DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2021

Comparing Monolithic and Event-Driven Architecture when Designing Large-scale Systems

FELIX EDER

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Comparing Monolithic and Event-Driven Architecture when Designing Large-scale Systems

FELIX Eder

Degree Programme in Computer Science and Engineering
Date: June 21, 2021

Supervisor: Hamid Reza Faragardi
Examiner: Dejan Manojlo Kostic
School of Electrical Engineering and Computer Science
Host organization: Skatteverket
Swedish title: Jämföra monolitisk och event-driven arkitektur vid design av storskaliga system

Abstract

The way the structure of systems and programs are designed is very important. When working with smaller groups of systems, the chosen architecture does not affect the performance and efficiency greatly, but as these systems increase in size and complexity, the choice of architecture becomes a very important one. Problems that can arise when the complexity of software scales up are waiting for data accesses, long sequential executions and potential loss of data. There is no single, optimal software architecture, as there are countless different ways to design programs, but it is interesting to look at which architectures perform the best in terms of execution time when handling multiple bigger systems and large amounts of data.

In this thesis, a case called "The Income Deduction" will be implemented in a monolithic and an event-driven architectural style and then be put through three different scenarios. The monolithic architecture was chosen due to its simplicity and popularity when constructing simpler programs and systems, while the event-driven architecture was chosen due to its theoretical benefits of removing sequential communicating between systems and thus reduce the time systems spend waiting for each other to respond. The main research question to answer is what the main benefits and drawbacks are when building larger systems with an event-driven architectural style. Additional research questions include how the architecture affects the organisation's efficiency and cooperation between different teams, as well as how the security of data is handled.

The two implementations where put through three different scenarios within the case, measuring execution time, number of HTTP requests sent, database accesses and events emitted. The results show that the event-driven architecture performed 9.4% slower in the first scenario and 0.5% slower in the second scenario. In the third scenario the event-driven architecture performed 49.0% faster than the monolithic implementation, finishing the scenario in less than half the amount of time.

The monolithic implementation generally performed well in the simpler scenarios 1 and 2, where the systems had fewer integrations to each other. In these cases it is the preferred solution since it is easier to design and implement. The event-driven solution did perform much better in the more complex scenario 3, where a lot of systems and integrations were involved, since it could remove certain connections between systems. Lastly, this thesis also discusses the sustainability and ethics of the study, as well as the limitations of the research and potential future work.

Keywords

Event-driven architecture, monolithic architecture, domain-driven design, service-oriented architecture, organisational structure

Sammanfattning

Strukturen som system och program designas efter är väldigt viktigt. När en arbetar med mindre grupper av system så kommer den valda arkitekturen inte att påverka prestandan mycket. Men när dessa system växter i storlek och komplexitet så kommer valet av arkitektur vara väldigt viktigt. Problem som kan uppstå när mjukvarukomplexiteten ökar är väntandet på dataaccesser, långa sekventiella exekveringar och potentiell förlust av data. Det finns ingen optimal mjukvaruarkitektur, det finns oräkneligt många sätt att designa program. Det är intressant att kolla på vilka arkitekturer som preseterar bäst sätt till exekveringstid när en hanterar ett flertal större system och stora mängder data.

I den här avhandlingen kommer ett fall, kallat "Ingångsavdraget", att implementeras i en monolitisk och en event-driven arkitekturell stil och sedan köras igenom tre olika scenarion. Den monolitiska arkitekturen var vald på grund av dess enkelhet och populäritet vid utveckling av enklar program och system. Den event-drivna arkitekturen valdes på grund av vissa teoretiska fördelar, så som att kunna undvika sekventiell kommunikation mellan systemen och därmed reducera tiden som systemen väntar på svar från varandra. Den huvudsakliga forskningsfrågan som ska besvaras är vad de största fördelarna och nackdelarna är när man bygger större system med en event-driven arkitekturell stil. Andra forskningsfrågor inkludera hur arkitekturen påverkar effektiviteten hos en organisation och samarbetet mellan olika team, samt hur datasäkerheten hanteras.

De två implementationerna sattes igång tre olika scenarion inom fallet, där exekveringstid, antal HTTP-anrop skickade, databasaccesser och event skickad mättes. Resultaten visar att den event-drivna arkitekturen presterade 9.4% långsamare i det första scenariot och 0.5% långsamare i det andra scenariot. I det tredje scenariot presterade den event-drivna lösningen 49.0% snabbare än den monolitiska lösningen och avslutade därmed scenariot under hälften av tiden.

Den monolitiska implementationen presterade generellt väl under de simplare scenarion 1 och 2, där systemen hade färre integrationer till varandra. I dessa fallen så är den den föredragna lösningen eftersom det är lättare att designa och implementera. Den event-drivna lösningen presterade mycket bättre i det mer komplexa scenario 3, där många system och integrationer var inblandade, eftersom den kunde ta bort vissa kopplingar mellan system. Slutligen så diskuteras även hållbarhet och etik i studien, samt begränsningarna av forskningen och potentiellt framtida arbete.

Nyckelord

Event-driven arkitektur, monolitisk arkitektur, domändriven design, service-orienterad arkitektur, organisationsstruktur

Acknowledgments

I first and foremost want to thank my supervisors at Skatteverket; Arman Kurtagic, Erik Anderberg and Per Burman. You helped me at times where the project seemed to be impossible to complete and without your help I would never have completed this thesis in time, which I am very grateful for.

I also would like to thank my friends and family for their continuous support, through the tougher and easier times. I would especially like to thank Ann-Sofi Axås Eder, Emil Berg-Lundfeldt and Henrik Kälvegren for proofreading my first draft and helping make sure the final version could be of the highest quality possible.

Thank you all very much.

Stockholm, June 2021
Felix Eder

Contents

List of Figures

List of Tables

Listings

List of acronyms and abbreviations

API Application Program Interface

CLI Command Line Interface

CSV Comma-Separated Values

DDD Domain-Driven Design

DES Discrete Event Simulations

EDA Event-Driven Architecture

HTTP Hypertext Transfer Protocol

IAV Ingångsavdraget

IAVC Ingångsavdragscertifikat

JSON JavaScript Object Notation

LISA Line Information System Architecture

MDA Model-Driven Architecture

NPM Node Package Manager

RPC Remote Procedure Calls

SOA Software-Oriented Architecture

Chapter 1

Introduction

This chapter will introduce the subject of this thesis and the research question to answer. Firstly, some general background information is given about the subject, followed by a description of the problem area and the research question. The purpose and the goals of this study is then presented, followed by the chosen research methodology. Lastly, the delimitations of the thesis is shown followed by a general structural outline of the report.

1.1 Background

"Software architecture" is often a broad and confusing term with a lot of different kind of meanings and ideas intertwined. There are many ways to define it and there are multiple design philosophies regarding how systems should be designed[1]. The key software architectures that will be handled in this thesis is the monolithic architecture and the event-driven architecture (EDA). A monolithic architecture is a well-used architecture that consists of a single application layer, which is advantageous especially when designing relatively simple applications and is very common in cases like the one that will be looked at in this thesis[2]. Systems that follow the EDA style are decoupled systems that run in response to events. Events are defined as a change in the state or update of the program[3]. The main theoretical benefit of an EDA in this case is that it removes the need for sequential communication between systems and replaces it with emitted events[4]. An EDA applied in this specific case could potentially remove the need for systems to wait for each other to complete tasks and could thus reduce the main execution time[5]. Therefore is it interesting to compare a monolithic architecture and an EDA in this specific case. Chapter 2 describes the realted background in more detail.

1.2 Problem

The way the structure of systems and programs are designed is very important. When working with smaller groups of systems the chosen architecture does not affect the performance and efficiency greatly, but as these systems increase in size and complexity, the choice of architecture becomes a very important one. If the architecture these systems are built with is not carefully chosen and evaluated, problems that can arise when the complexity of software scales up are waiting for data accesses, long sequential executions and potential loss of data. There is no single, optimal software architecture, as there are countless different ways to design programs, but it is interesting to look at which architectures perform the best when handling multiple bigger systems sending large amounts of data between each other in order to perform certain tasks.

In this thesis, a specific case will be implemented in both a monolithic architecture and an EDA, and the two solutions will then be compared in terms of performance and efficiency. The monolithic architecture was chosen because of its simplicity and popularity when building programs, and the EDA was chosen because of its potential to fix or improve upon the previously mentioned issues such as slow sequential communication and slow data access. No similar studies has been made comparing these two software architectures in a case with multiple large systems with many data integrations, and it is therefore interesting to research in this thesis.

There are also other potential problems when working with large systems, and one of them is the organisational structure and team cooperation. If a single team is responsible for the development of a system, the amount of connections and integrations this system has with other systems could cause inefficiencies in work procedure. For example, if one team is updating their system in a certain way, perhaps other teams needs to update their systems as well in order for the integration between the systems to work properly. The choice of architecture when designing this set of systems could affect how these systems integrate with each other and could therefore affect the efficiency and cooperation of the development teams. This aspect of the research is qualitative in nature, as it is difficult to measure this exactly. Since this is mainly a quantitate research, the question of organisational structure and efficiency when comparing software architecture will only be a minor part of the study.

Another problem related to software architecture and the construction of larger systems is the question of security. Security is a very broad term, but in this context it is about the way data is handled. The systems in this thesis all

handle extremely sensitive personal data and it is of the highest priority that this data does not get intercepted by third parties or leaked from the systems. Due to the scope of the thesis, the question of security will only be a minor one that will be explored if there is enough time after the bigger questions have been answered. This would be a quantitative study where the amount of data leaked or intercepted by third parties could be measured.

1.2.1 Research Question

The subject to be researched and question to be asked in this thesis is quite broad and has therefor been divided up into a few smaller ones.

1. What are the benefits of introducing an Event-Driven Architectural style to an ecosystem of large-scale systems?
    (a) What are the main drawbacks and consequences of doing so and how can they be handled?
    (b) What are the effects on the organisation in terms of work procedure and cooperation between teams?
    (c) How will this architectural style affect the security of the systems?

1.3 Purpose

The purpose of this thesis is to construct a set of systems in two different software architectures, a monolith and an EDA. They will then be compared in terms of performance and efficiency when put through various scenarios within a given case. The effects on the organisation and cooperation between software teams as well as the security that these different software architectures provide will also be looked at if there is additional time.

The host organisation that this thesis will be performed at is Skatteverket, the Swedish Tax Agency. Skatteverket is a government administrative authority with a deep and long tradition of digital handling of taxation and civil registration. That means that they have an IT environment consisting of many generations of platforms, frameworks and architectures with a deep legacy of batch-based systems and Remote Procedure Calls (RPC), mixed with Software-oriented architecture (SOA) as well as microservices in modern container platforms. They have many older monolithic systems that all have their own replicated data and, therefore, need to synchronise it when changes are made to it. This all amounts to systems with high coupling between them,

which is hard to scale and is not very efficient. They are interested in what benefits and drawbacks an EDA could provide for their large systems, as well as the effect on organisational structure and security of data.

1.4 Goals

The main goal of this project is to find out what the benefits and drawbacks are of implementing a set of large systems in an EDA style, compared with an monolithic style. The comparison will look at metrics such as execution time, number of database accesses, events emitted and HTTP request sent, in order to determine which software architecture is the best fit when working with larger systems.

Additional goals include looking at how the organisational structure and team cooperation will be affected by the choice of software architecture, as well as how the security of the data of these systems is handled.

1.5 Research Methodology

The research methodology consists of a few different steps, as can be seen in Figure 1.1. Firstly, the existing problems will be identified, based on the information given by the host organisation and what their goal with researching the subject is. When the problem has been identified, the research question will be formulated and checked with the host organisation in order to make sure it aligns with their vision for the project. Then the pre-study will start, which is a process where earlier works within various related areas will be researched and presented, in order to establish what the current state of the art within the field is. Based on the state of the art, the chosen methods and solution in order to perform the study might be updated in order to reflect the pre-study findings. Then the design and implementation of the solution will be performed, which will be an iterative process with a feedback loop from the host organisation in order to make sure that the solution that is designed ensures a certain level of authenticity. After the solution is satisfactory, data will be generated and put through the systems in order to simulate the various defined scenarios and data will be collected. When sufficient levels of data has been collected, it will then be analysed and displayed in various tables and graphs. The results will then be discussed and conclusions will be drawn and presented.

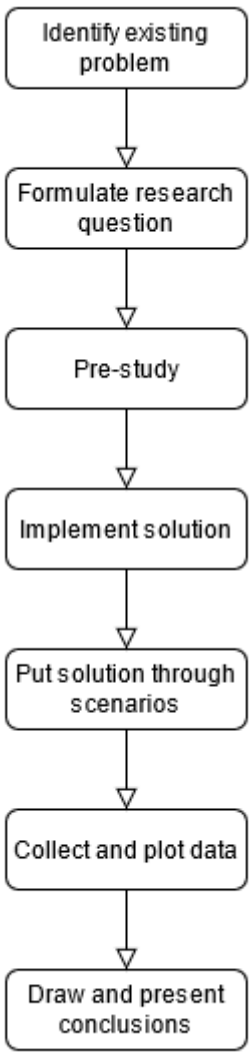

Figure 1.1: The research methodology of this thesis.

1.6 Delimitations

The scope of this project will be limited to the given case provided by the host organisation. Projects regarding system architecture can grow quite large since it is a very broad topic that can be hard to put a limit on, and therefore only a central system as well as a handful of helper systems will be designed and implemented. There are a lot of ways a system ecosystem like this can be measured and evaluated, a lot of metrics that can be logged in any number

of places. Therefore only certain scenarios in the process, that are of special interest, will be defined and looked at in detail. This will keep the results as well as the following discussion and conclusions within a reasonable size, and also make it easier to categorise the data and structure of the discussion section of the report. Following a specific case is also of interest for the host organisation, as they work on similar cases and a more focused approach on a specific case would be more useful for them than a more broad and abstract research.

Another delimitation is the perspectives the results of the solution will be looked at from. The primary interest is performance in terms of execution time and various actions performed, and thus other metrics that could be evaluated will not be part of this study. When it comes to the fallacies of distributed computing, due to lack of time and complexity, only availability will be a measured part of this study. Availability was chosen since it relates the most to the other results collected in this study of the fallacies and thus makes the most sense to collect and analyse together with the rest of the data.

1.7 Structure of the Thesis

In chapter 2 the fundamental parts of the background of the subject will be presented in-depth in order to prepare the reader for the methods and further analysis of the collected data. In chapter 3 other works within related areas of research will be presented in depth in order to establish the current state-of-the-art of the area. In chapter 4 the methods, research process and overall design, implementation and data collection will be presented. In chapter 5 the results of the solution will be presented in various tables and graphs, divided up based on the scenarios that the data was collected from. In chapter 6 the methods and results will be discussed in depth, as well as a reliability and validity analysis and a general reflection about sustainability and ethics. In chapter 7 conclusions will be drawn based on the discussion from the superseeding chapter. A section about the limitations of the study and future work will also be presented.

Chapter 2

Background

In this chapter the most fundamental parts of the subject will be presented at an in-depth level in order to prepare the reader for the methods and further analysis of collected data.

2.1 Software Architecture

The term "software architecture" is a broad and confusing one. It means a lot of different things to a lot of different people and is particularly hard to define exactly. The renowned software architect Martin Fowler says that software architecture is too fundamental to define, but would rather explain it as "the shared understanding that the expert developers have of the system design"[1]. He also adds that software architecture is the set of design decisions that needs to be made early on in a project, or rather, the design decisions that you wish that you had made early on in a project. In general though software architecture is said to be an abstract explanation of how a system behaves, including which components it consists of and how they integrate with each other. Software architecture does generally not concern itself with the technical implementation of the system[6].

2.2 Monolithic Architecture

Monolithic architecture is a widely-used and popular architecture when building software applications. According to Microsoft documentation[7], systems following the monolithic architectural style consist of a single application layer that supports the user interface, the business logic as well as the data manipulation. The data could be stored persistently in a separate location, but the logic

to access the data is stored in the same application layer. Microsoft presents Microsoft Word as an example of a monolithic application, but the architecture style is also commonly used in web development[2].

Chris Richardson says that[8] a monolithic approach is advantageous when a team is developing a relatively simple application that needs to go to market quickly. It is relatively easy to deploy since it only consists of a single file or directory hierarchy. A lot of the developer tools (IDEs etc) are also built with monolithic architectures in mind and are therefore easy to use.

There are some drawbacks with using a monolithic architectural style Richardson writes, which start to arise when the application starts to increase in complexity. As the code base of the applications grows bigger, it becomes harder for new developers to understand the code and start contributing to it. The size of the application also becomes a problem if the developers practice continuous deployment, since each small release or hotfix means re-deploying the entire application, which could be a slow and costly operation. Other issues that could arise is the problem that a monolithic application is commited to a single technology stack and it is hard to change or update it since that would mean doing changes to the whole code base. Single layer applications could also have problems with performance as the complexity increases as because everything is run as a single application.

2.3 Domain-Driven Design

Domain-Driven Design (DDD) is a set of principles and practices for how software development should proceed. It is an abstract and large topic that covers a lot of different things. According to Scott Millett in his book "Patterns, Principles, and Practices of Domain-Driven Design", DDD can be distilled into the thought that every subject area that you are building software for has a problem domain. The people who are working on a specific problem domain become experts within that specific problem domain in order to create as good a product as they can for the subject area[9, p. 6-7]. The complexities of the design is then based on a model that fits the subject area and a creative collaboration between the technical and domain experts is initiated in order to fit the problem domain and model ever closer to the subject area. DDD is a complex promise that starts out simple but quickly increases in complexity as the real world problem that it tries to model becomes ever-increasingly complex[10].

2.4 Microservices

A microservice architectural style is today a relatively well known way to structure a larger scale system. Martin Fowler again writes that there is no precise definition for microservices, there are certain common characteristics around organisation and business capacity, automated deployment, intelligent endpoints and decentralised control of languages and data[11].

In most basic terms, Chris Richardson writes, a microservice architecture is an architectural style that structures an application as a collection of services that are loosely coupled, highly maintainable and testable, independently deployable and organised around business capabilities. By following a microservice architecture, it enables a team to make rapid, frequent and reliable deliveries for large, complex applications[12].

It is useful to compare a microservice architectural style to a monolithic architectural style, Fowler writes. As mentioned in section 2.2, monolithic architectures become problematic as the complexity increases and the application becomes harder to maintain. Microservices became a popular response to this problem, as they enforce building applications as suites of services. Since they are independently deployable and scalable, they can be written in different technology stacks as well as be managed by different teams, which solves a lot of the most common issues with monolithic architecture[11].

2.5 Event-Driven Architecture

Event-driven architecture (EDA) builds on the foundation of microservices, it is a decoupled system that runs in response to events. It uses events in order to trigger and communicate with other decoupled services within the same application and is therefore a good fit when extending the microservice architectural style[13]. An event in this case can be defined as a change in state or an update. In the case for Skatteverket, an event could be that a fee has been paid or a declaration is made. An EDA has three main components: event producers, event routers (also called event bus) and event consumers[3].

An event producer detects an event and represents the event as a message and is transmitted through the event router to the event consumers. The producer does not know what the event itself means or who the consumers of the event are. The event is then sent from the event router to the subscribed event consumers and they are immediately informed of the event. The affected consumers will then either process the event or may only record it. EDA in

practice means that systems, services and components do not need to communicate with each other directly, instead they simply react to the events that are sent through the system. This is why an EDA enables a system to have a higher degree of decoupling[5].

2.6 Event Sourcing

Event sourcing is a design pattern that can be used together with an event-driven architectural approach, where the events themselves are stored in a database instead of the state of the application. By doing this, the developers can safely assume that they are saving each state atomically since they can reproduce it by re-applying all the different events up until a wanted state[14].

2.7 Event Streaming

Event streaming (also called event stream processing) is the practice of taking action from data inputs (events) coming from a continuous source. With the rising popularity of microservices and event-driven architecture, event streaming has become a popular addition to the architectural style. In event streaming, events are continuously and immediately processed and sent to the relevant subscribers and can be compared against batch-based processing, where the data is processed in batches instead of in a stream. Event streaming is useful when the events detected by the system needs to be handled with as little latency as possible. Examples of this can be online payment services, maintenance systems and fraud detection[15].

2.8 Simple Event Processing

Simple Event Processing is a simple concept as it only concerns itself with small, system-wide events sent on infrequent times. These events are usually state-changing and can propagate through a lot of different applications, initation further downstream action in a system[16]. An example of a simple event could be that some sort of information has been entered into the system, which emitts an event and triggers a process that then starts to analyse the data and perform further tasks on it. Simple Event Processing is useful for improving real-time work flow, as it removes the need for sequential communication and could reduce latency and cost of processes[17].

2.9 Event Storming

Event storming is a flexible workshop format that is designed to find out what is happening to complex business domains. It can be used for different goals and in different scenarios, but is mainly used for assessing the health of an existing line of business, exploring new viable startup business models, envisioning new services and designing clean and maintainable event-driven software. Event storming was developed in the context of domain-driven design and can be specifically useful when developing an EDA system since it enables the developers to define the events from the start. The process itself consists a few different steps and is preferable to do physically, with a big wall and sticky notes of different colours. Firstly the domain events of the system are defined, as they are the most important in the system, and then a variety of processes, views and commands are added to the structure. This structure is visualised by the multi-colored sticky notes attached to the wall, an example of this can be seen in Figure 2.1 Since the workshop starts with discussing the main events of a system, it can be a useful starting point when designing an EDA[18].

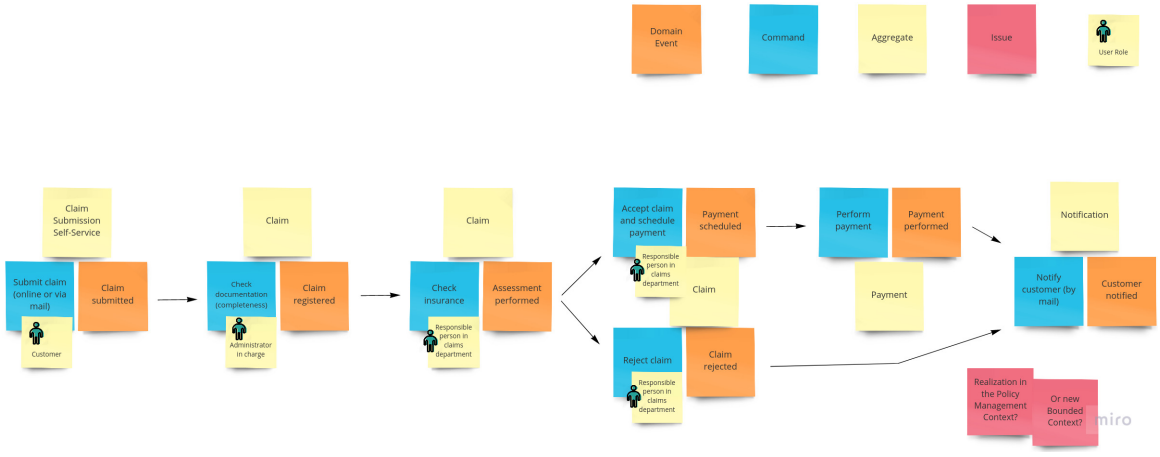

Figure 2.1: An example of an event storming board.

2.10 Conway's Law

Melvin Conway, a computer programmer, formulated an adage in 1967 that says: *"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure."*[19]. This concept became known as "Conway's law" and has been found

to be the case in many software companies in the following years. In a study by MacCormack et al, multiple codesbases were analysed to see if Conway's law applied. Their results showed that product focused teams tended to have monolithic codebases with high coupling, while open source projects had more modular and loosely coupled codebases[20].

2.11 Inverse Conway Maneuver

In section 2.10 it is stated that Conway's law makes organisations design structures that are a copy of its communication structure, which can cause some unintended consequences. The Inverse Conway Maneuver is a counter-measure for this and argues that the team and organisational structure should be evolving around the software structure in order to promote the desired architecture[21].

Chapter 3

Related works

In this chapter related works similar to the subject of this thesis are discussed in depth. This section is divided into multiple subsections around specific types of EDA approaches. Lastly, the findings are summarised and the research gap is presented.

3.1 Event Sourcing

Papers that have researched event sourcing will be presented in depth in this section. Event sourcing is the practice of instead of storing the complete state of an application, record all the state-changing events[22]. In order to recreate a given state of the application, simply re-apply all the previous events to the state until the wanted state is recreated. Event sourcing can be used when building a set of systems following an EDA style, as it is a natural fit since both use events as a central component. Event sourcing and an EDA do however work well on their own and there is no need to combine them in order to be used efficiently[14].

Benjamin Erb et al. took a closer look a Discrete Event Simulations (DES) and event sourcing, in order to see if there was a way to combine them efficiently for scalable, distributed applications. They found out that DES and event sourcing could benefit each other in a few different ways. One example of this is the event log and ability to restore any state from event sourcing, a feature that is very valuable for DES, since restarting a simulation from scratch each time is both cost and time intensive. They conclude the study by stating that a hybrid architecture looks promising, but that there still are some issues with it that needs to be researched in the future before it becomes a viable solution[23].

Event sourcing is a very interesting area within the general subject of EDA, as the idea of storing the state of the application through event history is beneficial for a potential solution. However, since the states used in this thesis are not very large, event sourcing will not affect the performance greatly. Since perfomance is the main metric of the study, event sourcing will thus not be used in this thesis.

3.2 Event Streaming

In this section works within the related field of event streaming will be presented. Event streaming is a type of event process management that can be used in conjunction with an EDA system. If an EDA were to be combined with event streaming, it would provide a set of systems that can respond to a lot of moving data and make quick decision. This combination can be useful if one were to develop a set of systems handling a lot of data and where decisions have to be made fast, such as in cases involving machine learning[24].

The difference between event streaming and the type of EDA systems that will be researched in this paper (called business-oriented EDA) is that in event streaming, events are continuously sent through the system, while in business-oriented EDA system the events are more infrequent and usually span the entire ecosystem of applications[15].

In [25] Ovidiu-Andrei Schipor et al. thought that all the new smart devices that we use today have a lot of inputs and events that our current software architectures are not particularly adept at handling efficiently. They introduced EUPHORIA, an event-driven architecture implementation that should enable fast easy prototyping and deployment for flexible interactions between heterogeneous devices. The system is built with four different quality attributes in mind: adaptability, modularity, flexibility and interoperability. They put their implementation through a few different scenarios to specifically test message-size and environment-complexity on response times in the system. Their results showed that EUPHORIA delivered average response time under 150 ms for applications that handled large messages sent between a large number of producers and consumers. They concluded that EUPHORIA would help connect the Application Program Interfaces (API) of different smart devices and enable more interesting data to be collected and analysed, but more work is still needed within the field in order to perform more complex tasks involving more smart devices.

In [26], an EDA style was implemented for a decision support system used for Traffic Management. The problem they encountered was that the system

needed to cope with a high volume of continuously generated events and could not handle them all efficiently. Their research method was in performing a case study where they illustrated their approach by modelling it after an EDA system, where they defined their own types of events and then implemented it in an event-processing engine named Esper. Their results showed that the EDA framework can be used to greatly improve the real-time features of such a system as the one they were researching.

In [27], a group of people were trying to develop a smart city solution using an event-driven architecture, focusing on monitoring public spaces. They developed a concept for an architecture and then implemented it in a testbed focusing on a subway scenario. They defined their own event managers, started to aggregate data and set up a chart for how the events should be sent throughout the testbed. In the end they concluded that the proposed architecture that they implemented in the testbed made the integration of their sensors and systems easier and more efficient due to the event-driven approach.

A separate paper from 2017 wanted to improve the manufacturing systems of the industry and presented an EDA based architecture called "Line Information System Architecture" (LISA). The focus on it was on integration of devices and services, which simplified the hardware changes and smart device integration. LISA proved to be very efficient and is now being installed at a large automotive company[28].

Event streaming is a very important area within EDA design, and as these reports have shown, it is used a lot in various types of configurations. However, event streaming uses continually streamed data that needs to be reliable and predictable in order to work properly. The systems involved in this study will have events sent at random times, and thus event streaming might not be appropriate for this research.

3.3 Simple Event Processing

In this section EDA system that mainly uses Simple Event Processing (also called Business Events) will be presented.

In [29], the authors wanted to find out what the best kind of architecture would be useful when building a support system for handling a crisis scenario, in this case a realistic flood scenario set up by official French services. The study was mainly a literature review where a few different types of software architectures were discussed and then compared to what the case at hand demanded. In the end they settled on a mix between an EDA and a Model-Driven Architecture (MDA) to best solve the presented issues.

In 2016, Jorge Veiga et al. implemented a new MapReduce framework using an EDA approach. MapReduce applications are common within organisations that want to analyse big quantities of data. As the amount of data increases, so does the computation time and thus a more efficient implementation of the MapReduce paradigm was needed. The authors implemented "Flame-MR", an EDA framework that was based on the popular Apache Hadoop framework. In order to evaluate it, they performed various performance tests for common tasks, like sorting a lot of data or searching for a specific piece of information. In the end they concluded that Flame-MR can improve the performance and scalability of I/O bound workloads (like sort and search), reducing execution times by up to 34% compared to the Apache Hadoop framework.[30]

In [31], the authors wanted to research how concurrency issues were affecting server-side EDA systems by implementing a fuzzing library for Node.js. They started their research by performing a thorough bug characteristic study of real world open-source event-driven applications, built with Node.js. Based on the results they got they implemented a fuzzing test aid library for server-side event-driven programs called Node.fz. They used this software in various server-side EDA programs and managed to recreate common problems that occurred in EDA systems of this kind faster than before. The software also managed to expose new bugs in the programs that were tested, while not affecting the performance notably.

Simple Event Processing is the most promising approach for this thesis, as it handles simple state-changing events that are sent on irregular intervals. The problems with the findings in this section is that they implement an EDA in small systems, usually only containing a program or two. This thesis wants to look at an EDA within the context of large-scale systems.

3.4 Summary

In this section, the related works will be summarised and it will be explained how it relates to the subject of this study. The research gap found in the research will be stated and it will be explained how this thesis will fill that gap.

As can be seen, the subject of EDA is very broad and contains a wide variety of topics. The topics of Event Sourcing, Event Streaming and Simple Event Processing were chosen to be looked at in detail, as they were of the most interest for the subject. From studying other related works, it is shown that Event Sourcing is a very interesting subject and can be extremely useful when constructing an EDA. However, it will not affect the performance of the systems, and will therefore not be relevant for the implementation. Event

Streaming is a big area with a lot of applications within the area of EDA. Since it needs the events to be emitted constantly and on regular intervals, it is not a great fit for this problem area. Simple Event Processing is the most relevant area that was studied during the research phase, as the events emitted are simple, state-changing on irregular, which is needed for this study. However, all the relevant research found within the area mostly look at simple systems and programs, which is not quite the scope of this study, and thus a research gap is discovered.

3.4.1 Research Gap

The gap in the research that was studied is that there does not seem to be much research done with Simple Event Processing within the context of large systems, like the systems that the host organisation Skatteverket is developing. It would be interesting to see if and how an EDA using Simple Event Processing can improve the design and performance of large collections of systems in terms of execution time and data sent. This gap will be tried to be filled by the contributions of this thesis.

Chapter 4

Methods

In this chapter the research process and the methods used to perform the research will be presented. The design of the systems will be presented in detail along with the different processes within them. The scenarios within the systems to be measured will also be explained. Lastly, the data handling and planned analysis will be presented.

4.1 Research Process

The goal of this research is to perform a quantitative study comparing two different architectural approaches to solve the same case (see section 4.2 for more details on the case). The main metric to measure is execution time in milliseconds, see section 4.4 for more information about the metrics for each scenario. The two different architectures to compare is a monolithic approach and an event-driven one. The first part of the research process was thus to go through the requirements of the case and then design a monolithic solution for this problem. After the design was approved, the monolithic solution was implemented. The implementation was iterated through many different versions until it matched the given case in an appropriate way. Data was then generated for the solution and put through it (for more information about the data used and examples, see section 4.5). The solution was constantly tweaked and improved as issues arised and needed to be fixed.

After the metrics from the monolithic approach were fetched and stored, the design of the event-driven approach started. The main events of the system were identified and the flow of the system was updated in order to accommodate for the new event-driven approach.

After both implementations were done, they were tested in various scenar-

ios with the generated data and then tweaked and improved upon if issues or inefficiencies were found. In the end the data was recorded and then analysed.

4.2 Case

In order to properly test these two architectures in question, a specific, real-life case at the organisation was selected to be studied, called "The Income Deduction" ("Ingångsavdraget" in Swedish) (IAV). IAV was proposed by the Swedish government in order to make it easier for lower income people to get employment by lowering the costs for the employer for the specific person as well as other benefits.[32] If a person matched a set number of requirements, they would be issued a certificate (IAVC), stating that they are eligible for the IAV discount. Each month, as new income information would be recorded in the system, each active IAVC would be checked in order to make sure the person has not earned too much money. If their income was higher than a certain threshold, the certificate would be marked invalid and the person would no longer be eligible for the discount.

The proposal itself was supposed to be implemented by Skatteverket, but was then later rejected for various reasons, and thus Skatteverket had started designing the system but never did implement it. Therefore, IAV is an interesting case study when comparing architecture styles within a context of large scale systems since the case consists of many different systems and data flows as well as both internal and external integrations with other programs and systems.

4.3 Design and Implementation

The IAV case was broken down into smaller parts, including what data it needed to handle, which external systems it needed to interact with and, of course, how its internal components would work. The final design that was approved can be seen in Figure 4.1.

Figure 4.1 details the topology of the systems designed, divided into three different types. The color red indicates a database system, a system which main task is setting up and connecting to a database and provide data to other systems through various integrations. The color blue indicates a calculation system, a system that receives some sort of data, performs a task on it and then performs one or more actions based on the result of the calculation. The color green indicates a communication system, a system that is responsible for

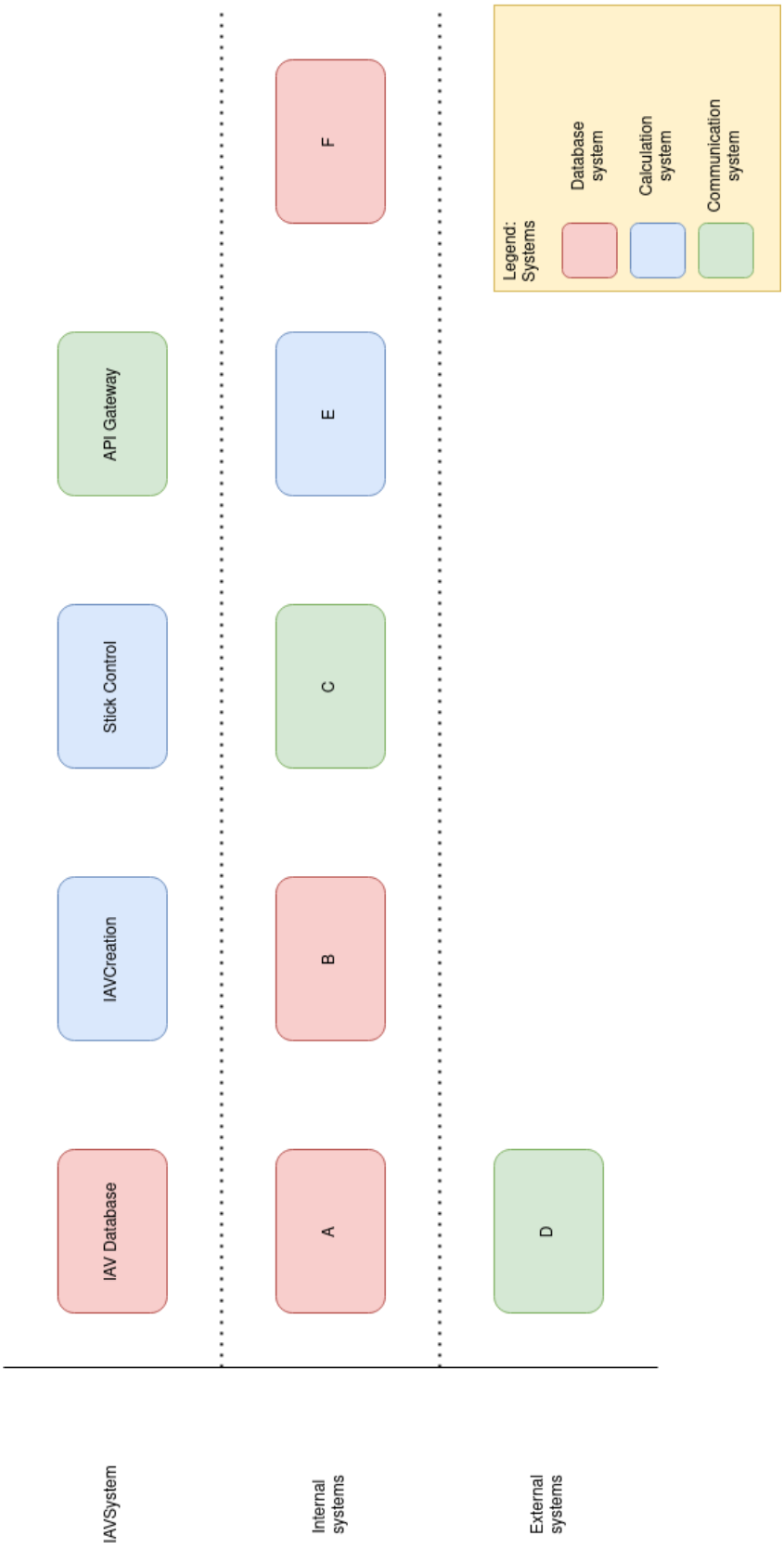

Figure 4.1: System overview

simply communicating with other systems, relaying data to external parts or other similar tasks.

The basis of the solution is the central system, called the "IAVSystem". This is the system that bears the responsibility of performing the main task of the system, determining who will get an IAVC and updating the information as more monthly income is sent to the organisation. It has its own API gateway that can route incoming requests to the relevant system, its own database to handle its own data, as well as two internal calculation systems, called "IAVCreation" and "StickControl". The details of what these systems do are not important for the research, they both perform various calculations and tasks that relate to the issuing and maintaining of IAVC.

Other than the IAVSystem, there are six other minor systems that simulate both internal and external systems at the organisation. Systems A and B have their own database that handle various personal information, while system C is responsible for sending out information to external parts. System D simulates external employers that send in payslip information to the internal systems. Systems E and F handle incoming payslips, performing various minor tasks and checks with the information as well as storing it in a database. Most of these systems are based on real systems at Skatteverket in to make the simulation as authentic as possible.

The systems were mainly written in Kotlin, with the relational database PostgreSQL as well as other minor third-party packages to help with the HTTP and database connections.

4.3.1 Monolithic Implementation

The monolithic implementation of the case study was designed to be as simple as possible. A central, monolithic service would be responsible for setting up local databases, exposing endpoints and starting processes. This central application would also be responsible for calling other services in order to get more data or send out information. It would use strategies like requesting and reading sequentially from batch files, have scheduled process runs as well as mainly communicating using HTTP requests.

The process flow for the monolithic implementation can be seen in Figure 4.2, which builds upon the previously established systems topology. There are five different kinds of integrations between systems and other parts, indicated by the colors of the arrows. Blue arrows represents a batch file, which means that a request is sent to a system to get a large quantity of data. The system then reads from its own database and generates a large file (using

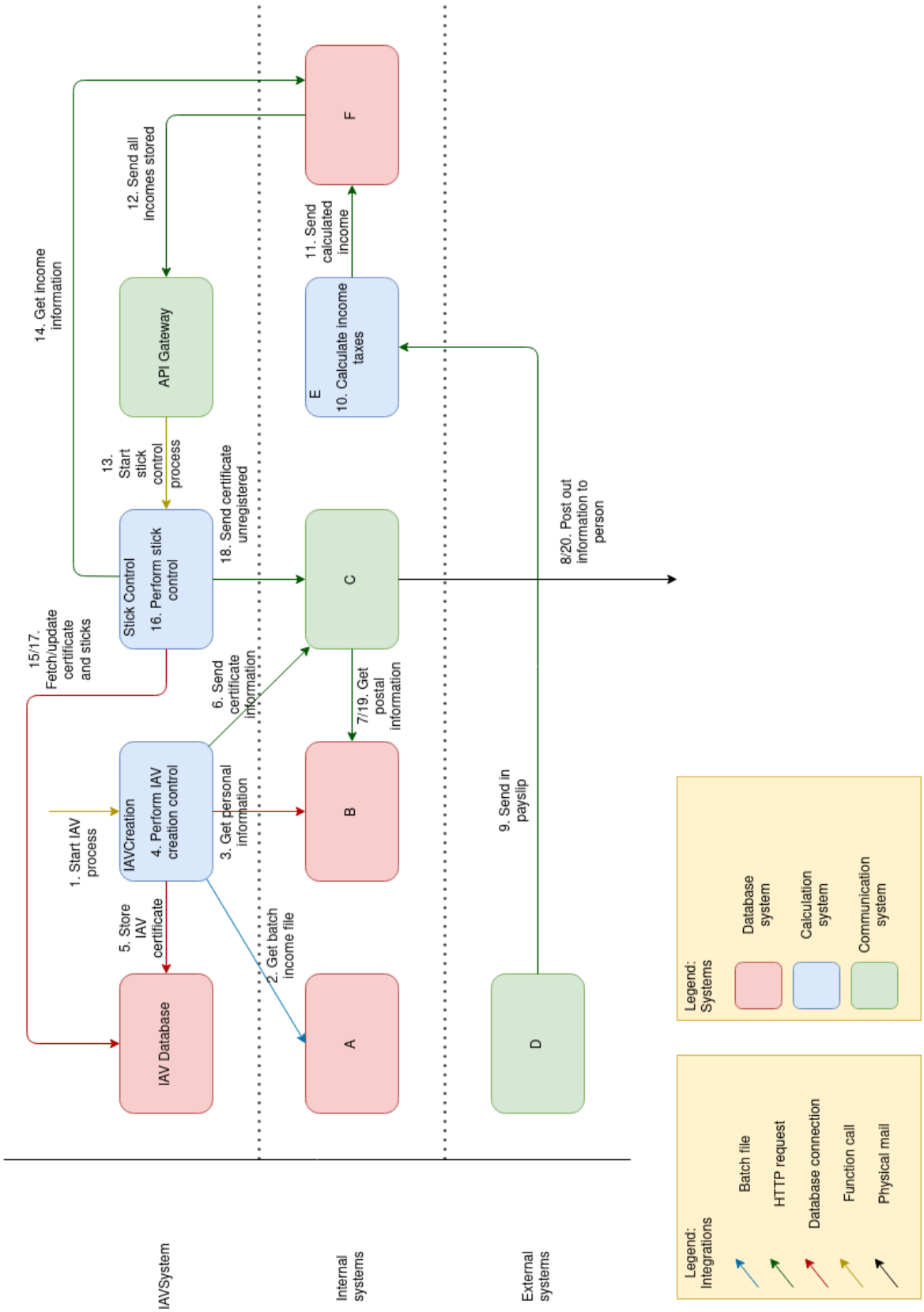

Figure 4.2: Monolithic process flow

JavaScript Object Notation (JSON) in this simulation), which is then stored in a general space that both systems can access. The receiver system then responds to the request with the id of the batch file and the original system can then start reading the data. Green arrows indicate a HTTP request (only GET and POST requests are used in this simulation). Red arrows indicate database reads (through an ORM-integration with the various databases) and yellow arrows indicate integration through a simple function call (only used within the IAVSystem). Black arrows indicate physical mail, which in this case only means sending out information to a specific individual (thus the arrow does not go to any other system), but in this implementation physical mail is replaced with a simple console print that achieves the same result.

The data flow of this solution can mainly be divided up into two processes. The first process handles the creation of the IAVC while the second process handles incoming payslip information and performs checks to see if the certificates are still valid. The complete source code for the monolithic implementation can be found at this public Github repository[33].

IAV Creation Process

The first process is started internally in the IAVSystem (step 1 in Figure 4.2), which fetches the information it needs from systems A and B and performs an IAV creation control. For the people that are eligible for IAV, a certificate is created and stored in the IAV database and then sent to system C. System C handles the posting of information to external parts, and thus fetches postal information from system B and then sends out the information to the relevant people. This is a scheduled process, and this means that it will run at a set time within a certain interval. In the case of the simulation, the process was run whenever new data needed to be collected for it.

IAV Control Process

The second process is started by system D, the only external system in the simulation and will act as a group of employers. The system can simulate any number of employers and number of employees for each one. Each employer will have a pre-generated set of employees, where about 10% of them have an IAVC already (and about 3% will falsely be marked as having one). Within a certain interval (representing a month, hereafter called a payslip period), each employer will send in a payslip detailing what each employee has earned in the latest payslip period to system E through HTTP. System E will perform a small

task for each payslip (representing calculating taxes) and then send along the information to system F, which will store it in its database.

System F knows how many employers need to send in payslips, and when the last payslip has been stored in its database, it sends a request to the IAV API gateway that all payslips for the current payslip period have been stored in the database. The API gateway tells the Stick Control system to start a complete stick control process. Stick in this context is simply a way to record the income of a person month to month and is used to track how close a person is to have their certificate removed from the database. Sticks are also stored in the IAVDatabase. After the stick control process is started, all valid certificates from the IAV database are controlled, fetching the respective persons income for the payslip period through an HTTP integration to system F. Based on the new income, the sticks are correctly updated in the database, and if a person would have gotten too many sticks and thus not longer eligible for the certificate, it will be removed from the database and information regarding this is sent to system C, which performs the same task as in the first process.

4.3.2 Event-Driven Implementation

The event-driven implementation of the case study was built after the monolithic one was completed and results were extracted from it. The initial design of the system was developed through a series of online event storming sessions. Based off of the monolithic design, different kinds of events in the process were identified and added as connectors between producers and consumers. The producers and consumers in this case were based on the services created in the monolithic approach. For each event, a descriptive name as well as data attributes were listed and thus an initial design was created. The update process flow can be seen in Figure 4.3.

A key difference in the event-driven process flow is that the batch integration has been replaced with a new event integration, marked with purple arrows. The system A has become an event producer, systems C, Stick Control and API Gateway have become event consumers while system E, F and the IAVCreation act as both producers and consumers. The two major processes from the monolithic architecture remain mainly the same, but with some noticeable differences. The events themselves were implemented using HTTP POST requests between systems. The complete source code for the EDA implementation can be found at this public Github repository[34].

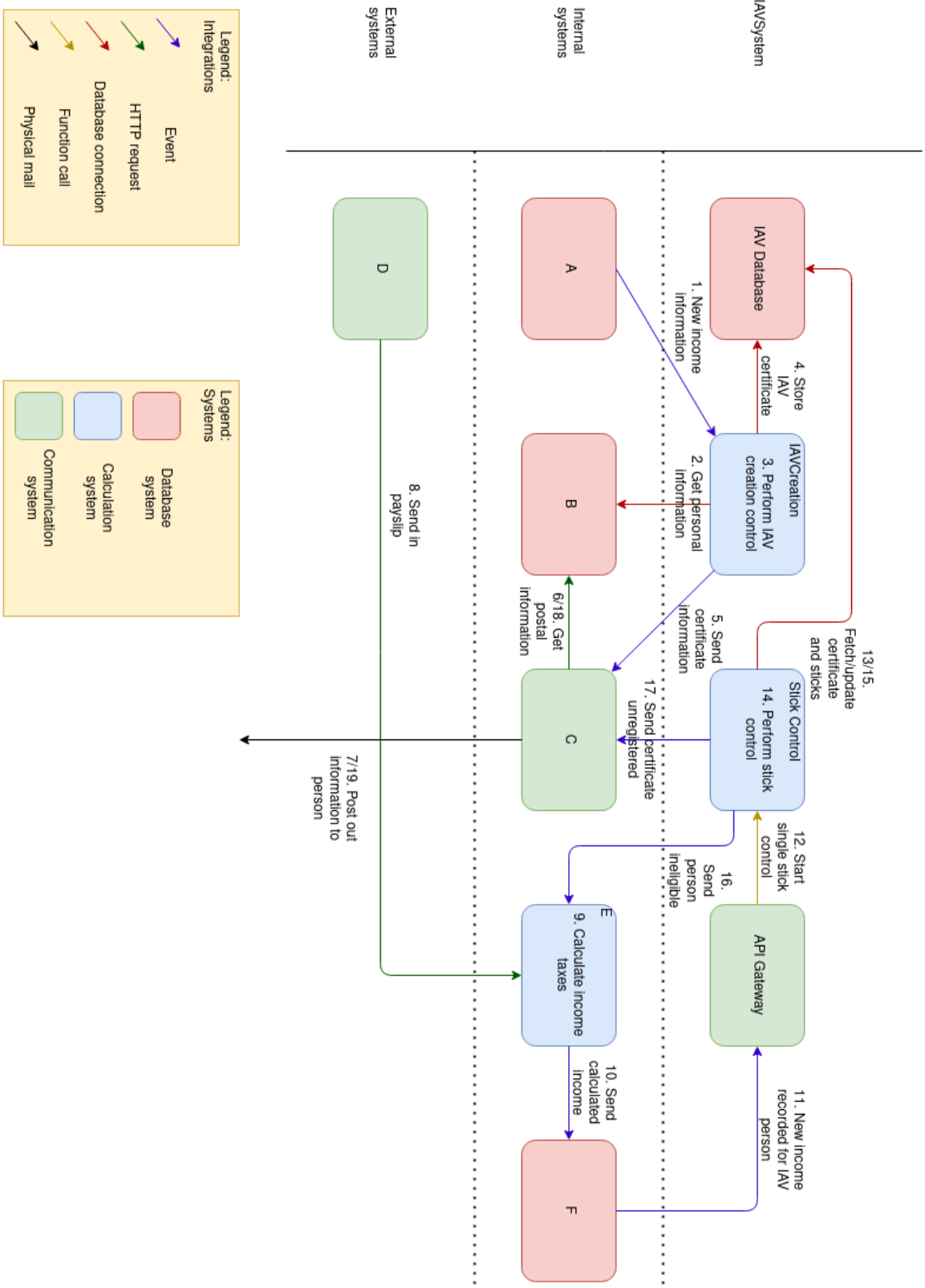

Figure 4.3: Event-driven process flow

IAV Creation Process

Instead of the scheduled batch processes of the monolithic implementation, the IAV creation process begins when system A has recorded new income information for a single person. When this happens, and the person fulfils certain requirements, an event is dispatched saying that a new person has become eligible for a IAVC. The key here is that the event is sent immediately when the information is updated and does not need to wait until a scheduled batch process, which in and of itself takes a long time. Other than that the process is largely the same, the IAVCreation system sends an event to system C instead of a regular HTTP request, but otherwise there is no big difference in functionality.

IAV control process

The IAV control process for the EDA implementation is similar to the monolithic implementation, but also has some substantial differences regarding how it works. Systems D and E remain largely the same, the only distinction being that system E now sends an event when the income has been calculated, instead of a regular HTTP request. System F works quite differently though. Instead of sending a request to the API Gateway that all the payslips for a certain period has been reported, it instead sends a request when a single person that is eligible for an IAVC has been recorded. This means that the stick control process can start before all the payslips have been recorded to the database. The event also sends the income for the person in question, which means that one less request has to be made when performing the process.

Other than that the process remains mainly the same, the only other distinctions being that the Stick Control system sends out two different events based on two scenarios. In the case that a person has accumulated enough sticks and has become ineligible for the certificate, an event is sent which is picked up by system C, which then sends out the information as usual. However, there is also a second event that is sent from the Stick Control system, marked as integration 16 in Figure 4.3. Since the Stick Control starts a process for each IAV-marked person that has their payslip recorded in system F, and are not based on all the stored certificates in the IAV database, multiple payslips could have people marked as IAV eligible, even though they are not, and this will not be known until the Stick Control where the person is checked against the database. If these issues are found, the Stick Control system will send an event that system E will listen after and then correct its information. About 3% of all employees will have this wrong information in both implementations, in

order to compare how they handle and correct faulty information.

4.4 Scenarios

In the two previously presented process flows, there are three different scenarios that are especially interesting to look closer at when comparing these two software architectures. In this section, three scenarios, directly related to the IAV creation process and IAV control process, are defined and measurable metrics within each one are mentioned. An overview of the scenarios can be seen in Table 4.1, where a brief description and metrics used is stated for each scenario.

Number	Name	Description	Metrics
1	First Time Imports	Determining who will be eligible for an IAVC for a large set of people.	Time (ms), Number of database access, Number of HTTP requests, Number of emitted events
2	Report Monthly Income	All employers send in their payslips to the systems for a given payslip period.	Time (ms), Number of database access, Number of HTTP requests, Number of emitted events
3	Stick Control Process	Checking that all IAV people who have a reported income have not earned over the limit.	Time (ms), Number of database access, Number of HTTP requests, Number of emitted events

Table 4.1: An overview of the three scenarios that will be used in this study.

4.4.1 Scenario 1: First Time Imports

The first scenario basically includes the whole IAV creation process, as it contains a scheduled batch run in the monolithic approach while the EDA is, of course, event-driven. In this scenario it is of interest to measure the time it takes to issue the same amount of certificates in both implementations, both in regards for each and every person that goes through the system, as well as the combined time. The amount of database reads and writes as well as HTTP requests and events will be measured too.

Metrics

The metrics that will be collected in this scenario are:

- *Complete scenario execution time (ms)*
- *Execution time for a single IAVC creation (ms)*
- *Number of database accesses*
- *Number of HTTP requests*
- *Number of events emitted*

4.4.2 Scenario 2: Report Monthly Income

The second scenario is found in the first part of the IAV control process, and it concerns the reporting of the payslips. In the simulation 10 000 employers, each with 100 employees, will be sending in their payslips during the same payslip period. It is interesting to measure how long time it will take from the first payslip sent in until the last one is recorded in the system F database. In this scenario the number of HTTP requests, database writes as well as number of events will be measured.

Metrics

The metrics that will be collected in this scenario are:

- *Complete scenario execution time (ms)*
- *Number of database accesses*
- *Number of HTTP requests*
- *Number of events emitted*

4.4.3 Scenario 3: Stick Control Process

The third scenario is the latter part of the IAV control process, and starts when system F has gotten its first payslip until all the current IAVCertificates have been controlled with the new income. This scenario is also interesting since the two implementations work in very different ways. What will be measured here is the amount of time it takes to control each of the IAVCertificates currently in the database. Other metrics such as HTTP requests, database accesses and events sent will be recorded as well.

Metrics

The metrics that will be collected in this scenario are:

- *Execution time for a single Stick Control Process (ms)*
- *Number of database accesses*
- *Number of HTTP requests*
- *Number of events emitted*

4.4.4 Metrics Collection

The metrics will be collected by a set of logging tools written separately from the main implementations. These tools log the relevant metrics and then write them to a set of Comma-Separated Values (CSV) files, to later be analysed.

4.4.5 Motivations of Metrics

All the metrics chosen to be measured in this study have some sort of relation to the performance of a system. Time is the most important factor in all scenarios and thus time is measured in a variety of ways in the scenarios. Other metrics that directly affect the perfomance were chosen as well, these includes the database accesses, HTTP and events emitted. The same metrics were collected from both implementations for each scenario, except for the events emitted which were only collected in the EDA implementation.

4.5 Data

In this section, the different types of data that will be used in the simulation will be shown and explained. The way the data will be generated and how it will be inserted into the system will also be explained.

4.5.1 Data Types

Since this study is looking at a case within Skatteverket, the main data that will be handled is personal information, including personal number, address and various incomes and salaries. There are also a few data types that are directly connected to the IAV case. All data is stored in the format of JSON.

Personal information

The most important type of data is the Personal Information, which represents a single person. Information includes personal number, full name, age, date registered in the system, home address and postal number. An example of the data type can be seen in Listing 4.1. The data was made to be as authentic as possible, but certain points, like personal number and postal number, do not follow the Swedish type. This was due to how the data generation worked, and since it has no functional impact on the simulation, the format was kept the way it was generated.

```
{
  "personalNumber":"6EHTJJ8LJYG50SGF",
  "fullName":"Cleveland Pettway",
  "age":29.7175,
  "dateRegistered":"1982-12-03",
  "homeAddress":"3648 Congresbury Road",
  "postalNumber":"LS83 9WC"
}
```

Listing 4.1: Personal Information

Five Year Income

Due to the requirement of the case, each generated person also needed to have income information for the last five years stored in a system, which needed to be split up into salary and capital income. In order to simplify the data

handling, the Five Year Income data type was created and can be seen in Listing 4.2. Each person thus has their own unique Five Year Income object stored in a database.

```
{
   "personalNumber":"ABLNV8O4CQNEDKU7",
   "salaryIncome":260434,
   "capitalIncome":50936
}
```

Listing 4.2: Five Year Income

Employee

Each employeer will have a unique set of employees, which will be a subset of all the persons generated in the simulation. In order to keep track of which employees belonged to which employees, a new data type, Employee, was created and can be seen in Listing 4.3. The Employee data type only has a personal number as well as optional IAVC information.

```
{
   "personalNumber":"2HUQZFGQS6QQ9LKU",
   "hasCert":true,
   "certId":19
}
```

Listing 4.3: Employee

4.5.2 Data Generation

Since the data used by Skatteverket is highly sensitive information, no data could be provided and everything needed for the simulation had to be generated. In order to ensure as valid results as possible, based of the specific case limitations, 1 000 000 Personal Information data types were generated as an initial data collection, using a Node Package Manager (NPM) Command Line Interface (CLI) command called datamaker[35].

As new data types were needed, like Five Year Income and Employee, it was important that they used personal number from the already generated population in order for the simulation to work. In order to accomplish this, a few utility programs were written that would use the initial data generated and then create the needed data and store to a new JSON file.

4.5.3 Data Insertion

After the data had been successfully generated, it needed to be inserted into the various databases before the simulation could be started. Since the data sets contained 1 000 000 JSON files, a BufferedReader was used in order to efficiently read each JSON object and then store it in the appropriate database.

4.6 Planned Data Analysis

After the data has been collected, it will then be analysed by various software tools in order to be more accessible in the results chapter of this thesis. The following sections details the planned data analysis.

4.6.1 Data Analysis Technique

The technique used for analysing the data will be relatively simple. Since all the data is stored in multiple .csv files, scripts will import this data and perform simple calculations on it. It will be counting occurences of events, summing up execution times for various processes and so forth. The calculated data will then be inserted in various tables in the results chapter of this report. Some data will also be plotted into graphs since it allows for some information to be extracted that cannot as easily be seen from a table summing up the same information.

4.6.2 Software Tools

The software tools used for performing the data analysis will be a few different Python scripts, using various libraries such as Pandas and Pyplot. These scripts have been written by myself and can be found on this public Github-repo[36].

Chapter 5

Results

In this chapter the results of the research will be presented, separated into the three different scenarios defined in chapter 4. The data will be grouped into different types of metric, which mainly are execution time, database accesses, as well as the number of HTTP requests and events emitted. The data is mostly presented in a few different tables, but some data is also plotted out in graphs.

5.1 Scenario 1: First Time Imports

Here the results from the first scenario, "First time imports", will be presented for both implementations. The systems included in this scenario are the IAVSystem, A, B and C. Systems D, E and F are thus not included in these tables and graphs. In Table 5.1 the complete execution time for the scenario (without including the scheduled downtime of the monolithic batch-based process) is displayed. It can be seen that the EDA takes longer time to execute, it performs roughly 9.4% worse than the monolithic implementation.

Implementation	Total time (ms)
Monolithic	17 260 620
EDA	18 884 213

Table 5.1: The complete execution time for scenario 1.

5.1.1 IAV Creation per Person

In Table 5.2 the measurements of the IAV creation process for each person is presented, both for the monolithic and event-driven implementation. The

metric of time is milliseconds (ms) and the column "Creations" indicate in total how many IAVC that were created during the process. The EDA performs about 6.4% slower than the monolithic implementation.

Implementation	Creations	Max time	Min time	Sum time	Mean time	Median time
Monolithic	111 474	2 464	184	24 427 798	219.1	204.0
EDA	111 474	2 734	189	25 986 210	233.1	219.0

Table 5.2: Summary of the IAV creation process.

Figure 5.1 and Figure 5.2 show the complete time spread for the complete IAV creation process for the two implementations. On the x-axis the number of creation processes is displayed and on the y-axis the time it took for each process is displayed. So if an x-value for 20 000 has a corresponding y-value for 500, it means that the 20 000:th control took 500 milliseconds to perform.

5.1.2 Database Accesses

This section details the number of database accesses for the first scenario, which includes the IAVSystem as well as systems A and B. Table 5.3 details the number of database reads and writes for the monolithic implementation, while Table 5.4 shows the number of database reads and writes for the EDA implementation.

System	Reads	Writes
IAVSystem	111 474	111 474
A	1	0
B	222 948	0

Table 5.3: The amount of database accesses for the monolithic implementation for scenario 1.

System	Reads	Writes
IAVSystem	111 474	111 474
A	0	0
B	222 948	0

Table 5.4: The amount of database accesses for the event-driven implementation for scenario 1.

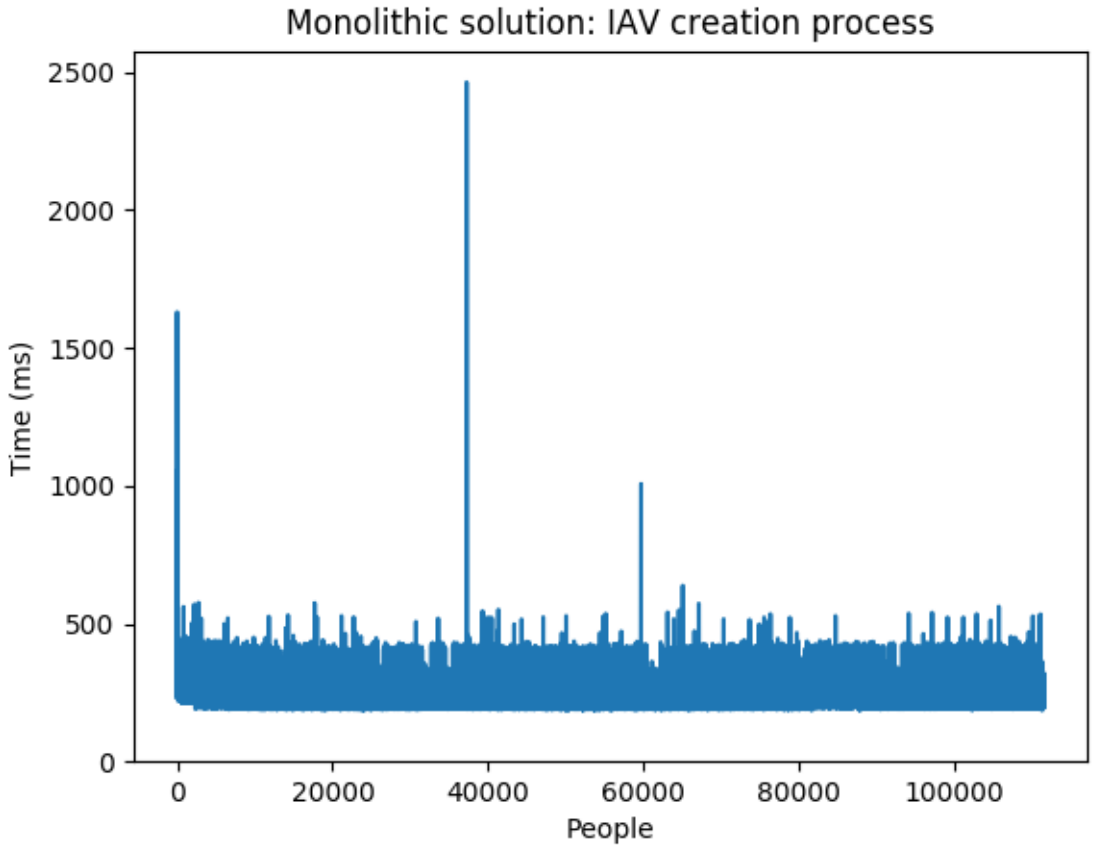

Figure 5.1: The complete spread of the duration for all IAV creations for the monolithic implementation.

5.1.3 HTTP Requests

Both implementations use HTTP requests to communicate between the different systems, although the event-driven approach has replaced quite a few of the requests with events instead, which will have their own results section. Both incoming and outgoing requests were measured for all services, where latency and bandwidth were noted for each request. Table 5.5 shows the incoming HTTP request for the monolithic implementation and Table 5.6 shows the outgoing HTTP requests for the monolithic implementation.

Table 5.7 details the incoming HTTP requests for the EDA implementation and Table 5.8 shows the outgoing HTTP requests for the EDA implementation.

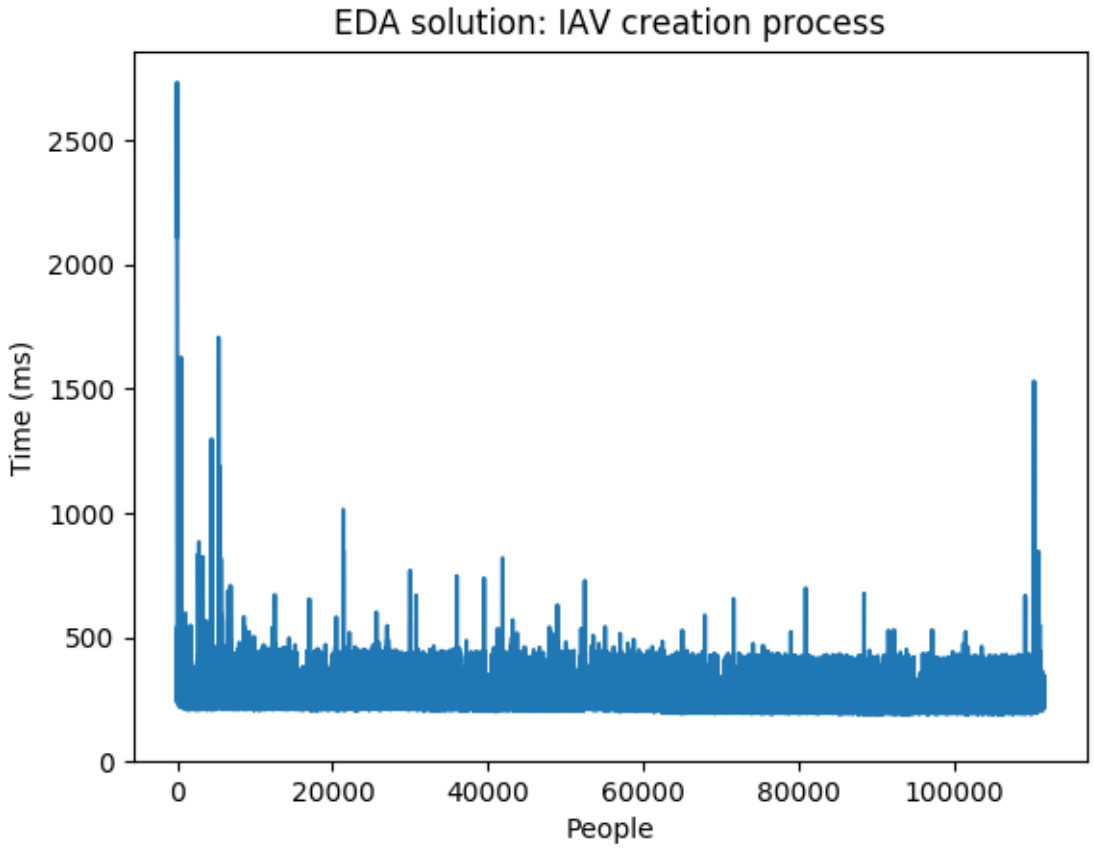

Figure 5.2: The complete spread of the duration for all IAV creations for the EDA implementation.

System	Type	Count	Total content size (bits)
IAVSystem	GET	0	0
IAVSystem	POST	0	0
A	GET	1	0
A	POST	0	0
B	GET	111 474	0
B	POST	0	0
C	GET	0	0
C	POST	111 474	148 040 416

Table 5.5: Incoming HTTP requests for the monolithic implementation for scenario 1.

System	Type	Count	Total latency (ms)
IAVSystem	GET	1	2 126
IAVSystem	POST	111 474	332 294
A	GET	0	0
A	POST	0	0
B	GET	0	0
B	POST	0	0
C	GET	111 474	5 888 863
C	POST	0	0

Table 5.6: Outgoing HTTP requests for the monolithic implementation for scenario 1.

System	Type	Count	Total content size (bits)
IAVSystem	GET	0	0
IAVSystem	POST	0	0
A	GET	0	0
A	POST	0	0
B	GET	111 474	0
B	POST	0	0
C	GET	0	0
C	POST	0	0

Table 5.7: Incoming HTTP requests for the event-driven implementation for scenario 1.

System	Type	Count	Total latency (ms)
IAVSystem	GET	0	0
IAVSystem	POST	0	0
A	GET	0	0
A	POST	0	0
B	GET	0	0
B	POST	0	0
C	GET	111 474	6 820 314
C	POST	0	0

Table 5.8: Outgoing HTTP requests for the event-driven implementation for scenario 1.

5.1.4 Events

The event-driven implementation replaces a few of the HTTP requests with events, where the only metric that was recorded were the number of emitted events. Table 5.9 shows the number of events that were emitted in total in the EDA solution for scenario 1.

System	Number of events omitted
IAVSystem	111 307
A	111 229
B	0
C	0

Table 5.9: The number of emitted events in the event-driven implementation for scenario 1.

5.2 Scenario 2: Report Monthly Income

Here the results of the second scenario, "Report monthly income", will be presented for both implementations. This scenario concerns the reporting of payslips from employers to be stored in an internal database. This scenario includes systems D, E and F and the rest will thus not be included in the tables and graphs. All the results here are from simulations that uses 10 000 different employers, each with 100 employees and thus 1 000 000 different incomes are to be recorded. In order to simulate a more realistic load, each employer sends in their payslip with a time margin of 600-1000 ms, increasing by 100 ms for each request until it reaches 1000 and then resets to 600 ms. This is a realistic load time based on input from the host organisation. This is a deterministic wait since the time comparison should not be affected by this. There is also a 3% chance that system E will be down due to heavy load and the employer thus needs to retry the request. Both implementations use the same random seed to ensure that they fail the equal amount of times in order to not change the validity of the results.

Table 5.10 shows the complete execution time for the monolithic and EDA implementation, where the EDA performs about 0.5% slower than the monolithic solution.

Implementation	Total time (ms)
Monolithic	8 441 633
EDA	8 480 288

Table 5.10: The complete execution time for scenario 2.

5.2.1 Database Accesses

This section details the number of database accesses for the second scenario, which in this case only includes system F, which is the only database system in the scenario. Table 5.11 shows the number of database reads and writes for both implementations.

Implementation	System	Reads	Writes
Monolith	F	0	10 000
EDA	F	0	10 000

Table 5.11: The amount of database accesses for both implementations for scenario 2.

5.2.2 HTTP Requests

In this scenario HTTP requests from systems D and E will be presented (the outgoing requests from system F are only relevant for the third scenario and will thus not be presented here). The event-driven solution has replaced a few of the HTTP requests with emitted events which will have their own results section. Both incoming and outgoing requests will be listed. Table 5.12 shows the incoming HTTP requests for the monolithic implementation, and Table 5.13 shows the outgoing HTTP requests for the monolithic implementation.

System	Type	Count	Total content size (bits)
D	GET	0	0
D	POST	0	0
E	GET	0	0
E	POST	10 000	821 402 368
F	GET	10 000	964 517 736

Table 5.12: Incoming HTTP requests for the monolithic implementation for scenario 2.

System	Type	Count	Total latency (ms)
D	GET	0	0
D	POST	10 000	358 057
E	GET	0	0
E	POST	10 000	24 252

Table 5.13: Outgoing HTTP requests for the monolithic implementation for scenario 2.

Table 5.14 shows the incoming HTTP requests for the EDA implementation and Table 5.15 shows the outgoing HTTP requests for the EDA implementation.

System	Type	Count	Total content size (bits)
D	GET	0	0
D	POST	0	0
E	GET	0	0
E	POST	10 000	964 521 008
F	GET	0	0

Table 5.14: Incoming HTTP requests for the event-driven implementation for scenario 2.

System	Type	Count	Total latency (ms)
D	GET	0	0
D	POST	10 000	359 127
E	GET	0	0
E	POST	0	0

Table 5.15: Outgoing HTTP requests for the event-driven implementation for scenario 2.

5.2.3 Events

One new event has been introduced in the event-driven implementation of the second scenario, the integration between systems E and F, which replaces the monolithic HTTP integration between them. The only metric for measuring events is the number of events systems. Table 5.16 shows the number of emitted events for scenario 2.

System	Number of events emitted
E	9 709

Table 5.16: The number of emitted events in the event-driven implementation for scenario 2.

5.2.4 Availability

In order to simulate a real situation with heavy loads at certain moments, system E will go down 3% of the times when system D tries to send in their income information. In case a request fails, the system will delay for a bit and then retry the request, until it succeeds. In order to make sure that both implementations will fail the same amount of times, the same random seed was used. Table 5.17 shows the number of requests that failed due to systems being down and the total downtime for the systems, for both the monolithic and EDA implementations.

Implementation	Number of failed requests	Total downtime (ms)
Monolithic	308	327 063
EDA	308	326 749

Table 5.17: The availability status between the systems D and E for both implementations for scenario 2.

5.3 Scenario 3: Stick Control Process

Here the results of the third scenario, "Stick control process", will be presented for both implementation. This scenario concerns the control of monthly income and updating the sticks for the active IAVC. This scenario includes systems F, the IAVSystem as well as B and C to a lesser extent. Systems A, D and E will thus not be included in the tables and graphs of this section. This scenario is a direct continuation of the "Report monthly income" scenario and thus the same amount of employers and employees are used.

5.3.1 Stick Control per Person

Here each separate stick control will be shown, both summarised and in a spread graph. In Table 5.18 the metric of time is in milliseconds (ms) and the column "Controls" refers to the number of stick controls that were performed

in total. It can be seen that the EDA is a lot faster than the monolithic implementation. The results show that the EDA performs at 49.0% the time that it takes for the monolith to perform the same task and thus the EDA and takes less than half the time compared to the monolithic execution.

Implementation	Controls	Max time	Min time	Sum time	Mean time	Median time
Monolithic	111 474	3 383	243	41 313 176	370.6	322.0
EDA	125 540	3 326	86	20 238 300	161.2	149.0

Table 5.18: Summary of the stick control process for each person.

Figure 5.3 and Figure 5.4 show the complete time spread for the Stick control process for the monolithic and EDA respectively.

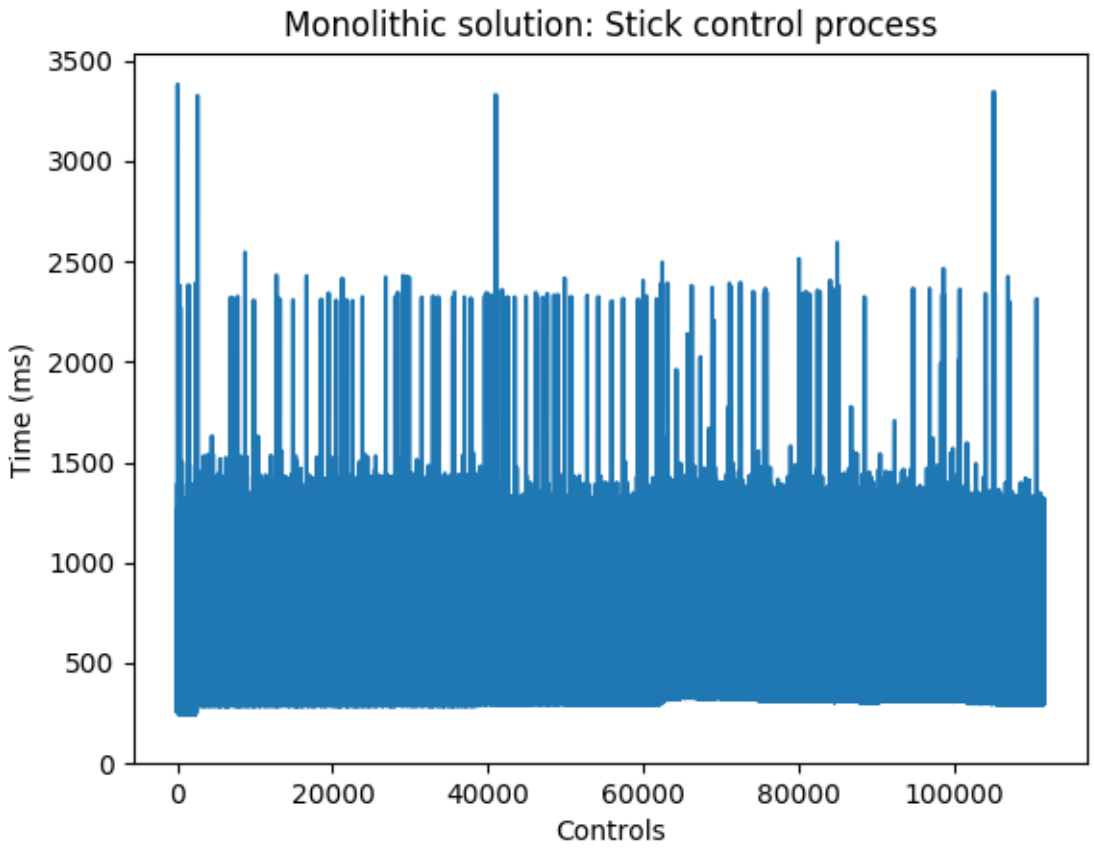

Figure 5.3: The complete spread of the duration for all stick controls for the monolithic implementation for scenario 3.

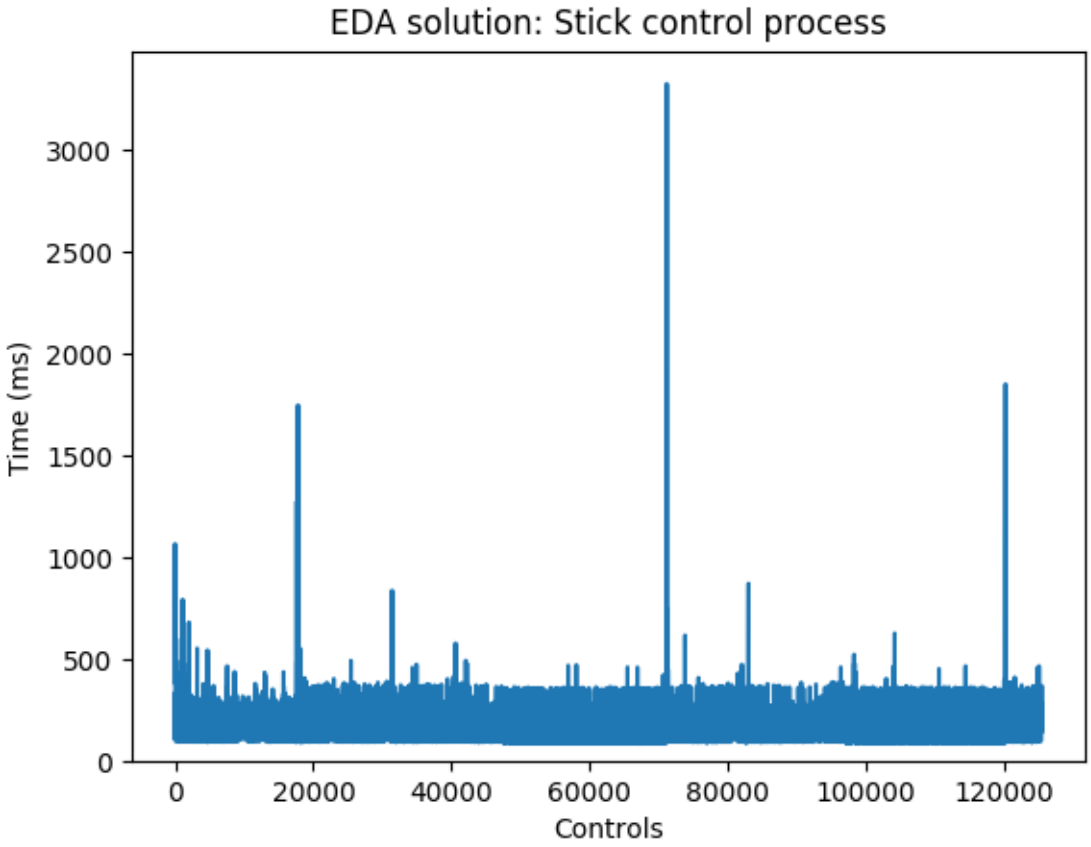

Figure 5.4: The complete spread of the duration for all stick controls for the EDA implementation for scenario 3.

5.3.2 Database Accesses

This section details the number of database accesses for the third scenario, which includes the IAVSystem as well as system F. Table 5.19 and Table 5.20 shows the number of database reads and writes for the monolithic and EDA implementations respectively.

5.3.3 HTTP Requests

Here the HTTP requests of the scenario will be presented, both incoming and outgoing. The event-driven solution have replaced a few of the requests with emitted events instead, which will be displayed in a separate results section. Both incoming and outgoing requests were measured for all services, where

System	Reads	Writes
IAVSystem	222 949	220 518
B	0	0
F	111 474	0

Table 5.19: The amount of database accesses for the monolithic implementation in scenario 3.

System	Reads	Writes
IAVSystem	474 240	128 570
B	3 029	0
F	0	0

Table 5.20: The amount of database accesses for the event-driven implementation in scenario 3.

latency and bandwidth were noted for each request. Table 5.21 and Table 5.22 shows the incoming and outgoing HTTP requests for the monolithic implementation respectively.

System	Type	Count	Total content size (bits)
IAVSystem	GET	1	0
IAVSystem	POST	0	0
B	GET	0	0
B	POST	0	0
C	GET	0	0
C	POST	0	0
F	GET	111 474	0
F	POST	0	0

Table 5.21: Incoming HTTP requests for the monolithic implementation in scenario 3.

Table 5.23 and Table 5.24 show the incoming and outgoing HTTP requests for the EDA implementation respectively.

5.3.4 Events

In this scenario, quite a few new events have been introduced in the event-driven implementation, removing some of the HTTP requests that were needed in the monolithic approach. The only metric used for measuring events is the

System	Type	Count	Total latency (ms)
IAVSystem	GET	111 474	19 545 662
IAVSystem	POST	0	0
B	GET	0	0
B	POST	0	0
C	GET	0	0
C	POST	0	0
F	GET	1	443
F	POST	0	0

Table 5.22: Outgoing HTTP requests for the monolithic implementation in scenario 3.

System	Type	Count	Total content size (bits)
IAVSystem	GET	0	0
IAVSystem	POST	0	0
B	GET	3 029	0
B	POST	0	0
C	GET	0	0
C	POST	0	0
F	GET	0	0
F	POST	0	0

Table 5.23: Incoming HTTP requests for the event-driven implementation in scenario 3.

System	Type	Count	Total latency (ms)
IAVSystem	GET	0	0
IAVSystem	POST	0	0
B	GET	0	0
B	POST	0	0
C	GET	3 029	262 978
C	POST	0	0
F	GET	0	0
F	POST	0	0

Table 5.24: Outgoing HTTP requests for the event-driven implementation in scenario 3.

number of events emitted, per system. Table 5.25 shows the number of events emitted by the EDA implementation in the third scenario.

System	Number of events emitted
IAVSystem	222 937
F	345 545
C	0

Table 5.25: The number of emitted events in the event-driven implementation in scenario 3.

5.3.5 Availability

The monolithic implementation has one more integration that the EDA does not have, an HTTP connection from Stick Control to system F (integration 16 in Figure 4.2). In order to simulate heavy load and availability issues between systems, this integration has a 3% chance to fail and has to be retried. Table 5.26 presents the number of times a request failed and how long time in total the process had to wait because of these issues.

Number of failed requests	Total downtime (ms)
3 486	4 124 071

Table 5.26: The availability status between the systems IAVSystem and F for the monolithic implementation in scenario 3.

Chapter 6

Discussion

In this chapter, the methods and overall implementation and execution will be discussed in detail. The results will then be discussed in depth, divided up into the three scenarios presented in chapter 4. A short discussion about the effects on organisation will also be presented. Lastly, a reliable and validity analysis will be performed of the data collected, as well as a reflections about sustainability and ethics.

6.1 Methods

Here the methods of this research will be discussed. All aspects regarding how the results were acquired will be included, from design to data collection.

6.1.1 Design

The research question of this thesis was entirely based on a real life case that the host organisation had previously worked at, and thus a large part of the work was to adapt this case into a problem that could be solved within a reasonable time. There were some issues with this, such as the scope of the case and that the solution needed to be built in two different system architectures. The design of the monolithic system that was implemented first took about two weeks, which was a reasonable time, but more time should have been allotted for the event-driven solution. The work flow was to first implement the monolithic solution, get the data for it, and then design and implement the event-driven one. Due to the scope of the implementation, the EDA design became very rushed and in the end deviated only slightly from the monolithic one in a few key areas.

6.1.2 Implementation

The implementation of the system took a lot more time than was planned from the start. This was due to the large scope of the system to be implemented as well as changing design ideas. For example, the database systems were initially thought to use a NoSQL implementation, and quite some time was spent developing that, until it was later changed to a SQL database for various reasons. A few of these situations occurred during the implementation, which kept drawing out the process. This is a common part of developing, the iteration of the design, but was not something that was thought of when the planning was made.

The event-driven implementation was made in a significantly shorter time than the monolithic one. After the design of it was approved, the system was directly based on the monolithic one, were a few events were introduced in various places that changed some of the process flows. If there was more time to construct this solution, it could have been built from scratch using an event-driven framework like Apache Kafka or similar, which could have resulted in a more authentic event-driven system than what was developed in this project.

6.1.3 Data Generation

The data that was needed for the different systems and scenarios was generated simulatenously as the systems were being developed. This way of generating data worked well and proved to not be a big issue. Some systems needed to be reworked a bit to make it work with how the data needed to be handled, but nothing that proved to take too much time.

6.1.4 Measurements

A few different measuring frameworks were evaluated and tested to see what would best fit the design of the system. Quite some time was spent on getting this to work and to get the results that were wanted. In the end, it proved to be easier to simply design and implement a set of logging tools that could be directly integrated with the systems. This provided a higher amount of freedom in what data should be recorded and when to do it. All the recorded data was stored in .csv files, and it was then easy to build a few Python scripts that would use this data and properly display it in tables and graphs.

6.2 Results

Here the results of the research will be discussed, divided up by the scenarios that were defined in chapter 4.

6.2.1 Scenario 1

The results of the first scenario are interesting, because at a first glance it might seem that the monolithic implementation is performing better than the event-driven one. As can be seen in Table 5.1, the total execution time of the monolithic implementation is shorter than the EDA (by about 1 624 seconds, or 9.4% slower) and the creation of each IAVC is also generally shorter (but only by a few milliseconds). Comparing the two figures detailing the IAV creation process, Figure 5.1 and Figure 5.2, it can clearly be seen that the monolithic implementation has a more even time distribution compared to the EDA, which has quite a few outliers. That could contribute to why it generally takes longer time for the EDA to perform this process for all data points, as it is 6.4% slower than the monolith. The database accesses are identical for both implementations and seems to be completely unaffected by the differences between them.

The tables for incoming HTTP requests of the two implementations, Table 5.5 and Table 5.7, are similar but the latter is missing the incoming requests to system C. This is however due to that the IavSystem instead sends events to it and therefore are not there any HTTP requests to receive. As for the outgoing HTTP requests, the difference is that in the monolithic implementation, the IavSystem sends one GET request as well as one POST request for each certificate created. The number of POST requests are expected and are simply replaced with events in the EDA solution, ultimately not making a big change in the performance of the systems. The GET request has a latency of over 2 seconds for a single requests, which can be explained by the process starting in the IavSystem in this implementation and has to make a request to system A in order to get the initial data to work with. System A then queries its database and creates a batch file of over 100 000 entires before the system sends a response. This entire process is avoided in the EDA solution, since it is system A that starts the entire process by sending events.

Regarding the events, both the IAVSystem and system A sends over a 100 000 events, which is to be expected, since they should send one event each for each certificate that has been created. One thing here is a bit worrysome though, and that is that the number of events emitted is lower than the number of certificates created or requested. This must mean that the number of events

recorded is somehow incorrect or that some kind of concurrency issues are affecting how the events are recorded when sent.

In general, it does not seem like that event-driven approach is much better than the monolithic approach for this scenario, and is in certain aspects worse. However, there is one big point to take into consideration here, and that is that the monolithic process is a scheduled batch process that starts on a given interval. The event-driven solution on the other hand, does not really start at given times, as the system A just sends an event whenever there is more data to run through the process. This means that there is no wait time for the process, which is a huge improvement when there is a large quantity of data. To put this into context, if some new data is added to system A but the next scheduled process run is in 6 months, this data would simply need to wait this time until it, and all other new data, will be processed at the same time. The EDA on the other hand would instantly send an event over to the IAVSystem and perform the process for only this piece of data. So, a more correct version of the Table 5.1 would be to add the amount of time that the scheduled process has between each run to the current time for the monolithic result, as a worst case scenario.

6.2.2 Scenario 2

The second scenario, "Report monthly income", is an interesting one to look at, as it is the one that has the least amount of changes from the monolithic flow to the event-driven one. The event-driven one has replaced the HTTP integration between systems E and F with an event instead, but in this case, they should work in pretty much the same way. The results support this, as both solutions provided pretty much the same results. The total execution times are very similar, but the monolithic implementation seems to have been a bit quicker, as the EDA performs about 0.5% slower. They have the exact same number of database writes and no reads, which makes sense since the same amount of data is required to be written to system F. The HTTP requests tables also makes sense, 10 000 requests in total is sent through this scenario and while the EDA solution is missing some sent between systems E and F. It is very interesting though that the number of events sent shown in Table 5.16 is 9 709, less than 10 000. Since 10 000 data points are stored to the database in system F for the EDA as well, it must mean that the recorded number is incorrect in this case as well. This case also handles availability of the system, but Table 5.17 shows that they fail the same amount of times with almost the same amount of downtime (the difference is less than 500 ms).

In general, this scenario shows that sometimes an event-driven approach does not really affect the performance of a system much, since it in this case just about performs in the same way as the monolithic implementation. There really is not any difference between how the two systems are implemented, since the events themselves are also implemented with POST request. This case is however pretty small, only concerning a tiny (albeit important) bit of the overall system, which makes the case that EDA might not be the best suited for smaller scale systems and process flows.

6.2.3 Scenario 3

Scenario 3 might be the best argument for the advantages of an event-driven architecture. As can be seen in Table 5.18 the total execution time is about half the amount of time compared to the monolithic solution, even though it performs more than 10 000 additional controls. Even though they have about the same maximum time for a single control, the mean time is about half the amount and the EDA is an 49.0% improvement over the monolith in terms of execution time. This can also very clearly be seen in Figure 5.3 and Figure 5.4, were they have some of the same maximum points, but in general the time of each control is much lower for the event-driven solution.

The huge difference in time in these results can be explained by looking back at the process flows for the two different implementations in, detailed in chapter 4. Comparing Figure 4.2 and Figure 4.3, it becomes quite easy to see that the event-driven solution has one less integration, since the Stick Control process does not need to make a request to system F to get the latest income information, as it has to do in the monolithic implementation, because that information is already sent through an event.

As for the database accesses, it varies quite a bit between the two solutions. The monolithic implementation has less reads and writes in the IavSystem itself, but it does have 111 474 reads in the system F (one for each active certificate), while the EDA has zero reads and writes in the system. The EDA does have more than double the amount of reads in the IAVSystem while it has about half of the writes, which is interesting. The database reads of the system F can be explained by the additional integration of the monolithic implementation. For each active certificate that it checks, it needs to make a request to system F, which results in the extra database read. The system B has 3 029 reads in the EDA solution but 0 in the monolithic one, which can only mean that the integration between systems B and C is only used in the EDA. This is interesting though, because this integration is present in both implementa-

tions and must then come down to the result of the stick control process. For the EDA, 3 029 people failed the control and thus their certificate was made invalid and information sent out. This must be because they had too high of an income when this scenario was executed.

As for the HTTP requests, they vary a lot between the implementations for both the incoming and outgoing ones. The monolithic solution has a single incoming request for the IAVSystem (the initial request from system F to start the entire process) as well as one for each control in system F (in order to get the new income for each control). The outgoing versions of these requests can be seen in Table 5.22, were the requests going from the IAVSystem to system F can be seen listed. It should be noted that the total latency for these requests is 19 545 662 milliseconds, a huge amount of time to wait that the EDA can simply skip, which probably contributes to the vast time improvement that the EDA is in this scenario. As can be seen in table Table 5.23, the only HTTP requests is between the systems B and C which (as explained in the previous paragraph), makes sense for the integration. There are no other HTTP requests, which is logical since it has replaced most of them with emitted events instead.

When looking at the number of events emitted, listed in table Table 5.25, it can be a bit confusing as there are more than 500 000 events in total sent, almost five times the amount of active certificates. This can however quite easily be explained, as there are three different kinds of events that are emitted in this scenario. As can be seen in Figure 4.3, system F sends an event to the IAVSystem when a person with a certificate has recorded a new monthly income, which explains part of the events sent from system F. However, there are quite a few more events sent compared to how many active certificates there are in this scenario. That can explained by one key fact, all employers of system D have a 3% chance of falsely marking a person as an IAVPerson, even if they do not have a certificate. System F has no way of knowing this though, and still sends an event to the IAVSystem for each marked person. The Stick Control process then has to manually check in the IAVDatabase if there is a valid certificate for each person sent from system F. This explains the much higher count of database reads for the EDA, since it simply has more to check. If a person does not have a certificate, and is thus invalid, an event is emitted from the IAVSystem to system D to inform them it of what happened. This all explains the high amount of sent events and database reads, the EDA solution simply has to handle more faulty data because of it's design.

As previously stated, certain system integrations have a chance of becoming unavailable at certain intervals in order to simulate heavy load. This is the

case with the HTTP integration between the IAVSystem and system F. As can be seen in Table 5.26, there are in total 3 486 requests in the monolithic solution that fails and needs to retry, resulting in a total downtime of more than 60 minutes. This simple fact contributes in a large part as to why the monolithic solution takes almost double the amount of time to perform the same scenario. It has an integration that the EDA avoids through it's design, and that integration is so heavily used by the monolith that it adds more than an hour of extra execution time.

There is however one more aspect to take into consideration when looking at the results of the second and third scenario together, that might not be entirely apparent by just going through the data and results individually. In the monolithic solution, the third scenario starts when all the income information has been recorded in system F and it sends a request to start the Stick Control process. This means, if the execution time of these two scenarios were to be measured together, one simply would need to add the complete execution times of both scenarios together, as the third only starts when the second has completely finished. This is where one of the biggest advantages of the EDA can be seen. For the EDA, it starts the third scenario as soon as a single income has been recorded in the database of system F, and then continues to perform the tasks of the two scenarios simulatenously until both are finished. This is a massive improvement over the monolithic solution, since it simply removes the time one of the systems are waiting for information to be sent. The EDA executes its tasks as soon as a single data point is received, and in this specific case that is a massive advantage over the monolithic solution.

Summing up, the third scenario is the strongest argument as to the advantages of using an event-based architectural design compared to a monolithic one. It avoids the need to wait until all the required data is collected before starting a large sequential process, and instead performs the work it can based on the data it has already received. It makes for a much more efficient implementation when handling multiple systems, all dealing with extremely large data sets.

6.3 Effects on Organisation

The effects of the organisation, as stated in one of the sub-research questions in subsection 1.2.1, was also a point of interest for this research. The results themselves do not directly have much to do with the question of organisational effects and cooperation between teams. However, looking at the two different systems that were implemented, the Inverse Conway Maneuver, mentioned in

section 2.11, could be used in order to adapt the structure of the organisations based on the system architectures from Figure 4.2 and Figure 4.3. Taking that into account, the difference of the two organisations would be that the EDA teams would need to interact with each other to a lesser extent, since the architecture has fewer integrations. The simple fact that a system only needs to listen for events, handle them somehow, and then emitt a new event, makes it so that the need for fetching additional information and waiting for HTTP callbacks is eliminated. The less integrations different system teams have between each other, the less they need to wait for other systems to update or risk that systems cannot communicate when another system deploys an update that breaks their contract somehow. The difference between the two designs in this thesis are not that substantial and thus if the organisations were to be evolved around them, the structures and interactions would not be that different from each other. The small differences do affect it somewhat and would these structures and systems be scaled up, it could provide a substantial difference in how the teams of the different architectures work and interact.

6.4 Security of Data

The last sub-research question stated in subsection 1.2.1 was not given a lot of time due to the scope of the implementation and the limited time available to perform this study. It can however be said that since the EDA implementation allowed the system to have fewer data integrations and thus a lesser risk of leaking data or in some other way jeopardizing the confidentiality of sensitive information. This is however not a sufficient answer to the proposed question and should be studied further in future work.

6.5 Reliability Analysis

In order to ensure high reliability, each of the three scenarios were run at least three times for each implementation. The results of each run were evaluated and compared to earlier runs. If there was an issue with any of the results, the implementation was updated and then run again in order to study the results. This process was continued until it was the results were deemed reliable. Effort was also made to always run the scenarios during the same time of day (during the night), without any other programs running at the same time, to ensure the equal amount of processing power and network performance for all runs. Some test runs were also made during the day with several other applications running

at the same time to see if it would cause any difference in the results. After some evaluation, the results were about the same, so the time of day or usage of the computer seemed to have little effect on the outcome of the program runs.

6.6 Validity Analysis

There are quite a few caveats with the validity of the results. Firstly, the fact that this distributed system was simulated on a single Windows 10 Pro computer is a potential validity issue, as it is possible that it would not give accurate results as for how such architectures perform when scaled up to multiple servers. The fact that the events of the event-driven solution were implemented using HTTP POST request and not an event-driven framework means that the full potential of the events might not be reflected in the results, as the same issues with HTTP would be present in the event-driven solution. Also, certain features of an event-driven design, such as an event broken in between systems to handle logic or sending events into the future, could not be part of this solution, which could have improved the event-driven implementation.

However, apart from the hardware and software frameworks used, the architectural idea behind the two implementation has been designed with the help of experienced system architects from the host organisation in an iterative feedback loop process, which should mean that the design idea of the systems reflect how similar systems in real life environments are designed and should thus help ensure the validity of the results acquired by the implementation runs.

6.7 Sustainability and Ethics

The subject of this thesis and the research that was performed does not have much to do with Sustainability. Since it concerns the design and construction of larger systems, issues such as the power needed to keep servers going could be a question of sustainability, and since this thesis tries find out which kind of architecture can produce the most amount of work for the shortest amount of time, it could be said to be related to sustainability. If this thesis would contribute to system architects designing system with an event-driven approach in mind in the future, it could be considered a good thing for sustainability, since it could potentially lower the amount of power needed to perform certain tasks because of a more efficient architecture. This is however a very loose connec-

tion and not a focus of this report. The benefits to sustainability that this report provides will probably only be very minimal, if at all.

Since this research is done at the Swedish tax agency Skatteverket, the question of ethics and privacy is a very big and important one. No real personal data was used during the research and development of this report, all data was generated, but it was made to simulate real personal information, including personal numbers, home addresses and income. This is of course highly sensitive information that needs to be handled with care. There is no way to avoid using this kind of information when designing systems at this organisation, but a case could be made that each system should only know as much as it needs to in order to perform its given task. With that said though, the handling of this sensitive information was not a priority when designing and implementing the solutions, both the monolithic and event-driven approach handle the data in almost the exact same way, even though the process of sending it around varies somewhat. Therefore, more efforts could have been made in order to achieve higher confidentiality of the sensitive data that is handled in these systems.

Chapter 7

Conclusions and Future work

In this chapter, conclusions will be drawn based on the discussions of the previous chapter. The limitations of the project will also be discussed, as well as a future works section describing what has been left undone and what should be done next.

7.1 Conclusions

The main research question of this thesis, presented in subsection 1.2.1, is *What are the benefits of introducing an Event-Driven Architectural style to an ecosystem of large-scale systems?* This question was then divided up into three sub-questions, and this section will try to answer these questions.

To answer the main research question and the first sub-question, *What are the main drawbacks and consequences of doing so and how can they be handled?*, the first conclusion that can be drawn from this research is that an event-driven approach is not always preferable. As can be seen in the results from the first and second scenarios, the EDA implementation performed 9.4% slower than the monolithic one in scenario 1 and 0.5% slower in scenario 2. It can be said that an EDA does not immediately perform better than a monolith, the biggest factor seems to be the systems that should be built around the architecture. Are you building a smaller program where data is mostly just sent from one system to the other, performing a small calculation and then sending it to the next system, there does not really seem to be much of a need for an EDA, as it would basically work and perform in the same way as the monolith solution.

There are however some clear advantages of introducing an event-driven architectural style, and that can plainly be seen from the results of the third

scenario, where the EDA vastly outperformed the monolithic solution, with an improvement percentage of 49.0%. However, this has nothing to do with the way events themselves work (in this simulation they are implemented with regular HTTP POST requests), but the re-design of the systems due to how events can be used. Because events act immediately as data is handled, an entire system integration could be avoided which arguably caused the biggest performance increase when comparing the two systems.

Then there are other important advantages that an EDA introduces that are not as clear from the results of this simulation, and that is that it eliminates the need for data to wait around until a process has been completed before continuing. The simple fact that the first scenario can be started individually for each person as soon as they have new data recorded, is a massive performance improvement compared to the monolithic approach that runs a scheduled batch run at the same time. The EDA is thus a big improvement over the monolithic solution when the scale of the systems and the size of the data increases.

So to answer the main research question, the benefits of introducing an EDA to an ecosystem of large-scale systems is that it can wastly improve execution times by reducing the time systems need to wait for data to be collected or other systems to finish up a task. Other benefits include the ability to immediately handle incoming data and not needing to wait for big batches of data to be sent and calculated at once.

To answer the first sub-question, the main drawbacks of an EDA is that it does not automatically provide a more efficient set of systems. Given the scale of the systems, it might even perform worse (albeit no more than 10% worse in this study). This can be handled by carefully going through the scale of the systems and data to be used and then deciding what kind of architecture would best fit the development goals of the project.

The sub-question *What are the effects on the organisation in terms of work procedure and cooperation between teams?*, was a smaller additional qualitative question in this study. As is briefly discussed in section 6.3, the changes that an EDA style introduces can evolve the organisation and team structure to be more efficient, if the Inverse Conway Maneuver is followed. So to answer the question, if an EDA style is introduced and allows the systems to be more decoupled, the effects on the organisation would be that the different teams would be able to work more independently of each other and could thus be more efficient with their time. This is however only briefly touched upon in this thesis and should be followed up with a separate research paper, only focusing on the organisational aspect of this question.

The last sub-question, *How will this architectural style affect the security*

of the systems, has sadly not been addressed in this thesis as there was no additional time to incorporate a security aspect in the implementation and is thus left for future research.

Summing up, an EDA is not always preferable to a monolithic solution, it depends on the scale of the systems and the size of the data that is handled. For smaller programs a monolithic approach might even be preferable as it is usually easier to design and implement. Where the EDA shines however, is when the number of systems and integrations between them increases and the size of the data sets becomes larger. Because of the ability of an EDA to remove certain integrations and the need for batches of data to be processed at the same time, it can outshine a monolithic solution in many ways.

7.2 Limitations

There were quite a few limitations for this project, both personal and technical. My personal limitations became the biggest hinderances for this project, as my inexperience designing and working with bigger systems like this one contributed to making the implementation taking a lot more time than was allotted for it. The process of understanding the given case and then come up with a design that would be suitable to build in a monolithic and event-driven approach proved to be a bigger task then I would have thought. Since this solution is also quite intertwined with the host organisation's existing systems, I needed to study them and understand how they work. Then I needed to design a suitable simulation for these systems, something that was not part of the original planning at all, and the amount of systems to interact with the IAVSystem caused the solution to scale up in complexity. The measuring part of the research also took a longer time than expected since I did not have experience with how these tools worked, and in the end I still decided to go with my own implementation since I deemed that it would go faster to implement. I also did not have any prior knowledge with any of the primary event-driven frameworks, and since the implementation of the monolithic system took more than double the time it was planned for, the idea for implementing an EDA system from scratch was cancelled and instead it would be built on top of the already existing monolithic solution, only changing a few key parts but keeping most of the system architecture the same.

7.3 Future work

Building on the results and conclusion of this thesis, future work could include designing even more complex sets of systems, with more integrations and data to be sent in between. It would be interesting to see if there are other scenarios in which an EDA does not perform better than a monolith. It would also be interesting to compare an EDA with other kinds of architectures, comparing it further with more pure microservices ecosystems and similar.

Due to time constraints, the event-driven part in this solution was implemented using HTTP requests, the same kind of integration that the monolith mainly uses, which caused the two solutions to become very similar in certain aspects. There are plenty of frameworks for event-driven architectures, like Apache Kafka, Apache Camel etc, that could have made an interesting comparison to a monolithic implementation using HTTP.

Due to the increasing scope of the implementation and time constraints, a few parts of the research question has been left unanswered. The idea from the beginning was to measure both availability and consistency between the different systems during the various scenarios. However, no good way of measure consistency was discovered and was thus excluded from the research. The question of consistency of data is of course still interesting and should be delegated to future work within the area.

Apart from comparing an EDA to a monolith in terms of computational performance and efficiency, the original research question also included a part of the work procedure and cooperation between teams in a larger organisation, as well as how an EDA would affect the security of the systems. Due to the scope and amount of time to implement and research the first part of the question, there was no additional time to answer the other two parts of the research question properly. A brief discussion of the effects on the organisation can be found in section 6.3, but it only briefly touches upon the question and cannot really properly answer it. In order to properly answer the question of effect on organisation, as well as the security of the organisation, a qualitative study should be made, based on the results of this study, where interviews could be held with relevant subject experts at the host organisation.

Based on the results of the research, the next obvious thing would to build an EDA using a framework like Apache Kafka or Apache Camel, in order to evaluate the performance of such a system, compared to building an EDA using HTTP requests. If such a system would be built, it could show a more accurate image of how an event-driven system would work compared to a monolithic one.

7.3.1 Cost Analysis

The simulation that was constructed in order to perform the research works, but from a cost perspective, it could be more efficient. It takes quite a long time to handle not that much data, and if this was to be scaled up even further, the server cost would be very high compared to what would actually be performed by them. If the host organisation would want to design new systems around the EDA style, a cost analysis should first be made, to be sure how high the server costs should run and then try to optimise this solution in order to make it more efficient and lower costs.

7.3.2 Security Analysis

Since this implementation handles extremely sensitive data, the questions of security are very important, but have sadly not been a priority when designing and implementing this solution. Most of these system integrations are not safe, as they use simple HTTP requests to send large amount of data, or use database servers with minimal password protection. If a large scale system should be built with the results of this solution in mind, a lot of time should be spent on analysing the security issues of this implementation and make sure that it becomes as secure as possible.

References

[1] "Software Architecture Guide," Jan 2021, [Online; accessed 11. Mar. 2021]. [Online]. Available: https://martinfowler.com/architecture

[2] S. Ul Haq, "Introduction to Monolithic Architecture and MicroServices Architecture," *Medium*, Jul 2018. [Online]. Available: https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63

[3] "Event-Driven Architecture," Feb 2021, [Online; accessed 16. Mar. 2021]. [Online]. Available: https://aws.amazon.com/event-driven-architecture

[4] "Event-Driven Architecture," Apr 2021, [Online; accessed 1. Jun. 2021]. [Online]. Available: https://www.ibm.com/cloud/learn/event-driven-architecture#toc-benefits-o-iYO1y_8_

[5] "What is event-driven architecture?" Mar 2021, [Online; accessed 16. Mar. 2021]. [Online]. Available: https://www.redhat.com/en/topics/integration/what-is-event-driven-architecture

[6] "What Is Software Architecture & Software Security Design and How Does It Work? | Synopsys," Mar 2021, [Online; accessed 11. Mar. 2021]. [Online]. Available: https://www.synopsys.com/glossary/what-is-software-architecture.html

[7] Archiveddocs, "Three-tier Application Model," Mar 2021, [Online; accessed 11. Mar. 2021]. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/office/developer/server-technologies/aa480455(v=msdn.10)?redirectedfrom=MSDN

[8] "Microservices.io," Mar 2021, [Online; accessed 11. Mar. 2021]. [Online]. Available: https://microservices.io/patterns/monolithic.html

[9] S. Millett and N. Tune, *Patterns, principles, and practices of domain-driven design*. John Wiley & Sons, 2015.

[10] "What is Domain-Driven Design?" Jan 2015, [Online; accessed 15. Mar. 2021]. [Online]. Available: https://www.dddcommunity.org/learning-ddd/what_is_ddd

[11] "Microservices," Jan 2021, [Online; accessed 16. Mar. 2021]. [Online]. Available: https://martinfowler.com/articles/microservices.html

[12] "Microservices.io," Mar 2021, [Online; accessed 16. Mar. 2021]. [Online]. Available: https://microservices.io

[13] K. M. Chandy, "Event-driven applications: Costs, benefits and design approaches," *Gartner Application Integration and Web Services Summit*, vol. 2006, 2006.

[14] "Microservices.io," Feb 2021, [Online; accessed 2. Mar. 2021]. [Online]. Available: https://microservices.io/patterns/data/event-sourcing.html

[15] "What is Event Stream Processing? How & When to Use It | Hazelcast," Oct 2019, [Online; accessed 17. Mar. 2021]. [Online]. Available: https://hazelcast.com/glossary/event-stream-processing

[16] B. M. Michelson, "Event-driven architecture overview, patricia seybold group," 2006.

[17] "Event Processing Approaches in Event-Driven Architecture | Tiempo Dev," Aug 2020, [Online; accessed 4. Jun. 2021]. [Online]. Available: https://www.tiempodev.com/blog/event-processing-approaches-in-event-driven-architecture

[18] "EventStorming," Mar 2020, [Online; accessed 17. Mar. 2021]. [Online]. Available: https://www.eventstorming.com

[19] "Demystifying Conway's Law | ThoughtWorks," May 2021, [Online; accessed 27. May 2021]. [Online]. Available: https://www.thoughtworks.com/insights/articles/demystifying-conways-law

[20] A. MacCormack, C. Baldwin, and J. Rusnak, "Exploring the duality between product and organizational architectures: A test of the "mirroring" hypothesis," *Research Policy*, vol. 41, no. 8, pp. 1309–1324, 2012.

[21] "Inverse Conway Maneuver | Technology Radar | ThoughtWorks," May 2021, [Online; accessed 27. May 2021]. [Online]. Available: https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver

[22] "Why Event Sourcing?" Aug 2020, [Online; accessed 3. Jun. 2021]. [Online]. Available: https://eventuate.io/whyeventsourcing.html

[23] B. Erb and F. Kargl, "Combining discrete event simulations and event sourcing." in *SimuTools*, 2014, pp. 51–55.

[24] "Event-Driven Architecture vs. Event Streaming," Feb 2021, [Online; accessed 3. Jun. 2021]. [Online]. Available: https://www.ibm.com/cloud/blog/event-driven-architecture-vs-event-streaming

[25] O.-A. Schipor, R.-D. Vatavu, and J. Vanderdonckt, "Euphoria: A scalable, event-driven architecture for designing interactions across heterogeneous devices in smart environments," *Information and Software Technology*, vol. 109, pp. 43–59, 2019.

[26] J. Dunkel, A. Fernández, R. Ortiz, and S. Ossowski, "Event-driven architecture for decision support in traffic management systems," *Expert Systems with Applications*, vol. 38, no. 6, pp. 6530–6539, 2011.

[27] L. Filipponi, A. Vitaletti, G. Landi, V. Memeo, G. Laura, and P. Pucci, "Smart city: An event driven architecture for monitoring public spaces with heterogeneous sensors," in *2010 Fourth International Conference on Sensor Technologies and Applications*. IEEE, 2010, pp. 281–286.

[28] A. Theorin, K. Bengtsson, J. Provost, M. Lieder, C. Johnsson, T. Lundholm, and B. Lennartson, "An event-driven manufacturing information system architecture for industry 4.0," *International journal of production research*, vol. 55, no. 5, pp. 1297–1311, 2017.

[29] A. Fertier, A. Montarnal, A.-M. Barthe-Delanoë, S. Truptil, and F. Bénaben, "Real-time data exploitation supported by model-and event-driven architecture to enhance situation awareness, application to crisis management," *Enterprise Information Systems*, vol. 14, no. 6, pp. 769–796, 2020.

[30] J. Veiga, R. R. Expósito, G. L. Taboada, and J. Tourino, "Flame-mr: an event-driven architecture for mapreduce applications," *Future Generation Computer Systems*, vol. 65, pp. 46–56, 2016.

[31] J. Davis, A. Thekumparampil, and D. Lee, "Node. fz: Fuzzing the server-side event-driven architecture," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 145–160.

[32] R. Och, "Ingångsavdrag för att fler ska komma i arbete," *Regeringskansliet*, Sep 2019. [Online]. Available: https://www.regeringen.se/pressmeddelanden/2019/09/ingangsavdrag-for-att-fler-ska-komma-i-arbete

[33] F. Eder, "master_thesis_mono_impl," Jun 2021. [Online]. Available: https://github.com/FelixEder/master_thesis_mono_impl

[34] ——, "master_thesis_eda_impl," Jun 2021. [Online]. Available: https://github.com/FelixEder/master_thesis_eda_impl

[35] "datamaker," Jun 2021, [Online; accessed 4. Jun. 2021]. [Online]. Available: https://www.npmjs.com/package/datamaker

[36] F. Eder, "master_thesis_data_analysis," Jun 2021. [Online]. Available: https://github.com/FelixEder/master_thesis_data_analysis

For DIVA


```
{
"Author1": {
            "Last name": "Eder",
            "First name": "Felix",
            "Local User Id": "u1kxz347",
            "E-mail": "felixed@kth.se",
            "organisation": {"L1": "School of Electrical Engineering and Computer Science ",
                             }
          },
"Degree": {"Educational program": "Degree Programme in Computer Science and Engineering"},
"Title": {
            "Main title": "Comparing Monolithic and Event-Driven Architecture when Designing Large-scale Systems",
            "Language": "eng" },
"Alternative title": {
            "Main title": "Jämföra monolitisk och event-driven arkitektur vid design av storskaliga system",
            "Language": "swe"
},
"Supervisor1": {
                "Last name": "Reza Faragardi",
                "First name": "Hamid",
                "Local User Id": "u1io8pqr",
                "E-mail": "hrfa@kth.se",
                "organisation": {"L1": "School of Electrical Engineering and Computer Science ",
                                 "L2": "DIVISION OF THEORETICAL COMPUTER SCIENCE" }
              },
"Examiner1": {
                "Last name": "Manojlo Kostic",
                "First name": "Dejan",
                "Local User Id": "u12eursm",
                "E-mail": "dmk@kth.se",
                "organisation": {"L1": "School of Electrical Engineering and Computer Science ",
                                 "L2": "DIVISION OF SOFTWARE AND COMPUTER SYSTEMS" }
              },
"Other information": {
"Year": "2021", "Number of pages": "xvii,69"}
}
```

TRITA-EECS-EX-2021:318