitMQ				
Virtual Users	Samples	Mean	Max	Min
1	299	2,1	4,44	1,28
5	1495	2,39	4,49	1,31
10	2990	2,88	6	1,68
20	5980	4,5	20,64	2,32
50	14950	10,31	26,01	2,06

Table 4 Shows statistics from all measure series for RabbitMQ

For RabbitMQ we can see a consistent rise in the mean value when the virtual users increase. The max value stays close for 1, 5 and 10 virtual users but rises to a significant amount at 20 virtual users and then a small amount at 50. The minimum values instead are close to each other for every treatment and only see a small rise for every virtual user increase. (Table 4)

Swagger Rest API				
Virtual Users	Samples	Mean	Max	Min
1	299	1	3,45	0,19
5	1495	0,92	3,24	0,15
10	2990	1,8	5,21	0,16
20	5980	1,64	5,75	0,16
50	14950	2,24	10,43	0,14

Table 5 Shows statistics from all measure series for SwaggerAPI

For Swagger API the mean value of the sample points is close to each other throughout the five treatments. Even if they are close, the increase for the mean value from 1 virtual user to 50 virtual users is approximately 124% showing that the average speed increases. Max value increases significantly when running 50 virtual users while the min value barely changes throughout the treatments with the lowest value at 50 virtual users. (Table 5)

7.2.2 Throughput

In this section we will look at the throughput and present it. This will be presented by showing how many messages were lost during the trial runs. With this information we may be able to answer research question two.

Messages Lost		
Virtual Users	RabbitMQ	Swagger API
1	0	0
5	0	35
10	0	31
20	0	55
50	0	92

Table 6 Shows message loss from all measure series for RabbitMQ and Swagger API

The result shows that RabbitMQ had no problem with message loss during the experiment, no matter how many virtual users were sending messages. The same was not true for Swagger API. At one virtual user there was no message loss, but when the user amount increased to five 35 messages were lost in the message process. At ten virtual users we see a decrease in lost messages but then at 20 users the increase starts again. The last treatment of 50 virtual users has the most lost messages with almost double the previous treatment.

7.3 Analysis

In this section the analysis of the result will be presented. To validate that the experiment has statistical difference an Anova test will be performed on the two treatments presented in the result section. By an Anova test we can see if the treatment has statistical difference and be able to reject the null hypothesis: No performance boost comes by changing from Message Queue to Rest API.

7.3.1 Latency

Virtual users: 1	Total
P-value	9,7E-127
F-value	965,2478

Table 7 Showing statistics from Anova test – 1 virtual user

The Anova results indicate that there is a statistical difference between RabbitMQ and Swagger API. The Anova test was conducted using the significance of alpha = 0,05. With a p-value of 9,7E-127 which is lower than the selected alpha Anova confirms that there are statistical differences. The null hypothesis can therefore be rejected. (Table 7)

The F-value represents the ratio of variance between the two communication systems to the variance within the systems. With a high F-value the variability between the systems is greater than the variability within themselves. The factor of changing the communication system most likely influenced the outcome. With a high F-value and a low P-value we can conclude that the experiment has a significant result. The probability of this not being true is effectively zero. This proves that the dependent variable has a meaningful difference for the treatments.

Virtual users: 50	Total
P-value	0
F-value	161557,2934

Table 8 Showing statistics from Anova test – 50 virtual users

The trials of 50 virtual users also showed statistical differences between the treatments. The P-value is lower and F-value higher than the values from the previous Anova. In the previous Anova test we concluded statistical difference and the effect of the factor, so the same would be true for these Anova values. (Table 8)

The P-value is written in the table as being zero, this is not actually true. Instead, the value is close to zero, with so many decimals that excel could not write it out. So, the P-value is not zero but as close as possible. This proves the statistical difference. With F-value being at 161557,2934 the factor probably influenced the outcome. The Anova tests have proven that the experiment's results have a statistical difference and can be used to analyze and make a conclusion. Both communication systems have been increasing when scaling up the number of messages being sent. The mean for RabbitMQ has gone from 2,1 for one virtual user to 10,31 for 50 virtual users. This is an increase of around 390%. By increasing the message load for RabbitMQ messages will on an average take longer time to be processed and be sent from K6

to Razor. For Swagger API the mean value changes from 1 to 2,24 when increasing the virtual users from 1 to 50. The same as RabbitMQ we can see an increase in the mean value when increasing the number of virtual users but only going up with 124% instead of 390%.

Swagger API has a lower mean value than RabbitMQ for all the treatments. At one virtual user the mean value of RabbitMQ is 110% higher than that of Swagger API. At 50 virtual users the difference is 360%. This shows that Swagger API tends to be more stable with message speed when the workload increases while RabbitMQ slows down significantly. When comparing the max values of both communication systems the values at a low workload are quite similar, but at 20 virtual users it spikes to almost four times that of Swagger API. The minimum value is consistent for Swagger API throughout the trials showing that even when the workload increases it can still perform as good for some individual messages. RabbitMQ instead sees a small increase for the minimum speed when workload increases.

7.3.2 Throughput

For latency we could see that Swagger API outperformed RabbitMQ, this was not the case for the throughput. When looking at message loss RabbitMQ had none, even when the workload increased RabbitMQ was stable and had no problems with transferring the messages. The moment the workload increased for Swagger API messages were lost in the transfer. At five virtual users, which represents 1500 messages, a loss occurred of 35 messages. In percent this would be a loss of two percent of all the messages. When increasing to 50 virtual users, which is 15000 messages, a loss of 92 messages could be observed. This would be around 0,6% of all the messages being sent. Swagger API does not have a consistent rise in the percent of messages being lost but still loses messages when the workload becomes too much for it. For throughput RabbitMQ outperforms Swagger API.

7.4 Conclusions

Swagger API has consistently a faster transferring speed of messages than RabbitMQ. The difference has statistical significance. Swagger API also has a lower minimum and max value for all the treatments, which means that the speed for RabbitMQ consistently is above and not fluctuating over the Swagger API speeds. For 50 virtual users the values had a sample range of 10,29 for Swagger API and 23,94 for RabbitMQ. This range can show us that Swagger API had more consistency during the trial run than RabbitMQ.

The goal of this thesis is to give an answer to the hypothesis: By changing solutions for data transferring from Message queues to REST API the performance should improve in transferring speed. The Anova test rejects the null hypothesis, and the result proves that for this setup and tools changing from the Message Queue RabbitMQ to a Swagger Rest API improves the transferring speed.

This thesis also went out to answer two research questions. For R1 the result shows that the difference depends on the workload, but Rest API always has a better time performance. With R2 the thesis wanted to check the reliability of the systems and if any message loss could occur. From the result we can conclude that RabbitMQ has a high reliability and saw zero message

loss through all the trials. Swagger API instead saw message loss when the workload increased, the percentage of messages lost was low and the reliability could be seen as acceptable.

8 Concluding Remarks

8.1 Summary

This thesis goal is to get a deeper understanding of what impact Message Queue and Rest API architectures have on the transfer speed of messages. Two research questions are asked, first about time performance and second about the system reliability. To answer these questions an experiment will be conducted.

The experiment concluded that Swagger API has a faster transfer speed than RabbitMQ. Five different treatments were used by communication system, for all of them Swagger API had a lower mean, max and min value than RabbitMQ. Two of the treatments, the one with the lowest number of virtual users and the one with the highest were tested for statistical difference using an Anova test. The Anova test concluded that the possibility for the results could not just be a coincidence. The mean value at 50 virtual users differed with 8,07 milliseconds. RabbitMQ had a mean of 10,31 and Swagger API 2,24 which is approximately a 4,6 times difference. The difference is significant and can be crucial depending on what the communication system shall handle.

This thesis also asked the question about the reliability of the communication systems. The experiment tested this by checking for message loss during the trial runs. RabbitMQ had no message loss during any of its trials while Swagger API had for everyone except the trial with only one virtual user. The percent of messages lost by Swagger API was under 1% and depending on situations and conditions this should be considered when building a message transferring system.

8.2 Discussion

Message communication between systems can grow in complexity when systems become larger and may need multiple tools to handle the message data. Having a complex artifact with a large size may produce unwanted factors which can influence the result. The artifacts created for this experiment are small and easy to use. A critique of the implementation is how the status received is handled. There is a slight difference between the systems where RabbitMQ measures from the Razor system and Swagger API makes its own measure. This critique is valid and could be a problem with this study's validity. Changing this would still probably create the same outcome but with a closer result.

The pilot study provided some important information and clarity regarding the artifacts. During the pilot a table was created on the razor pages with a row for every message, this could make the razor pages load slow. This discovery provided insight into how the front-end could affect the experiment which measured back-end processes. With this information the table was removed, and all message information was saved in internal memory. During the pilot study the K6 scripts lacked some modifiability for increasing the number of virtual users, this was updated in the code to use one variable that could be manually altered inside the code file. The pilot study also showed that both communication systems' first message being transferred had an irregular high speed compared to the rest of the messages. During the experiment this was also observed for all the trial runs and happened to the first message sent per virtual user. To make the result cleaner and more consistent these measure points were removed from the result and analyzed. As both systems saw the same spike, removing this should not impact on the result in a negative way.

In the experiment five different treatments were applied to each of the communication systems. Every treatment changes the number of virtual users sending messages and thus increases the number of messages sent. The result does not show how the system would handle the different number of messages if they were sent from only one virtual user. The treatment with one virtual user could prove that Swagger API has faster latency speed but the increasing difference between the two systems could have to do with the increase of users. Swagger API also had problems with throughput when the users increased, and we do not know from this study if the reason is the large number of messages or the increase in users sending messages. We also do not know the threshold for when Swagger API starts to encounter message loss.

The experiment and the different communication systems had a proven statistical difference. This difference between the systems was high but both still had a fast transfer speed. When the speed is under a certain threshold a user would perceive the speed as instantaneous, the thresholds for this was first explained by (Miller, 1968) and later confirmed by (Nielsen, 1994) to be at 0,1 seconds. The results from the study showed measure points that reached a maximum of 0,026 seconds. Because of this an experiment that measured human perception of the system would not be feasible. The speed is too fast for a human to perceive a difference between the systems, and the result would likely be made from placebo.

This thesis discusses the differences between the communication architectures Message Queue and Rest API but for the experiment only one type of each architecture is used. The decision to only use RabbitMQ and Swagger API made the scope of the implementation and experiment trials feasible for this thesis but leaves room for doubt considering the hypothesis and the research questions. The result from the experiment conducted by (Hong et al., 2018) showed that Rest API became slower than RabbitMQ when the number of users increased number their experiment, they used a larger set of virtual users but at 50 virtual users the difference between Rest API and RabbitMQ was minimal. This goes against the conclusion from this study where we could see that the differences between Swagger API and RabbitMQ increased when the number of virtual users increased. This could point to the setup and tools having an impact on the transferring speed and that it is important to disclose what has been used during the experiment.

8.2.1 Ethics

For an experiment and paper, it is important to follow ethics and disclose information that could affect the result and the conclusion. By ensuring that threats to validity are disclosed the experiment can get scientific integrity. This has been a point for this paper, the tools and how to install them are described for the experiments. Explanations for how to use the artifacts are also provided on the paper. To make sure to show full transparency a GitHub repository holding all the artifacts has been created and is open for the public, the address is provided in the reference chapter, and the commit number is written in the text for the Pilot study and the experiment.

Early architectural decisions are important when developing a new system. If a change of architectural level happens later in development the system could suffer, development might have to restart which can be expensive. In (Tofan et al., 2014) they state that architecture shall be created in early development phases and that architectural decisions are costly to change. Giving insight into what communication architecture has the best performance and to provide this information is the goal of this thesis and its paper. The information is meant to ease an architecture's decision making when selecting what communication architecture to develop.

This paper focuses only on providing insight into latency and throughput for the architecture, other reasons not measured in this experiment could favor one of the architecture types over the other. Earlier research has been done for Message Queue versus Rest API, but no study has been conducted testing RabbitMQ and a Rest API using Swagger as its framework versus each other.

All the artifacts during this experiment have been running locally on a client. In a real-world scenario this may not be the case and other factors like sending data between servers may yield a different result. The artifacts produced are also small in scale and larger, more complex systems may also change the result. The experiment also only tested RabbitMQ versus one Rest API using Swagger, meaning that the result and conclusion only can be made for these tools and not the entire extent of Message Queue versus Rest API.

The message data used are meant to reflect a real-world scenario where a message is sent to create a transport assignment in a system. The data objects are of type JSON, and every object has seven elements. Three of the elements are of type string and four of type int. Some of the elements change and some stay the same between the objects. Every virtual user iterates through the same JSON objects. A change that could be made is to use more than one set of JSON objects, by using multiple sets we could observe if the result is consistent through the sets and thus eliminate one threat to validity. Another change would be to randomize the order of the sent messages and use a larger sample of JSON objects. Both changes could potentially create a more accurate result but would probably not change the outcome and conclusion of the experiment. In the experiment there was a small difference between how Swagger API and RabbitMQ handled the message speed calculation. This has been disclosed in the paper, and even if the conclusion probably would have been the same, the experiment results may have been impacted by the difference.

When selecting a communication architecture sustainability and environmental aspects are also important to take into consideration. With a better latency the energy it takes to send a message may be lower and for a system that sends and retrieves a large amount of message data this could potentially be measurable while a smaller system may not see any energy saving. Something that needs to be considered when talking about the energy consumption is how reliable the communication system is. If the communication system is poorly made or the developers do not possess the necessary knowledge messages may not go through or the communication system stops working. In the case of a system malfunction the energy cost and worker recourses may exceed the savings from having a low latency. Therefore, it is still important to consider throughput and reliability when creating a communication system and the selection of the communication architecture. It is in most cases good to re-use already existing tools and to make sure the developers are confident in implementing a solution.

8.3 Future work

In this thesis we have checked the transfer speed and message loss of two communication architectures, but when choosing what architecture to implement more than these two parameters need to be taken into consideration. Scalability, maintainability and developer experience are three parameters that could be important to research. By implementing these metrics into the research from this thesis, more insight for communication architectures could be achieved and ease the decision making for which to implement into a system. For testing the architecture, one interesting study could be a case study where an in-production system was tested, this could also give insight into how the developers perceive the difference in developing the communication architectures. As stated, before this experiment only RabbitMQ versus Rest API using Swagger are being tested, to expand on this more research could be done testing different Message Queue and Rest API types like IBM WebSphere versus

Express API. With more testing of the communication tools a more definitive answer for the question which architecture between Message Queue and Rest API is the fastest. The tools used in this thesis are all open-source, would there be a difference in the result if non-open-source tools were used.

In this experiment we tested throughput using message loss but could not conclude where the thresholds exist for Swagger API. A continuation of this work could experiment more with message loss to find out exactly what the threshold for message loss is in Swagger API. This paper provides approximately the range for when the threshold appears and works as a basis for future work. The message size is fixed, and the same JSON objects are always iterated through in this experiment. In future works the same artifacts provided from this paper could be implemented but using different JSON objects and changing the algorithm for what order to send the objects in.

When working on this thesis limitations existed, both a time limit but also the number of available recourses. In the hypothetical scenario where no limitations exist a future work could investigate the actual cost, both for an automative production but also the impact on the environment regarding the two different communication types. In such a scenario a study could investigate how much recourses and time it takes to implement a REST API vs a Message Queue for a full-scale production system. This would later be measured against how much energy it takes to send a message and how often developers need to fix problems like message loss. By looking at both energy from developers but also the energy it takes to send messages one could get an answer regarding what communication type is most suited in an environment aspect. This could be an important aspect to keep track of as new environmental requirements are created by organizations like the European Union.

9 References

A. J. Poulter, S. J. Johnston and S. J. Cox, (2015). "Using the MEAN Stack to Implement a RESTful Service for an Internet of Things Application," *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, Milan, Italy, 2015, pp. 280-285, doi: 10.1109/WFIoT.2015.7389066, Institute of Electrical and Electronics Engineers Inc., New York City.

A. Sedaghatbaf, M. H. Moghadam and M. Saadatmand, (2021). "Automated Performance Testing Based on Active Deep Learning," *2021 IEEE/ACM 2nd International Conference on Automation of Software Test (AST)*, Madrid, Spain, 2021, pp. 31-40, doi: 10.1109/AST52587.2021.00010, Institute of Electrical and Electronics Engineers Inc., New York City.

Dotnet YouTube (2023). Dotnets official YouTube channel. Available online: https://www.youtube.com/playlist?list=PLdo4fOcmZooULFjxrOagaERVAMbmG20Xe

D. Tofan, M. Galster, P. Avgeriou and W. Schuitema, (2014) "Past and Future of Software Architectural Decisions – A Systematic Mapping Study, Information and Software Technology, vol. 56, no. 8, pp. 850-872, doi: 10.1016/j.infsof.2014.03.009, Elsevier B.V., Amsterdam, Netherlands.

F. Svahn, L. Mathiassen and R. Lindgren, (2017). "Embracing Digital Innovation in Incumbent Firms: How Volvo Cars Managed Competing Concerns," *MIS Quarterly*, vol. 41, no. 1, pp. 239-253, doi: 10.25300/MISQ/2017/41.1.12, University of Minnesota, Minneapolis-Saint Paul.

Grafana (2026). Grafana official webpage for K6 documentation. Available online: https://grafana.com/docs/k6/

J. L. Fernandes, I. C. Lopes, J. J. P. C. Rodrigues and S. Ullah, (2013). "Performance Evaluation of RESTful Web Services and AMQP Protocol," *2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN)*, Da Nang, Vietnam, 2013, pp. 810-815, doi: 10.1109/ICUFN.2013.6614932, Institute of Electrical and Electronics Engineers Inc., New York City.

J. Nielsen, (1994). *Usability Engineering*, Chapter 5, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

J. S. Dos Santos, L. G. Azevedo, E. F. S. Soares, R. M. Thiago and V. T. Silva, (2020). "Analysis of Tools for REST Contract Specification in Swagger/OpenAPI," *Proceedings of the 22nd International Conference on Enterprise Information Systems - Volume 2: ICEIS*, 2020, pp. 201-208, doi: 10.5220/0009381202010208, SciTePress.

M. J. Price, (2025). *Real-World Web Development with .NET 10 - Second Edition: Build Websites and Services Using Mature and Proven ASP.NET Core MVC, Web API, and Umbraco CMS*, Book.

Microsoft (2026). Microsoft official webpage for dotnet. Available online: https://dotnet.microsoft.com/en-us/

M. Niswar, R. A. Safruddin, A. Bustamin and I. Aswad, (2024). "Performance Evaluation of Microservices Communication with REST, GraphQL, and gRPC," *International Journal of Electronics and Telecommunications*, vol. 70, no. 2, pp. 429-436, doi: 10.24425/ijet.2024.149562, Polska Akademia Nauk, Warszawa.

M. Pinyagin and A. Sadovykh, (2025). "Automating Performance Testing in CI/CD - Tools Evaluation," *Testing Software and Systems: 37th IFIP WG 6.1 International Conference, ICTSS 2025*, Limassol, Cyprus, 2025, pp. 195-209, doi: 10.1007/978-3-032-05188-2_13, Springer Nature.

OpenAPI GitHub (2026). OpenAPI-Specification. Available online: https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.0.4.md

P. Dobbelaere and K. Sheykh Esmaili, (2017). "Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations," *Proceedings of the 11th ACM International Conference on Distributed Event-Based Systems (DEBS 2017)*, Barcelona, Spain, 2017, pp. 227-238, doi: 10.1145/3093742.3093908, Association for Computing Machinery, Inc., New York City.

Rabbit MQ Page (2026). RabbitMQ official webpage. Available online: https://www.rabbitmq.com/

RabbitMQ Stackoverflow (2017). Setup RabbitMQ consumer in ASP.NET Core application. Available online: https://stackoverflow.com/questions/43609345/setup-rabbitmq-consumer-in-asp-net-core-application

R. B. Miller, (1968). "Response Time in Man-Computer Conversational Transactions," *Proceedings of the December 9–11, 1968, Fall Joint Computer Conference, Part I (AFIPS '68)*, AFIPS Press, New York, NY, USA, 1968, pp. 267-277.

R. Maharjan, M. S. H. Chy, M. A. Arju and T. Cerny, (2023). "Benchmarking Message Queues," *Telecom*, vol. 4, no. 2, pp. 298-312, doi: 10.3390/telecom4020018, MDPI, Basel, Switzerland.

Swagger GitHub (2026). Swaggers GitHub repository. Available online: https://github.com/swagger-api/swagger-ui

Swagger io (2026). Swagger official webpage. Available online: https://swagger.io/docs/

T. F. D. B. Sarasola, A. García and J. L. Ferrando, (2024). "IIoT Protocols for Edge/Fog and Cloud Computing in Industrial AI: A High Frequency Perspective," *International Journal of Cloud Applications and Computing*, vol. 14, no. 1, pp. 1-30, 2024, doi: 10.4018/IJCAC.342128, IGI Global Scientific Publishing.

T. Stojanovski, M. Vučković and I. Velinov, (2012). "Empirical Study of Performance of Data Binding in ASP.NET Web Applications," *arXiv preprint arXiv:1201.0357*, 2012, doi: 10.48550/arXiv.1201.0357.

Thesis GitHub Repository (2026). Thesis project by a22wilas. Available online: https://github.com/a22wilas/Final-Exam

V. M. Ionescu, (2015). "The Analysis of the Performance of RabbitMQ and ActiveMQ," *2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER)*, Craiova, Romania, 2015, pp. 132-137, doi: 10.1109/RoEduNet.2015.7311982, Institute of Electrical and Electronics Engineers Inc., New York City.

X. J. Hong, H. S. Yang and Y. H. Kim, (2018). "Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application," *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, Jeju, Korea (South), 2018, pp. 257-259, doi: 10.1109/ICTC.2018.8539409, Institute of Electrical and Electronics Engineers Inc., New York City.