Migrating a monolithic system to Event-Driven Microservices - A case study in Philips

Course: 192399979, Submission Date: May 10, 2024

Yanchuan Zhang - s2888629
University of Twente
Netherlands
y.zhang-24@student.utwente.nl

ABSTRACT

The transition from monolithic systems to cloud-based, event-driven microservices architectures represents a significant shift in the software engineering landscape. This project explores the intricacies of such a migration within the context of Philips' Harmonized Business Case (HBC), a critical financial modeling tool. The monolithic nature of HBC, reliant on Excel and VBA, limits its scalability and integration with contemporary technologies. By applying an event-driven microservices architecture, the business processes in which HBC is involved are automated. These microservices can also be reused by other departments at Philips that have similar models. Additionally, the loose coupling feature of this event-driven microservices architecture increases its interoperability.

Employing the Design Science Methodology, this project begins with a thorough investigation into the current state, followed by the design and validation of a novel software migration approach tailored for Agile and DevOps environments. This approach advocates for the emergence of a satisfactory software architecture through continuous small refactoring—a principle aligned with Agile practices.

The proposed event-driven microservices system facilitates decoupled interactions and enhances system modularity, which promotes flexibility and scalability. Validation through software testing and quality analysis further substantiates the effectiveness of the migration approach.

While the project concludes with a functional and validated architecture, limitations due to the incomplete user interface design and lack of deployment in a production environment are acknowledged. Future work suggests a focus on refining the migration process and expanding the research to encapsulate a complete design cycle, including production deployment and user feedback analysis.

This research contributes to the field by providing a framework for successful migration to microservices within the constraints of modern software development practices like Agile and DevOps, ultimately improving business processes through enhanced system design.

1 INTRODUCTION

The use of microservices is rising not only in large companies, such as Amazon[1], Netflix[2], and Spotify[3], but also in small and medium-sized companies like SoundCloud[4][38]. Microservices are a software architecture that has emerged from Service-Oriented Architecture. It comprises several independent microservices, each of which can be developed, tested, and deployed individually by different teams, even using different programming languages.

For companies still using a monolithic system, re-architecting or migrating the pre-existing system to a microservices-based system is becoming popular. This shift facilitates scalability, maintainability, and fault tolerance[9]. Moreover, combining microservices and event-driven architecture also constitutes an emerging trend(See, e.g., [10], [24], [23], [35], [33] , and [37]). In event-driven system, since microservices exchange event information, they do not require detailed information of each other's inner workings. Thus, the system will benefit from further reduced coupling and also near real-time latency[37].

In this project, we explore the challenges and intricacies of transforming Philips' Harmonized Business Case (HBC). HBC is a critical financial modeling tool, currently operating in a monolithic system and integral for consolidating various deliverables across Philips' diverse business sectors. However, its reliance on Excel and VBA presents scalability challenges, integration issues with modern technologies, and inefficiencies in processing large volumes of data input. Additionally, certain manual business processes associated with the HBC are inefficient and could benefit from automation to enhance overall productivity. Moreover, the existence of tools with similar functions in other Philips departments leads to unnecessary resource expenditure. These limitations lead to the need for migrating HBC towards a more dynamic, cloud-based, event-driven microservices architecture. Such a migration aims to enhance performance, foster real-time data processing, and improve user and development experience, aligning with Philips' evolving business needs and technological advancements.

1.1 Methodology

The Design Science Methodology[40] is selected for this project due to its pragmatic and iterative approach to problem-solving

[1]https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-theplatform/
[2]http://nginx.com/blog/Microservices-at-netflix-architectural-best-practices/
[3]www.infoq.com/presentations/linkedin-Microservices-urn
[4]https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith

in information systems. It divides design science into two parts, namely design and investigation. Design corresponds to design problems which "call for a change in the world", while investigation aligns with knowledge questions that "ask the knowledge about the world". For this project, HBC aims to migrate their financial model to a cloud application, aligning with the nature of design problems. Thus, a design cycle that contains *Problem Investigation*, *Treatment Design*, and *Treatment Validation* is most suitable for this project. However, Wieringa[40] also states that the word 'treatment' can be misleading, as the artifact cannot solve all problems. Therefore, in this paper, we use the term 'solution' instead.

By applying this methodology, this project aims to not only propose solutions for the problems of context, but also empirically validate its effectiveness.

1.2 Research Questions

Main Research Question. According to Wieringa[40], a technical research question needs to have four elements, namely an artifact, requirements, stakeholder goals, and problem context. However, the requirements, stakeholder goals, and problem context in this project cannot be summarized by one sentence. Thus, in Chapter 2 we will detail the requirements, stakeholder goals, and problem context separately. We formulate the main research question as:

How to migrate a monolithic system to a cloud-based event-driven microservices architecture system?

Sub-Research Questions.

- Why does the current system need to be transformed into a cloud application?
- What is the software migration approach?
- What challenges will be faced to apply event-driven architecture on microservices?
- What technologies can be used to improve the quality of the migration?
- How to ensure the data integrity and consistency in the new system?
- What benefits would this migration bring?

1.3 Contributions

This project first proposes a software migration approach that is suitable for a migration project which adapts DevOps practices and has an Agile team. Compared with existing software migration approaches, it can shorten the system's development life cycle by kicking off with a working architecture skeleton, which could save the upfront effort to construct a complete architecture. Then a satisfactory software architecture would emerge through continuous small refactoring[13]. Following the migration approach, we propose an event-driven microservices system. In the end, the system's architecture is validated to justify our proposed migration approach.

1.4 Paper structure

The structure of this paper is in accord with design cycle which is mentioned in Chapter 1.1. Chapter 2 will first introduce several concepts and existing works that relates to this project. The problem content, stakeholders and their goals will also be identified. Thus, this chapter is the first phase in design cycle. Chapter 3 represents the *Treatment Design* phase by proposing both migration approach and event-driven microservices system. Chapter 4 will validate the design qualitatively and quantitatively. Chapter 5 will discuss the limitation and future work of this project. Finally in Chapter 6, we will conclude this project and presents the main findings.

2 BACKGROUND

Wieringa[40] define design science as the "design and investigation of artifacts in context". This chapter shows the first phase in the design cycle which is *Problem Investigation.* We first identify and investigate the problem context by recovering the business process and designing the desired architecture. Then, we recognize the stakeholders and their goals. Then we introduce several crucial concepts that relate to the artifacts in this project.

2.1 Problem Investigation

In this chapter, HBC and its functions are first introduced to gain a better understanding of the problem context. After that, the first research question is answered through the analysis of the current business processes in which HBC participates. Then, stakeholders and their goals are identified based on functional limitations. Lastly, the business requirements of this project are formulated.

2.1.1 Harmonized Business Case. The Harmonized Business Case (HBC), at Philips is a versatile tool designed for broad applicability across various business sectors and markets. It systematically consolidates all deliverables and performance obligations from contracts, ensuring pricing aligns with the customer's perceived value. Especially in Long Term Strategic Partnerships (LSPs), which require specialized pricing, quoting, and financial management mechanisms, the HBC plays a crucial role. The current solution for LSP is shown in Figure 1. Currently, the HBC solution processes approximately 10% of the sales of Philips.

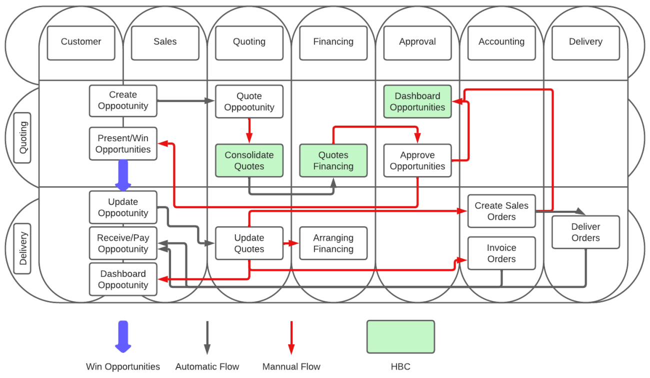


Figure 1: Current Solution for LSP

HBC Functional Overview: HBC enables the generation of various customer-facing scenarios, utilizing master data for accuracy and efficiency. It also facilitates the creation of an integrated Profit & Loss (P&L), Relative Fair Value (RFV), and Cashflow by amalgamating data from various quoting tools, as well as solutions and Long-term Strategic Partners' (LSP) specific inputs, subsequently calculating the Total Contract Value (TCV) which is crucial for the approval of projects. HBC also incorporates indexation and ongoing RFV calculations. These are crucial for financing calculations, such as inflation rates and business assessments over time. It supports the approval and updating of deals throughout their execution phase, and offers benchmarking and dashboard tools for monitoring deal performance during the delivery stage. Lastly, HBC assists in financial optimizations, potentially paving the way for offloading, alongside the recognition and derecognition of revenue and assets, thereby enhancing fiscal management and reporting practice.

HBC Workflow: The current Harmonized Business Case (HBC) is a financial model which is encapsulated in an Excel file. In the Excel file, the user inputs data into the input sheet and executes calculations through VBA code in the calculation sheet. The results are then presented in output sheets. A more detailed description of HBC workflow is shown in Appendix B.

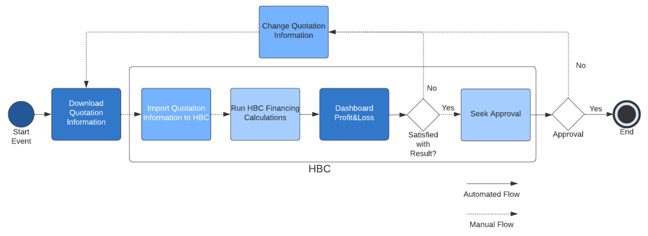

Change Quotation Information
Start Event
Download Quotation Information
Import Quotation Information to HBC
Run HBC Financing Calculations
Dashboard Profit&Loss
No
Yes
Satisfied with Result?
Seek Approval
No
Yes
Approval
End
HBC
Automated Flow
Manual Flow

Figure 2: HBC Process

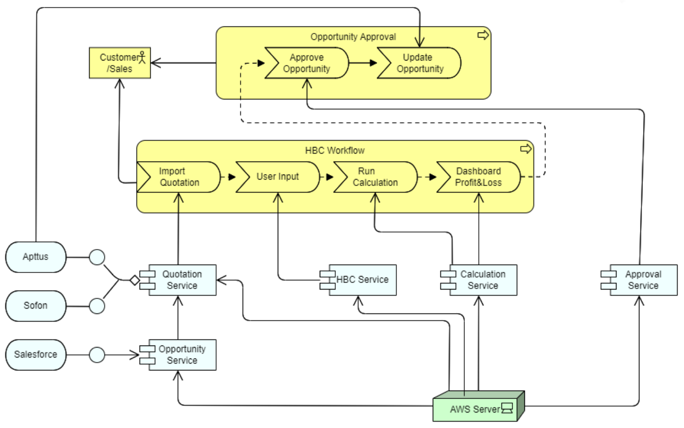


Figure 3: Enterprise Architecture of HBC After Automation

2.1.2 *Business Processes Analysis.*

Business Process Model and Notation (BPMN)[18] is a standardized graphical notation that depicts the steps of a business process from end to end. We utilize BPMN to map out and analyze the processes in which the Harmonized Business Case (HBC) is involved. This visual representation facilitates a clearer understanding and identification of potential improvements.

The Business Process Model, as shown in Figure 2, outlines the sequence of events and activities that constitute the HBC process. It details each step, from the initiation of the process, such as downloading quotation information, to decision points like the approval of the project.

The model reveals several critical manual flows: the download and import of quotation information into HBC for calculation, subsequent approval requirement, and order processing. These manual steps are bottlenecks that reduce the efficiency of the process, presenting security risks and inconsistency in standardization.

The identified manual flows represent opportunities for enhancement through automation. By leveraging cloud services, these steps can be transformed into automated processes. This automation would streamline operations, increase data security, and ensure standardization and interoperability across the business process.

2.1.3 *Enterprise Architecture Model.*

To show the automated flow and the overall architecture, ArchiMate[21] is selected as the modeling language due to its ability to clearly communicate complex architectures across business, application, and technology layers. It provides a standardized framework that enhances understanding among stakeholders from various backgrounds.

The model shown in Figure 3 automates every manual flow depicted in Figure 2 into a cloud service. Note that from Figure 1 it become clear that the business events associated with the services shown in Figure 3 are owned by different departments of Philips. Thus, each service should be developed by different development teams and deployed individually, making a microservices architecture[1] the ideal solution due to its characteristics. However, the microservices need to interact with each other, either through REST API calls or event messages. In this case, an event-driven architecture is considered superior in terms of both performance and interoperability[20]. Additionally, external services like Salesforce also employ an event-driven architecture, which could simplify integration.

Even though the scope of this project only focuses on Calculation Service, it is beneficial to gain a comprehensive overview of the architecture so that, according to the components within the architecture, different design patterns can be used to maximize efficiency and avoid cumulative rework in the future[30].

2.1.4 Stakeholders. According to Wieringa[40], stakeholders of a problem are people who "will be affected by treating the problem". The most important stakeholder of this project is the HBC team. On the other hand, stakeholders interacting with the artifact are identified previously in Chapter 2.1.1, including the groups shown in Figure 1, namely sales, quoting, financing, approval, and accounting team. For convenience, we refer to these stakeholder groups as "users" of HBC. Besides, with Philips there are other departments such as Enterprise Monitor As A Service (EMAAS) and Enterprise Informatics (EI) need to perform similar financing calculations like HBC. Currently, they operate their tools independently. The artifact in this project may have a potential effect on them by consolidating the existing calculation tools.

2.1.5 Functional Limitations. These are the functional limitations within the current system that prevent stakeholder goals from being met. FLs are identified according to the HBC functional overview and workflow.

No.	Description
FL1	When salesperson trying to have approval for a project, they need to manually present the HBC P&L result in teams meetings to grant permission. Similarly in Figure 1, all the arrows represent a manual flow which is inefficient and would possibly hinder the profitability.
FL2	When users trying to use Current HBC, the user interface consists of not only necessary data input cells but also optional data input cells, which would confuse and overwhelm users.
FL3	When project input contains hundreds of items, the excel needs more than 10 seconds to calculate the output and response.
FL4	When the current HBC tool fails, it is hard to debug.
FL5	When HBC need to show different content to different user groups, it is difficult to restrict user permissions in current excel file.
FL6	When other departments within Philips want to reuse existing calculation function in HBC, they cannot directly reuse it and have to make an individual tool.
FL7	When making a change to excel file, the existing rigid monolithic style of the system makes changes and updates cumbersome.
FL8	When HBC team doing collaborative work, it can achieve is hard to do version control in Excel side.
FL9	When there is the need to scale up, the current set up doesn't support it.
FL10	When adapting modern technologies, it could be problematic because of compatibility.
FL11	When testing a new version of the Excel file, it is not feasible to do test automation.

Table 1: Functional Limitations

2.1.6 Stakeholder Goals. Stakeholder goals are the high-level objectives or outcomes that stakeholders wish to achieve through the project, aligning with the overall mission and vision of the HBC initiative. We identify stakeholder goals based on stakeholder groups, previous description of HBC functional overview, workflow and functional limitations. In Table 2, each stakeholder goal is linked to a functional limitation.

No.	Description	Functional Limitation	Stakeholder
G1.1	Automation Efficiency	FL1	HBC Users
G2.1	Enhance access control	FL5	HBC Development team
G2.2	Streamlined development	FL7	HBC Development team
G2.3	Scalability	FL9	HBC Development team
G2.4	Version control improvement	FL8	HBC Development team
G2.5	Economical technological adaptation	FL10	HBC Development team
G2.6	Maintainability	FL4 FL11	HBC Development team
G2.7	User experience excellence	FL2	HBC Development team
G2.8	Performance Optimization	FL10	HBC Development team
G2.9	Robust	FL4	HBC Development team
G2.10	Real-time processing	FL3	HBC Development team
G3.1	Organizational standardization and reusability	FL6	Other Philips departments

Table 2: Stakeholder Goals

2.1.7 Business Requirements. According to the stakeholder goals in Chapter 2.1.6, twelve business requirements are initiated, discussed, and refined with the project owner. The BRs are shown in Table 3. BR stands for Business Requirement, numbered from 1-15. The BRs use Easy Approach to Requirements Syntax (EARS)[27]. The Goals correspond to the stakeholder goals of each business requirement. The priority is added to each business requirements using MoSCoW[14].

No.	Goal	Description	MoSCoW
BR1	G1.1 G2.2 G2.3 G3.1	The new system shall be a cloud application.	Must
BR2	G2.10	The new system shall process real-time data and provide a response in less than 500 milliseconds.	Must
BR3	G2.9	The new system shall handle faults effectively without system failure.	Should
BR4	G2.8 G2.5	The new system shall be able to interoperate with other Philips applications.	Should
BR5	G2.8	The new system shall ensure data integrity and consistency.	Must
BR6	G3.1	The new system shall be reused and be able to integrated by other Philip departments.	Could
BR7	G2.7	The system shall take input from and return output to user interfaces.	Must
BR8	G2.10	The new system shall save the calculation result into a database.	Must
BR9	G2.10	The new system shall read from data tables.	Must
BR10	G2.6	The new system shall log user operation record.	Must
BR11	G2.5	The new system shall be budget friendly to maintain and adapt new technologies.	Should
BR12	G2.1	The new system shall show different content to different user groups.	Could
BR13	G2.8	The new system shall produce the same output as the current Excel calculation.	Must
BR14	G2.2	The design cycle should be short.	Must
BR15	G2.7 G2.8	The system should be able to handle requests from 1,000 users within a 16-hour period each day.	Must

Table 3: Business Requirements

Additionally, the HBC group has transformed the existing system into a detailed design, outlined in a PowerPoint presentation. An illustrative slide from this presentation is shown in Figure 12. This presentation will henceforth be referred to as the 'design PowerPoint'. The project objective is to adhere to the guidelines set forth in the design PowerPoint, and to redevelop the existing Excel-based functionalities into a Java-based cloud application.

As a financial model for calculation, the functions can be recovered and categorized by their calculation outputs. However, only basic calculations have been chosen for this migration. As shown in Table 14 and Table 15, all calculations are divided into three categories: *equipment*, *customer Service* and *management service*.

2.1.8 Constraints. While requirements would capture feature and functions of the system, constraints defines non-functional aspects of the system. Before this design cycle has started, the HBC group has already first initiated the microservices framework. Towards an IT project, Philips has the following constraints:

- ITAAP(IT as a Platform) department will provide support and supervision.
- The project utilizes Git for version control, hosted in a GitHub repository within Philips' internal resources. The code cannot be directly merged into main branch. Instead, any change must be first made in a branch.
- Utilizes Java as the programming language.
- Employs Spring Boot as the framework.
- Secure application access with OAuth 2.0.
- It uses Azure CI/CD pipeline.
- It uses Docker container managed by Kubernetes.
- It uses test automation and Infrastructure as Code(IaC).

2.2 Monolithic

The word "monolith" is used to describe a single-process application that gets over-complicated[3]. Single-process means the application where its core functionalities like presentation, business logic, and storage, are intertwined and packed in a singular deployment unit. Characteristically, in a monolithic design, which usually contains a client-side user interfaces (HTML pages with JavaScript running on user machine), databases (normally a relational database containing many tables), and server-side applications (where all the business logic is at) are intricately linked. The server-side application manages HTTP interactions, carries out the core operational logic, interfaces with the database for data retrieval and modification, and chooses HTML pages for transmission to the web browser. This server-side application can be called a monolith, functioning as a single logical executable.[1].

Due to its cohesive nature, such a consolidation of all elements into one executable structure ensures ease of development, deployment, and operations. However, it presents inherent challenges. Even minor changes mean that you have to update and deploy the whole system. A single-process approach also implies that it may not meet all the feature requirements for the newest technology. Eventually, it will become over-complicated due to difficulty in scaling. On a human resource management perspective, multi-teams working on a unitary code base require extra synchronization management to avoid conflicts[31].

Yet, despite its limitations, monolithic applications have been historically successful and represent a natural initial approach to system development. It is only with the advent and proliferation of cloud-based deployments, coupled with evolving demands, that the constraints of monolithic systems have become more pronounced.

2.3 Microservice

Microservice is a software architectural style by developing a suite of independent services that can be developed, tested, and deployed individually. Each microservice is built on a single business capability and runs its own process and communicates with other microservices using light weight mechanisms like HTTP resource API[1]. In other words, changes happens in one microservice on the server side will not affect another microservice.

Based on these features, the advantages of microservices could be summarized as:

- Loose-coupling: Individual service are decoupled and can be developed and maintained by different teams and even using different programming languages[1].
- Scalability: Independence of each microservice makes it possible to scale only the necessary services. Moreover, the independence of each microservice allows them to be deployed across multiple servers or even in different data centers, enhancing scalability by more effectively distributing loads. [1].
- Robustness: The system won't fail even if some of the microservices fail[41].
- Continuous Delivery: It is easier to achieve continuous delivery because of its light weight compared with monolith. The short deployment cycle also makes it easier to adapt new technologies[41].

2.4 Event-driven

Event-driven microservices architecture refers to a system, which consists of loose-coupled microservices that communicate by exchanging event messages. In an event-driven architecture, three main components are involved: the producer, the consumer, and the event bus. The microservice which produces the message is called a "producer" and the microservice that consumes this message is a "consumer". The producer sends event messages to the event bus, and the consumer listens to the event bus for new event messages. The event messages are also replayable[20].

2.4.1 Event-Driven Topology. There are two main topologies for event bus: mediator and broker.

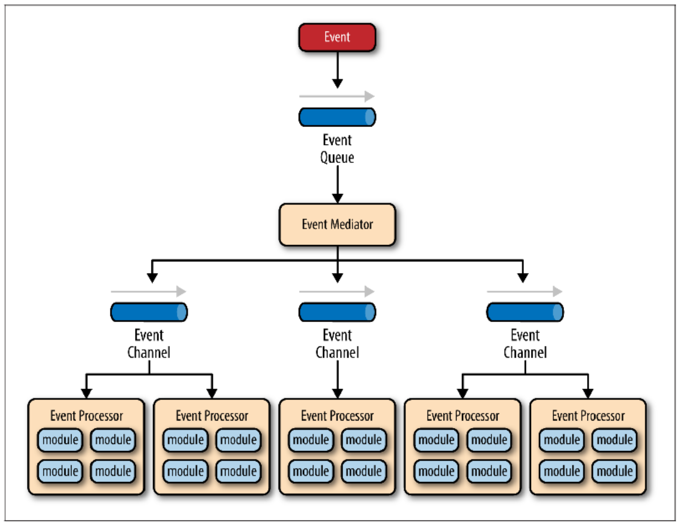


Figure 4: Event-driven architecture mediator topology [34]

Mediator. Mediator consists of four main components: event queue, event mediator, event channel and event processor. The event queue transports the event sent from the client to the mediator, whereas additional asynchronous events are sent to event channels. Event processors listen to event channels, and process the event by executing business logic[34][41]. The architecture of a mediator is shown in Figure 4.

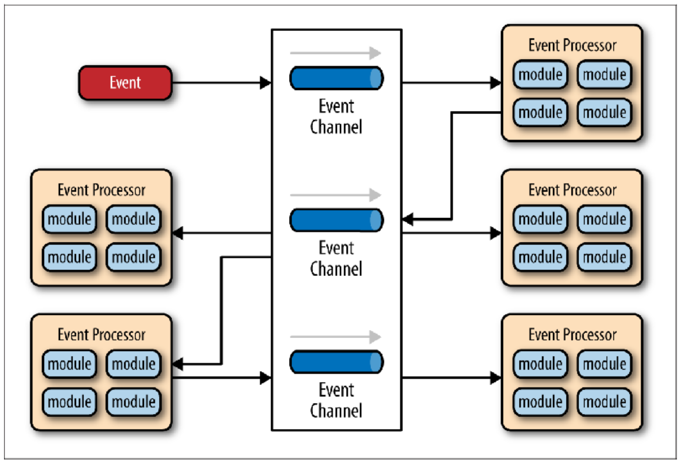


Figure 5: Event-driven architecture broker topology [34]

Broker. As shown in Figure 5, there are two components in the broker topology: the event processor and theevent broker. Inside the broker, it provides event channels that are either message queues, message topics, or a combination of both. The processors are responsible for processing and publishing events[34][41].

2.5 Software Migration

Software migration, also known as software modernization, refers to a re-engineering process of migration or adaption that applies a specific method[12]. Strangler and Horseshoe approaches, which are two well-documented migration solutions are outlined below.

2.5.1 Strangler. The Strangler approach[4] suggests a gradual transition from an old system to a new one. Instead of attempting a direct and risky replacement of the entire system, the new system is developed incrementally around the edges of the existing system. As new components are built and integrated, they begin to replace the corresponding parts of the old system. This process continues until the new system has entirely strangled the old one.

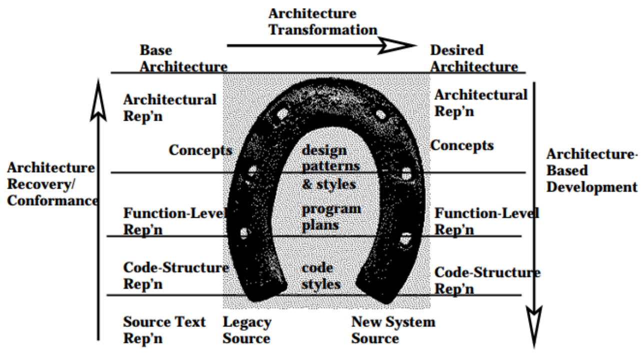


Figure 6: The horseshoe approach[22]

2.5.2 Horseshoe. The CORUM II model[22], commonly known as the horseshoe model, provides a metaphorical framework for integrating code-level and architectural reengineering perspectives in software systems. The horseshoe is divided into three related processes, operating across four levels of software representation. The three processes are described below:

- I: Code and Architecture Recovery and Conformance Evaluation: This involves recovering the architecture of an existing system from its source code artifacts.
- II: Architectural Transformation: Given a desired new architecture based on specific system requirements, the existing architecture is reengineered to match the desired new architecture.
- III: Architecture-Based Development: The high-level architectural design detailed in the second process is initiated and entails various low level transformations.

As for the four levels of software representation, they are:

- Architectural
- Functional
- Code-Structure
- Source-Text

The horseshoe represents transitions from code-level facts to an understanding of software architecture (Code and Architecture Recovery and Conformance Evaluation), manipulation of architectural concepts (Architectural Transformation), and architecture-based development (Architecture-Based Development).

Horseshoe approach is particularly relevant for tasks like updating core technologies, incorporating new technologies, moving to object-oriented architectures, distributing systems using modern architectures, and more. The horseshoe model provides a structured way to approach these complex reengineering tasks.

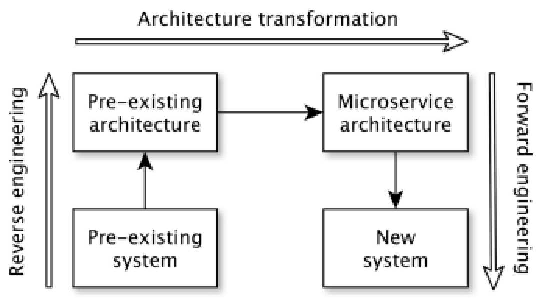

Architecture transformation
Reverse engineering
Pre-existing architecture
Microservice architecture
Forward engineering
Pre-existing system
New system

Figure 7: Migration to microservices[17]

Adapted horseshoe: To have a distinctive view on how to migrate towards microservices, Francesco et al.(2018)[17] present a model shown in Figure 7 which is an adaption of the horseshoe model specifically for the purpose of migrating to microservices. They also conducted a survey with 18 practitioners that provide information about migrations towards microservices architecture in industry. For each process, the activities carried out by the practitioners can be concluded as:

Reverse engineering: The practitioners first obtain knowledge from both low-level and high-level sources:
- low-level: For details of the pre-existing system, they would directly go into source code and tests.
- high-level: At a higher level of abstraction, sources such as textual documents, slides, architectural documents, data models, and insights from developers within the company are utilized.

Architecture transformation: The activities in designing the new microservices architecture involves:
- Divide the domains into sub-domains.
- Identify services for the new system.
- Divide the pre-existing system into several modules.

Following these activities, a step to demonstrate the migration's feasibility is necessary. This could be realized through a service that validates the migration's feasibility or through developing Minimum Viable Products (MVPs), which are artifacts that, despite being incomplete in functionality or quality, exhibit characteristics that help to determine their business value[29].

Forward engineering: Depending on the features of the pre-existing system, some practitioners start the migration by directly adding new functionalities as microservices, while others reengineer the functionalities of the pre-existing system into microservices. It is also noticeable that in some cases the practitioners would start the migration by implementing the existing functionalities into microservices system.

2.6 Challenges

Based on the stakeholder goals and BR, we can conclude that the HBC team is searching for a cloud application that is easy to scale to handle potential user growth, can be incrementally developed, leverages the power of DevOps practices like the CI/CD pipeline to accelerate the design cycle, is reusable across other departments within Philips, and is easy to maintain. This combination of requirements makes microservices architecture an ideal fit, as it inherently supports scalability, incremental development, and seamless integration with DevOps practices, effectively meeting the diverse needs of the HBC team.

Additionally, by implementing an event-driven architecture, the microservices are able to communicate with each other and integrate into a larger system more efficiently than with REST API calls.

As for the software migration approaches, the Strangler approach emphasizes gradually 'strangling' the pre-existing system by incrementally building around it. However, this approach is not suitable for this project, where microservices are built individually as a new system. As for horseshoe and the adapted horseshoe mentioned in Francesco et al (2018)[17], they are suitable to be used for migrating a monolithic system to microservices.

However, the HBC team employs the Scrum methodology[36], which means the work is done incrementally through sprints, each lasting ten working days. Moreover, the constraints listed in Chapter 2.1.8 indicate that this project adopts DevOps practices. In contrast, the horseshoe and adapted horseshoe methods do not align with Agile and DevOps practices, particularly during the Architectural Transformation phase. This phase contradicts the Agile principle of avoiding extensive up-front design and a waterfall process, which could potentially slow down the design and deployment cycles of microservices. Furthermore, according to a systematic literature review on migrating to microservices[7], no existing software migration approach fully integrates DevOps practices. This review also notes that there is no documented application of event-driven or message broker patterns in microservices under DevOps practices.

Glen[2] identifies "finding where to break up monolithic components" as the most challenging aspect faced during the migration of legacy systems. Francesco et al[17] also highlight 'High coupling among parts of the pre-existing system' and 'Setting up the initial infrastructure for microservices to work' as common challenges during the architecture transformation and forward engineering phases, respectively. As mentioned previously in Chapter 3, HBC team has broken down HBC Excel model into a design PowerPoint. Nevertheless, the individuals responsible for breaking down the HBC model are not software engineers. Thus, during the "Architecture Transformation" phase of the horseshoe and adapted horseshoe model, setting up the microservices architecture could pose a significant challenge.

Meanwhile, business requirement BR14 mandates shortening the design cycle due to business considerations. In Agile methodology, it is generally considered that extensive upfront effort in architecture planning delivers little value to stakeholders[8]. However, neglecting software architecture can not only affect the performance of the system but also increase the amount of rework required[30]. Thus, it is challenging to satisfy stakeholders' needs while maintaining a good software architecture with existing software migration approaches.

In short, the first artifact of this project is the software migration approach. It could be improved in the context of DevOps and Agile practices. The second artifact, the event-driven microservices system, addresses two primary problem contexts: the existing business processes and the Excel file. By creating a cloud application, the existing business processes can be automated, and the performance of the Excel file can be enhanced through migration to an event-driven microservices architecture.

3 DESIGN

This chapter presents the design phase of the design cycle, where we propose two solutions: a software migration approach and an event-driven microservices system. These are aimed at addressing parts of the problem context identified previously in Chapter 2.1.

3.1 Migration Approach

As discussed in Chapter 2.6, existing migration approaches are not fully compatible with DevOps practices and may hinder the Agile properties inherent in microservices architecture. Additionally, the complexity and high coupling of the pre-existing monolithic system present significant challenges in setting up a microservices architecture.

3.1.1 Emergence of Architecture through Continuous Refactoring. The adoption of Agile methodologies has significantly influenced the practices of software development in the industry. Proponents of Agile methods often perceive minimal benefit to customers from the initial design and assessment of a system's architecture[8]. Meanwhile, advocates of architecture-centric approaches argue that neglecting architecture at any stage of a system's lifecycle can lead to severe repercussions.

As a result, one of the most fundamental debates, which is "whether or not a satisfactory architecture emerges from continuous small refactoring in agile software development"[19] is answered in Chen and Barba[13], by identifying 20 key factors, which is shown in Table 16. The result shows that for the projects which satisfies the contexts characteristics, it is likely that a satisfactory architecture would emerge from continuous small refactoring.

Thus, we use 20 key factors to evaluate whether this project is suitable for a satisfactory architecture to emerge by continuous small refactoring. However, some of the 20 key factors involve aspects like knowledge, culture, and mindset, which cannot be measured directly. Thus, a survey with seven questions is designed and conducted within the development team for the purpose of measuring these aspects. The questions are defined according to the answers in Chen and Barbar[13]'s survey. Questions of the survey is shown in Table 4. The participants of this survey are HBC development team members, in total five person. They are asked whether they would agree with the questions in the survey by choosing from 1-5:

- 1: Strongly Disagree
- 2: Disagree
- 3: Neutral

- 4: Agree
- 5: Strongly Agree

This scale, developed by Rensis Likert[25], allows for a nuanced understanding of participants' levels of agreement or disagreement with the survey questions. The result of the survey is the average of participants' responses to each question.

Questions	**Average Points**
Do you think this team is mature in knowledge of reference architectures, design patterns, and tactics?	4
Do you think change should be acknowledged as a part of the development process?	5
Are you willing to learn technology and try to adapt to it?	5
Do you think within this team there are good communication channels?	5
Do you think the culture within this team encourages people to take ownership and commitment, be open, and blame-free?	5
Do you think the organization structure is embracing the openness of Agile approach?	4.2
Do you think the team would provide effective governance on software architecture?	4

Table 4: Survey Questions

- *Rate of Change*: While significant implementation has not occurred before and the business requirements are defined (as mentioned in Chapter 3), changes in software architecture and technical details may occur during the project.
- *Size of a project*: This project does not involve external groups. Given that the team consists of fewer than six people, all located in the Netherlands, it qualifies as a relatively small project.
- *Type of Project*: As described in Chapter 2, this project involves a series of functions that can be incrementally built and released.
- *System Age*: The system for this project is newly designed using contemporary principles and technologies, unlike older systems that may be overly complex and coupled. This makes it easier to achieve a unified and integrated architecture.
- *Type of ASRs (Architecturally Significant Requirements) and Their Criticality*: The Architecturally Significant Requirements (ASRs) for this project, outlined in Chapter 3, are in compliance with Philips IT standards
- *Safe Net*: A safety net is provided through the integration of test automation within the Azure Continuous Integration Pipeline.
- *Continuous Integration*: Continuous Integration is facilitated by an Azure pipeline, automatically triggered by any Git pull request.
- *Good Design Principles*: Contemporary design principles, which are employed in this project, will be discussed in more detail in Chapter 3.2.
- *Management*: The team employs a SCRUM methodology, which is well-suited to accommodating changes.

Consequently, considering the survey results and the detailed analysis provided above, it is determined that this project is well-suited for developing a satisfactory architecture through the process of continuous small refactoring.

3.1.2 Proposed Migration Approach. As a result, Figure 8 presents the proposed migration approach.

There are in total four phases in this approach:

Reverse Engineering. Similar to the horseshoe and adapted horseshoe approaches discussed in Chapter 2.5, this phase involves recovering and documenting the pre-existing system, ranging from high-level documents to low-level code.

Architectural Transformation. This phase is inspired by the 'Walking Skeleton' strategy, which refers to a small implementation that links together the main architectural components. While the final architecture may not be used initially, the goal is to create a working architecture skeleton. This minimizes initial effort on architectural issues and accelerates project kickoff, leading to early success and increased team confidence[15].

Forward Engineering. Unlike existing migration approaches, the "Forward Engineering" phase follows "Reverse Engineering". It encompasses code implementation, continuous integration and delivery, and continuous small refactoring, allowing the architecture to evolve incrementally alongside system functionality[15]. This phase will cycle until the satisfactory architecture of microservices emerges during the continuous small refactoring process.

Satisfactory Architecture Development. In the final phase, the satisfactory architecture emerges from the continuous small refactoring during forward engineering. This involves recovering and documenting the architecture of the new system.

As the *Reverse Engineering* phase was completed before this design cycle began, this project follows the proposed migration approach starting from the *Architecture Transformation* phase.

In this project, the walking skeleton is presented in Chapter 2.1.3. Additionally, the microservices architecture is employed within the calculation service, organizing the functions outlined in the design PowerPoint presentation into distinct microservices. Continuous small refactoring occurs throughout the *Forward Engineering* phase, with the emergent architecture detailed in Chapter 3.2.

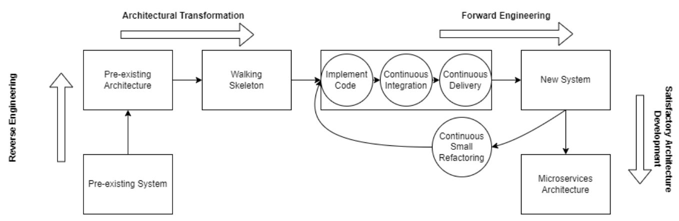

Architectural Transformation
Forward Engineering
Reverse Engineering
Pre-existing Architecture
Walking Skeleton
Implement Code
Continuous Integration
Continuous Delivery
New System
Continuous Small Refactoring
Pre-existing System
Microservices Architecture
Satisfactory Architecture Development

Figure 8: Proposed Migration Approach

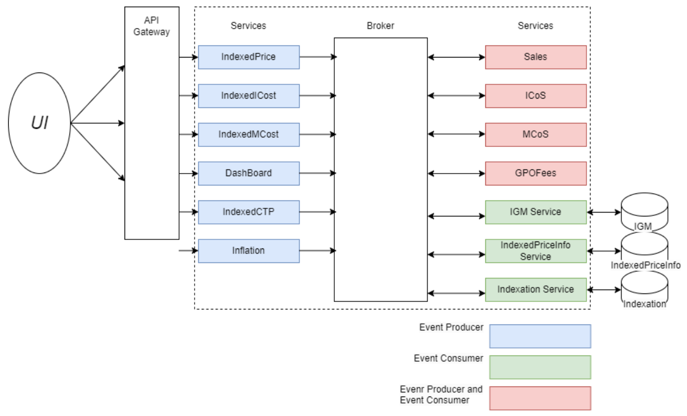


Figure 9: Overview of microservices architecture

3.2 Event-Driven Microservices

3.2.1 Design Principles and Quality Control. During the design of each microservice, the principles outlined below are followed:

DRY (Don't Repeat Yourself). The DRY (Don't Repeat Yourself) principle is implemented through the use of Service and Repository layers. This approach centralizes business logic in the Service layer and data access logic in the Repository layer, effectively avoiding code repetition across different parts of the application[39].

KISS (Keep It Simple, Stupid). In adherence to the KISS (Keep It Simple, Stupid) principle, the architecture features a clear separation of concerns across different layers. This organizational strategy keeps the architecture straightforward and manageable, with each layer handling specific operations, thereby simplifying both development and maintenance[16].

Single Responsibility Principle. Consistent with the Single Responsibility Principle, each layer in the architecture is designed with a singular focus, ensuring that it has only one reason to change. This approach enhances the system's modularity and maintainability[26].

Open/Closed Principle. The Open/Closed Principle is applied to ensure that while the system remains open to extension, it is closed to modification. This design philosophy allows for the expansion and adaptation of the system's functionalities without altering existing code[28].

This approach ensures that when the HBC team continues development in the future, there will be a smooth transition to production. Additionally, it minimizes the need for extensive refactoring and aligns with the success conditions listed in the Table 16.

3.2.2 Microservices Architecture. Following the proposed migration approach in Chapter 3.1, overall architecture of the microservices is emerged and shown in Figure 9. There are five components in this architecture: UI (user interface), API Gateway, Microservices, Event Broker and database.

UI. The User Interface (UI) of this project, developed using React, is designed for capturing user inputs and subsequently displaying the outputs generated by the microservices.

API Gateway. The API Gateway serves as the central entry point for all client interactions. It uses the NGINX Ingress controller to efficiently route requests to appropriate microservices. The Gateway also handles OAuth 2.0 authentication by verifying tokens from Microsoft Azure's identity platform, thus ensuring secure access. Additionally, it is configured for load balancing to enhance performance and reliability.

Microservices. The microservices architecture is based on the Spring Model-View-Controller (MVC) pattern, consisting of several distinct layers:

- The Controller Layer manages client requests, translating user inputs into actions to be performed by the Service Layer.
- The Service Layer contains core business logic and rules, generating responses according to user inputs.
- The Repository Layer directly interacts with the database, handling data persistence and retrieval.
- The Entity Layer represents the data model, mirroring the structure of the database entities.
- The DTO (Data Transfer Object) Layer serves as a means to transfer data between layers, especially from the Controller to the Service Layer, ensuring that only necessary data is communicated.

This is a common structure in Spring applications, where each layer has a distinct responsibility, enhancing maintainability, scalability, and clarity.

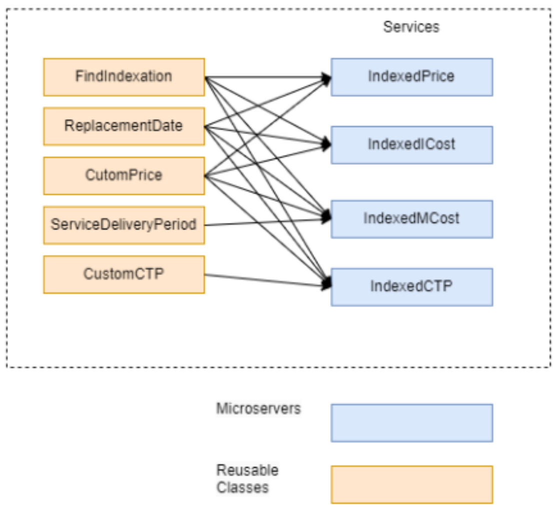


Figure 10: Reusable Classes

As initially outlined in the architecture skeleton in Chapter 3.1, each slide in the design PowerPoint was to represent an individual microservice. However, following continuous small refactoring cycles, the team decided to design certain calculations from the PowerPoint slides as reusable classes. This decision aligns more closely with the design principles discussed in Chapter 3.2.1. Table 5 details these classes, explaining their functions and uses. Figure 10 illustrates how these reusable classes are integrated within the microservices architecture.

Name	Description
ReplacementDate	Calculate the replacement date of equipment and customer service depends on contract information, equipment, and customer services information.
FindIndexation	Find the multipliers according to project information and each replacement date.
CustomPrice	Change price according to user need.
ServiceDeliveryPeriod	Calculate service delivery period of management cost (MCost) according to the project input.
CustomerCTP	Calculate the country target price according given threshold.

Table 5: Reusable Classes

Table 14 and Table 15 display the calculations designated to become individual microservices. Furthermore, Table 6 details the complete list of microservices in the proposed event-driven system. In alignment with the event-driven architecture discussed in Chapter 3.2.3, each microservice is specifically designed for asynchronous processing.

Equipment	Customer Services	Management Services	Others
Indexed price	Indexed price	Indexed price	Dashboard
Indexed CTP	Indexed CTP	Indexed CTP	IGM Calculation
Indexed icost prices	Indexed icost price	Indexed icost prices	IGM Percentage Calculation
Indexed mcost prices	Indexed mcost price		Inflation
Sales	Sales	Sales	
ICoS	ICoS		
MCoS	MCoS	MCoS	
GPOFees	GPOFees	GPOFees	

Table 6: List of Microservices in the Proposed System

3.2.3 *Event-Driven Architecture.*

In this event-driven architecture, the Broker topology is selected due to its lightweight nature and strong decoupling capabilities, which are essential for enhancing system scalability and flexibility. The event-driven structure is depicted in Figure 11 Regarding the event broker, two viable options were considered, each with its own set of advantages and aligning with different aspects of the system's requirements.

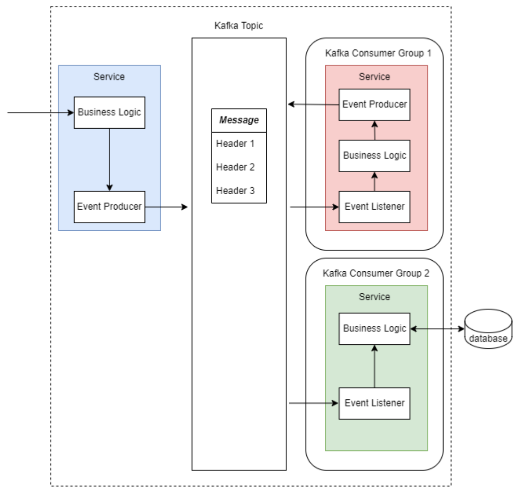


Figure 11: Event-driven Architecture

Apache Kafka. Apache Kafka is a distributed streaming platform primarily used in event-driven architectures. Its core abstraction is a topic, a feed where records are published and stored. Topics are split into partitions for scalability and parallel processing. Producers publish messages to topics, then it will evenly distribute messages among all available partitions. Only one consumer in a consumer group will receive message due to load balancing. However, the partitioning mechanism in Kafka presents a challenge for this project, especially since all calculation outputs need to be saved for the dashboard and some microservices produce intermediate results. Thus some of the microservices need to produce messages for multiple services.

To mitigate these challenges: First, placing each listener in a different consumer group ensures message distribution without overlap. Second, introducing message headers allows listeners to determine the relevance of a message before proceeding with processing

Spring Boot Event. The second solution for event-driven architecture is Spring Boot Event. It provides a robust mechanism for handling internal application events through its event publication and listening capabilities. When an event is published by the event publisher, it is dispatched to the Spring application context. This context acts as an internal broker, managing the flow of these events within the application. Subsequently, the event listener components are designed to listen for specific types of events. Upon catching an event, these listeners acts similarly like the listeners described in Apache Kafka architecture. The Spring Boot Event is default synchronous for handling events. However, it also allows components within the application to communicate asynchronously via extra configuration. This mechanism is crucial for applications needing internal, lightweight event handling without the overhead of external brokers.

These two solutions, Apache Kafka and Spring Boot Event, will be further evaluated in Chapter 4 to determine the most suitable option for this project.

3.2.4 Database and data management. In microservices architectures, especially asynchronous ones, managing data across multiple services poses significant challenges. Challenges like "retrieve data from multiple services" and "implement business transactions that maintain data consistency across multiple services"[41] would arisen when multiple microservices interact with the same data repository.

In this project, we face the same issue. Thus, ideally each microservice should manage its own database or data table. However, it would be difficult to manage and maintain the database with the amount of database or tables shown in Table 6. To address these, our project adopts a strategy where similar calculation outputs are aggregated, and microservices are tasked with specific save and read functions via events.

Ultimately, we utilize three main database tables to store calculation outputs, with each table managed by a single microservice. This approach helps maintain data consistency and integrity. Moreover, the ability to replay events in Apache Kafka is particularly crucial for ensuring data consistency and fault tolerance. In case a listener fails to process an event, it can be reprocessed, thereby contributing to the system's robustness and recovery capability.

3.2.5 Data Flow. In this project, the front-end user interface interacts with various APIs to retrieve data. This data, whether original or user-edited, serves as the input for the microservices. Therefore, the UI initiates the process by calling microservices APIs to transmit this data for calculations.

Upon being triggered by these API calls, the microservices perform the required calculations. Once the calculations are complete, they publish an event to the message broker. This action not only prompts downstream microservices to perform their respective functions but also triggers the saving of results into the database.

In the last step, the 'dashboard' microservices are activated to retrieve the calculated results. These results are then displayed on the user interface's dashboard, providing a visual representation of the data.

This sequence ensures a seamless flow of data from the UI to the microservices and back to the UI, facilitating a dynamic and interactive user experience

4 VALIDATION

This chapter presents the crucial validation phase of the design cycle, aiming to demonstrate how the proposed solution aligns with business requirements and supports stakeholder goals.

4.1 Software Quality Model

In software engineering, adhering to recognized standards like ISO/IEC 25010[6] is vital for the validation and evaluation of software quality. This standard provides a set of quality attributes crucial for assessing software products. Based on the business requirements detailed in Table 3, three key characteristics – performance efficiency, reliability, and maintainability – and their respective sub-characteristics have been selected for evaluating the two proposed event-driven microservices solutions discussed in Chapter 3.2.2. The correlation between the business requirements and these ISO 25010 characteristics is detailed in Table 7.

According to the business requirements stated in Table 3, the 'Interaction Capability' and 'Safety' characteristics from ISO 25010 are excluded because the proposed system is currently in the development environment and lacks a user interface. Thus, the interactions and their safety consequences between the proposed system and users cannot be measured. Besides, sub-characteristic like capability is also excluded because AWS will treat abnormal high load as DDoS attack and ban the IP temporarily. Subsequently, the most suitable sub-characteristic is selected for the assessment of each business requirement. The mapping between business requirements and ISO 25010 characteristics is shown in Table 7.

4.2 Software Testing

Software testing is vital in providing the development team with insights into software performance, ensuring that the web-based application meets quality standards before being deployed to production.

4.2.1 Testing Tools. Among various tools for testing web-based applications, Apache JMeter and Gatling have been chosen for comparison in this project, as highlighted by Paz and Bernardino[32]. These tools are selected due to their popularity and proficiency in the field. Although Apache JMeter scores highly overall, Gatling's asynchronous, non-blocking approach makes it more suitable for our project. This approach results in enhanced performance and resource efficiency, allowing Gatling to simulate more virtual users per unit of hardware. Additionally,

Sub-characteristics	Characteristics	Business Requirements	Evaluate Method
Time behaviour	Performance efficiency	BR2	Load Test
Resource utilization	Performance efficiency	BR11	Load Test
Fault Tolerance	Reliability	BR3	Architecture Review
Testability	Maintainability	BR1	Unit Test
Modularity	Maintainability	BR5	Architecture Review
Analysability	Maintainability	BR10	Architecture Review
Reusability	Maintainability	BR6	Architecture Review
Integrity	Security	BR5	Architecture Review
Confidentiality	Security	BR5	Architecture Review
Functional completeness	Functional Suitability	BR7 BR8 BR9 BR13	Validation with Pre-existing System
Functional correctness	Functional Suitability	BR13	Validation with Pre-existing System
Interoperability	Compatibility	BR4	Architecture Review
Scalability	Flexibility	BR15	Architecture Review

Table 7: Map ISO 25010 Characteristics with Business Requirements

Gatling's tests, which are written in Scala, offer greater flexibility for customization.

4.2.2 *Tests Design.*

Load Testing. The first step in designing the load test is to determine the target concurrency, which is essential to ensure that the system can handle the expected user load efficiently. For now, the total number of HBC users is fifty. And there are around one thousand potential target users within Philips. Using the formula from Czeiszperger[5], the concurrent users are estimated as follows:

(peak_hourly_visits * average_session_duration) / 3600

According to Table 3, the peak hourly visit can be calculated as:

The amount of all users / 16 (hour) = 62.5

The average session duration is set at 1 hour, and the calculated number of concurrent users is 62.5. Besides, each user will click and make an average of ten requests during the session. To assess the system's reliability, the test will simulate a scenario where there are 625 calls within one hour. As identified by Schroeder, Wierman, and Harchol-Balter[28], the proposed system is categorized as an open system, where new jobs arrive independently of job completions. Gatling's capabilities align well with open system testing requirements.The Gatling load test scripts, demonstrating the implementation of these tests, are included in Chapter 6, Listing 1.

Unit Test. This project adopts a Test Driven Development (TDD)[11] methodology, which involves creating unit tests prior to developing the actual code. This approach ensures that development is guided by pre-defined test cases, focusing on achieving the required functionalities through iterative testing and coding.

4.2.3 *Test Results.* In the Test Results chapter, we evaluate the system's performance under various conditions.

Time Behavior. The Time Behavior section focuses on measuring the system's response time under specific workloads. The results of the load testing, as outlined in Chapter 4.2.2, are summarized in Table 8. According to Table 3, both broker satisfies the response requirement.

Resource Utilization. Resource utilization refers to the efficiency with which the system uses resources like CPU and memory to perform its functions. During load testing, we closely monitor these parameters to evaluate how efficiently the system operates under load. The application, deployed in a Kubernetes cluster, allows us to use server CPU and memory usage as key indicators of performance. The results of this evaluation are presented in Table 9.

Kubernetes reports resource usage in specific units:

- CPU usage is measured in 'nanocores'. One core equals 1e9 nanocores. For example, a CPU usage of 601080303 nanocores, as seen in the Spring process, translates to approximately 0.601 CPU cores.
- Memory usage is reported in 'kibibytes' (Ki), where 1 KiB equals 1024 bytes. Hence, a memory usage of 882456Ki indicates that the process is using about 882.46 megabytes.

Event Broker	CPU Usage	Memory Usage
Spring	601080303n	882456Ki
Apache Kafka	431575551n	865680Ki

Table 9: Resource Utilization Matrix

4.3 Software Quality Analysis

4.3.1 *Confidentiality and Integrity.* From a confidentiality perspective, the microservices architecture utilizes an API gateway, as detailed in Chapter 3.2.2. This gateway plays a crucial role in safeguarding the microservices from unauthorized access. It achieves this by integrating with Azure application 'role' settings, a mechanism that strictly permits only authorized Philips personnel to access the microservices API. Consequently, any attempt at access by unauthorized individuals is systematically denied, enhancing the system's security posture.

Regarding data integrity, the microservices' layered architecture includes a repository layer that employs an ORM (Object-Relational Mapping) tool. This tool uses parameterized queries, significantly reducing the risk of SQL injection and thereby ensuring data integrity

Event Broker	**Error Rate(%)**	Response Time(ms)			
		Average	Min	Max	Std.Dev.
Spring	0	131	63	1232	173
Apache Kafka	0	202	98	922	116

Table 8: Time Behaviour Matrix

4.3.2 Maintainability. In the evaluation of the proposed system's maintainability, we use SonarQube[5]. SonarQube is an open-source platform used for continuously inspecting the quality of source code. The codebase achieved an 'A' rating in maintainability, which indicates a technical debt ratio of less than 5%. This excellent result suggests minimal corrective effort required, demonstrating the effectiveness of the adopted coding practices and adherence to the good practices mentioned in Chapter 3.2. The outcome not only highlights the software's readiness for future enhancements and modifications but also underlines the benefits of such maintainability in reducing costs and improving the stability of the software. This maintainability level supports the software's potential for long-term scalability and adaptability in response to evolving technological landscapes.

Testability. Test automation is a crucial part of the Continuous Integration pipeline. This pipeline automatically executes tests stored in the project folder, covering both unit and integration tests. The current test coverage stands at 83.06%, indicating a comprehensive testing process.

Modularity. The event-driven microservices architecture emphasizes modularity. Each microservice communicates via events, eliminating the need for individual services to be aware of each other's details. This approach ensures that changes in one microservice minimally impact others, enhancing system modularity.

Analysability. To improve analysability, the system employs extensive logging:

- In the Controller Layer: When the microservice API is triggered, it logs the request from the UI and publishes an audit event. These audit events are crucial for compliance reporting and analysis as they represent significant security-related actions or decisions within the application.
- In the Service Layer: Logging is utilized to document service errors. Various log statements categorize different types of failures, facilitating quick identification and resolution of issues.

Reusability. The proposed event-driven microservices system is designed for reuse within Philips. If another department within Philips wishes to leverage some of these microservices, they must undertake the following steps:

- Register a client application within the Azure portal.
- Ensure that the microservices application's API permissions are granted to the new client application.
- Connect to the existing Kafka Topic to consume the calculation output.

[5]https://docs.sonarsource.com/sonarqube

4.3.3 Fault Tolerance. As depicted in Figure 9, the microservices acting as event consumers depend on event producers. This dependency, however, does not compromise the overall system stability. If one microservice fails, the remaining services continue to operate. The primary impact is on those microservices reliant on the failed one, as they cannot perform calculations without the necessary input from the failed service.

4.3.4 Functional completeness and correctness. As outlined in Chapter 3, the HBC team employs a SCRUM methodology, working through user stories. Each story includes a screenshot displaying input and expected output. A user story is deemed complete and accepted when the output matches that of the Excel file. By the conclusion of this project, all user stories were completed and accepted, thus ensuring the system's functional completeness and correctness.

4.3.5 Interoperability. In the proposed event-driven microservices architecture, microservices communicate with each other via event messages. This structure facilitates the exchange and mutual use of information among services, simply by connecting to the existing Kafka topics. Such a design enhances the system's interoperability, allowing seamless integration with minimal configuration changes.

4.3.6 Scalability. As described in Chapter 2.1.8, the proposed system is deployed on AWS EKS with Kubernetes. This setup enhances scalability through horizontal scaling, which allows for adding application instances to manage increased load. AWS EKS supports automatic scaling and efficient load distribution, crucial for handling dynamic scaling. The architecture offers fine-grained resource control, ensuring optimal resource use and cost efficiency. Microservices on Kubernetes benefit from decoupling, allowing individual services to scale independently without affecting the overall application.

5 DISCUSSION

5.1 Answers to Research Questions

Main Research Question: How to migrate a monolithic system to a cloud-based event-driven microservices architecture system? The main research question regarding the migration of a monolithic system to a cloud-based, event-driven microservices architecture is comprehensively addressed through our proposed solutions detailed in Chapter 3. These solutions consist of a migration approach and an event-driven microservices architecture, tailored to the specific needs of the HBC team.

First of all, the HBC needs to be migrated to the cloud because the existing Excel structure and the manual processes surrounding it have limited HBC's efficiency and extended the time required to approve a business opportunity, thus hindering profitability. To automate these processes, the HBC team has decided to migrate HBC to a cloud service.

HBC team adapts SCRUM methodology, which emphasises iterative and incremental development. However, as highlighted in Chapter 2.5, existing software migration approaches do not align well with Agile principles, often involving excessive upfront effort in architectural transformation, contrary to building the architecture through continuous small refactoring[8]. Moreover, they also do not take microservices characteristics like decoupling and individually deployable into account. The debate on whether continuous small refactoring in Agile software development can lead to a satisfactory architecture has been longstanding. Chen and Barbar[13] have identified twenty key factors that are critical to this process. These factors significantly influence the success of developing a satisfactory architecture through the incremental refactoring method characteristic of Agile methodologies. To assess the suitability of this approach for our project, we conducted a detailed project-level analysis, including a survey.

The proposed software migration approach encompasses four phases, with each phase contributing uniquely to the project's success. Notably, the *Architectural Transformation* phase starts with an initial architecture skeleton, aligning with initial requirements. The *Forward Engineering* phase integrates Agile methodology with DevOps practices like Continuous Integration and Continuous Delivery, significantly enhancing project flexibility and reducing design cycles. Finally, in the *Satisfactory Architecture Development* phase, the architecture is thoroughly documented.

In summary, our proposed migration approach innovatively enhances existing software migration methodologies by integrating Agile and DevOps practices, thereby ensuring iterative development and continuous delivery.

The proposed event-driven microservices architecture integrates microservices with an event-driven approach, utilizing a broker to manage event communications. Producers within the microservices generate events, sending messages to the broker, while listeners consume these messages. Validation results indicate that although Spring Event offers shorter response times, its CPU usage is prohibitively high. This is attributed to Spring Event operating a broker internally, as described in Chapter 3.2.3, compared to Apache Kafka's external broker setup. The external broker setup, while potentially increasing response time, offers distinct advantages for our system:

- Apache Kafka is preferable for future scalability, especially for handling inter-service communication, a task that would be challenging for Spring Event.
- Lower CPU usage with Kafka reduces the risk of system failures.
- Apache Kafka's ability to scale by adding more topics offers flexibility for expanding the system.

The validation also confirms that the proposed software meets not only business requirements and user goals but also adheres to key software architectural principles. These include performance efficiency, reliability, maintainability, functional suitability, and security, thereby justifying our migration approach.

In conclusion, the proposed solutions will automate current business processes shown in Figure 2. They will replace the Excel-based system with a more efficient event-driven microservices architecture. This upgrade will automate key processes like 'Consolidate Quotes', 'Quotes Financing', and 'DashBoard Opportunities'. Additionally, the system is designed to enhance user experience and expand user groups, leveraging features like real-time response and scalability. Furthermore, it offers reusability for other Philips departments, allowing them to access calculation outputs through API authorization and participation in Apache Kafka Topics.

5.2 Limitation of Scope

Due to the extensive workload of the overall migration, only a subset of functions from the previous system has been migrated to microservices. Additionally, as the design of the user interface falls outside the scope of this project and remains incomplete, the following limitations have been identified:

- The application has not been deployed in a production environment by the end of this project. Consequently, treatment validation occurred only in the development environment, leaving the design cycle incomplete without the phases of treatment implementation and implementation evaluation. The availability and maturity of the proposed system, therefore, remain untested.
- Without an user interface, a comprehensive qualitative validation involving user interaction is not feasible.
- Load testing was conducted manually using Gatling. Integrating this into the Continuous Integration Pipeline could enhance early detection of performance issues, contributing to more stable and efficient deployments.

5.3 Future Work

Future research should focus on evaluating the migration approach from more perspectives that this project did not consider, for example, the interaction with users. There is also a need to explore the integration of emerging technologies, such as artificial intelligence, to automate parts of the migration process further, for example the architecture recovery and transformation. A longitudinal study could examine the long-term impacts of migration on system performance and business metrics. Additionally, future work could look into developing a toolkit that can cooperate with cloud technologies like CI/CD pipeline to evaluate a software system both qualitatively and quantitatively. Collaborations with industry could help refine the proposed methodologies, making them more robust and user-friendly. Finally, considering the rapid evolution of cloud technologies, ongoing research is required to keep the migration strategies up to date with the latest technological advancements.

6 CONCLUSIONS

This project successfully demonstrates a strategic migration approach from a monolithic system to a cloud-based, event-driven microservices architecture, significantly enhancing Philips' Harmonized Business Case (HBC) functionalities. This approach, characterized by continuous small refactorings within an Agile and

DevOps context, has proven to substantially improve operational efficiency and flexibility over traditional software migration methods. Notably, the implementation of automated workflows has optimized operational efficiency and potentially improving the profitability by shorten approval cycles. Furthermore, the decoupled nature of event-driven microservices allows for easier updates ,maintenance and better interoperability, enabling the system to adapt seamlessly to emerging business needs, technological advancements and service integration. This flexibility also facilitates the reuse of the system by other departments at Philips with similar functionalities. Future work will focus on refining the migration process to enhance efficiency. This research lays a robust foundation for future innovations in process automation and system integration.

This project follows the steps of a design cycle in *Design Science Methodology for Information Systems and Software Engineering*[40]. In Chapter 2, the *Problem Investigation* phase of the design cycle, we explored related works relevant to our main research question: 'How to migrate a monolithic system to a cloud-based event-driven microservices architecture system?' This investigation begins with a business process analysis to identify the manual flows in the current process. Subsequently, an ArchiMate model is used to depict the landscape of future process automation. Microservices and an event-driven architecture are then chosen as the desired design patterns following this analysis. This led to the identification of functional limitations, user goals, and a comprehensive list of thirteen business requirements, along with several constraints. It became evident that existing migration approaches were not fully compatible with the Agile methodology and DevOps practices, which are foundational to our project.

In Chapter 3, corresponding to the *Treatment Design* phase, we proposed a migration model inspired by the Horseshoe[22] and adapted Horseshoe[17] methodologies. This model is based on the theory that a satisfactory software architecture can emerge from continuous small refactoring, provided several key factors are met[13]. A survey was conducted to evaluate these factors in the context of our project, including aspects like team mindset and company culture. Consequently, we developed a software migration approach tailored for Agile and DevOps environments, resulting in an event-driven microservices system, detailed in Chapter 3.2.2.

Finally, Chapter 4 focuses on the *Treatment Validation* phase, where we conducted both quantitative and qualitative evaluations of the proposed software. The results affirm the emergence of a satisfactory architecture, thereby validating our software migration approach.

Despite these successes, the project encountered several significant challenges. These included the complexity of decomposing a monolithic system into microservices, designing an efficient event-driven model, and ensuring data integrity and system stability throughout the migration process.

A ABBREVIATION

Abbreviation	Full Name
EQ	Equipment
CS	Customer Service
CTP	Country Target Price
MS	Management Service
MCost	Marketing Cost
ICost	Industry Cost
RFV	Relative Fair Value
P&L	Profit and Loss

Table 10: List of Abbreviations

B HBC WORKFLOW

The workflow for the HBC model begins with users accessing the latest version from the HBC Teams folder. This initial step sets the stage for the subsequent data management process within the system. After downloading the model, users must establish a VPN connection to Philips' network and log into their Philips account to download the necessary master data. This data, along with specific project information input by the user, is crucial for executing the calculations.

Upon entering the required data in the sheets (see Table 11), users can initiate the VBA code calculations by clicking the 'Run Calculation' button. These calculations, detailed in Table 12, produce outputs such as profit and loss, which are displayed on the output sheets (refer to Table 13).

Upon completion of the calculations, users have the option to either upload their project information to the HBC database by clicking 'Upload Project' or to keep their data locally. It's noted that not all users opt to upload their data, with some choosing to share project information via email, leading to gaps in data availability for subsequent analysis and reporting.

Additionally, the system allows for the retrieval of previous projects through the 'GoldenID'. Users can download these projects for review and updates by clicking the 'Download Project' button.

In summary, the existing setup, which integrates the user interface, business logic (VBA code), and data handling all within the Excel environment, epitomizes a tightly coupled, single-tier architecture. This monolithic system structure presents inefficiencies in usage and scalability, highlighting the need for a more modular approach.

Sheet Name	Description
FrontPage	Philips internal front page including version
Project	Dashboard to control project inputs and to check if the project is perfoming in line with norms
Annual	Annual indexation and interest control
Custom	Empty customization sheet for users to perform analyses
Sofon-Apttus Input	Pricing inputs of equipment and services from Sofon-Apptus sources
Catalog	Pricing inputs of equipment and services
MgmtServ	Location to add additional services like staff, HTS, education and other
CAP	Location for capital asset plan and services incl legacy and room prep
PPU	Input sheet for PPU revenue profile (for both management best estimate and floor revenue)
Payments	Monthly payment profile to Philips from customer with option to adjust monthly payment

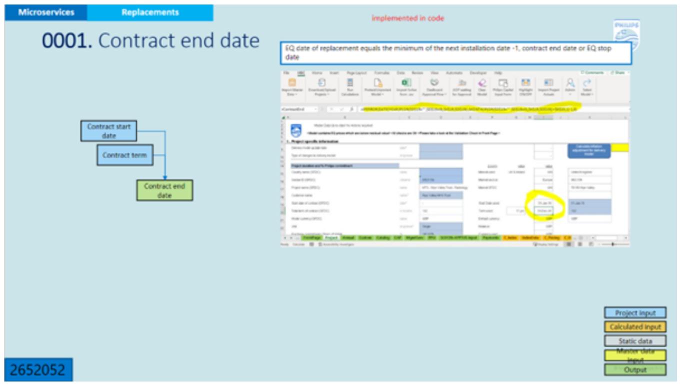
Table 11: Input Sheet

Sheet Name	Description
C_Index	Engine to calculate custom indexation for all MAGs
IndexData	Data dump of all index calculation per line item (MAG level)
C_Pricing	All pricing and lease classifications for all items
C_EQ.CS	All calculations for a single EQ and CS line item
C.MS	All calculations for a single MS line item
C_PPU_Excess	All calculations required for PPU type of offering and excess cash calculation
Data	Data dump of all calculation per line item
O_Prep	Preparation calculation for financial statement outputs

Table 12: Calculation Sheet

Sheet Name	Description
SoW	Share of Wallet - % calculation of Philips and non Philips assets and services
MgmtSum	IGM and Cash flow development as well as tornado charts
P&L RFV	Mixed P&L based on individual asset lease classification (this is the P&L recognised in the Philips books)
P&L_Offer	All calculations for a single EQ and CS line item
BS	Balance sheet - to determine max. on balance position
CF	Cash flow for either on or off balance financing
RFV_analysis	Analysis of the differences in relative fair values and fair values of equipment, customer services and management services
Project_vs_AOP	Comparision of market AOP targets, market AOP targets based on weighted sales of project and actual margins
EQ overview	Overview of equipment replacement over time
OIT	Expected Order Intake based on expected lead times on a quarterly timeline
LeaseSched	Overview of monthly principal outstanding per equipment line item

Table 13: Output Sheet



EQ date of replacement equals the minimum of the next installation date -1, contract end date or EQ stop date

Figure 12: Example of Business Logic Break Down

Equipment	Customer Services	Management Services
Indexation	Indexation	Indexation
Periods in contract	Periods in contract	Service period
Indexed price	Indexed price	Indexed price
Indexed CTP	Indexed CTP	Indexed CTP
Sales	Sales	Sales
Cost of sales	Cost of sales	Cost of sales
GPO fees	GPO fees	GPO fees
Indexed icost	Indexed icost price	Indexed cost

Equipment	**Customer Services**	**Management Services**
RFV allocation	RFV allocation	RFV allocation
Sales RFV	Sales RFV	Sales RFV
Operating expenses (market allocations)	Operating expenses (market allocations)	Operating expenses (market allocations)
AOP targets	AOP targets	AOP targets
Corporate income tax	Corporate income tax	Corporate income tax
Interest RFV		

Table 15: Calculations After Reallocation

C ISO 25010

The following definitions are used in this project and taken directly from ISO 25010:2013[6]. The precise wording is retained for technical accuracy.

Performance efficiency. This characteristic represents the performance relative to the amount of resources used under stated conditions.

- Time behaviour - Degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements.
- Resource utilization - Degree to which the amounts and types of resources used by a product or system, when performing its functions, meet requirements.
- Capacity - Degree to which the maximum limits of a product or system parameter meet requirements.

Reliability. Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time.

- Maturity - Degree to which a system, product or component meets needs for reliability under normal operation.
- Availability - Degree to which a system, product or component is operational and accessible when required for use.

Maintainability. This characteristic represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements.

- Testability - Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.
- Modularity - Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
- Analysability - Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.

D CHARACTERIZATION OF CONTEXT

	Factor	**Success Condition**
Project	Change	Medium to high rate of change
	Size	Small
	Type	Support small releases
	Maturity of AK	Mature Architecture Knowledge (AK)
	System Age	Green field
	Type of ASR	No demanding ASR that cannot be satisfied by refactoring
	Criticality	Low criticality
Team	Experience	Experienced
	Skill	Skilled
	Personality and Mindset	Willing to make change, learn, have passion, with dedication to good design
	Distribution	Collocated
	Team Size	Small
Practices	Safe Net	Automated testing with good coverage
	Continuous Integration	Continuous integration
	Good Design Principles	Applying good design principles such as DRY, SOLID, KISS
Organisation	Management	Management support and commitment
	Culture	Good communication channels, encouraging for taking ownership and commitment, open, blame-free
	Structure	Embraces the openness of Agile approaches.
	Governance	Proper architecture governance
	Maturity	Certain Level of Maturity

Table 16: Characterization of Contexts[13] for Evaluation

E TESTING SCRIPTS

Listing 1: Load Testing

```
val httpProtocol = http
  .baseUrl(URL)
  .header(header)
val scn = scenario("Load␣Testing")
  .exec(
      .post(api)
      .body(Request)
      .check(status.is(200))
  )
setUp(
  scn.inject(
    nothingFor(4.seconds),
    constantUsersPerSec(625 / 3600.toDouble) during (
    ).protocols(httpProtocol)
)
```

REFERENCES

[1] Microservices. https://www.martinfowler.com/articles/microservices.html.

[2] Microservices priorities and trends. https://dzone.com/articles/dzoneresearch-microservices-priorities-and-trends.

[3] *The Art of Unix Programming.* 2003.

[4] Stranglerfigapplication. https://martinfowler.com/bliki/StranglerFigApplication.html, 2004.

[5] Load testing basics: How many concurrent users is enough? https://www.webperformance.com/load-testing-tools/blog/2011/02/load-testing-basics-how-many-concurrent-users-is-enough/, 2011.

[6] Iso / iec 25010 : 2011 systems and software engineering — systems and software quality requirements and evaluation (square) — system and software quality models. 2013.

[7] Combining micro-services and event-driven architecture - a case study in philips, 2023.

[8] P. Abrahamsson, M. A. Babar, and P. Kruchten. Agility and architecture: Can they coexist? *IEEE Software*, 27(2):16–22, 2010.

[9] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi. From monolithic systems to microservices: An assessment framework. *Information and Software Technology*, 137:106600, 2021.

[10] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Migrating to cloud-native architectures using microservices: An experience report, 2015.

[11] K. Beck. *Test driven development: By example.* Addison-Wesley Professional, 2022.

[12] S. Bragagnolo, N. Anquetil, S. Ducasse, A. Seriai, and M. Derras. *Software migration: A theoretical framework (a grounded theory approach on systematic literature review).* PhD thesis, Inria Lille Nord Europe-Laboratoire CRIStAL-Université de Lille, 2021.

[13] L. Chen and M. A. Babar. Towards an evidence-based understanding of emergence of architecture through continuous refactoring in agile software development. In *2014 IEEE/IFIP Conference on Software Architecture*, pages 195–204, 2014.

[14] D. Clegg and R. Barker. *Case Method Fast-Track: A Rad Approach.* Addison-Wesley Longman Publishing Co., Inc., USA, 1994.

[15] A. Cockburn. *Crystal clear a human-powered methodology for small teams.* Addison-Wesley Professional, first edition, 2004.

[16] T. Dalzell. *The Routledge Dictionary of Modern American Slang and Unconventional English.* Routledge, 2009.

[17] P. Di Francesco, P. Lago, and I. Malavolta. Migrating towards microservice architectures: An industrial survey. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 29–2909, 2018.

[18] R. Dijkman, J. Hofstetter, and J. Koehler. *Business Process Model and Notation*, volume 89. Springer, 2011.

[19] H. Erdogmus. Architecture meets agility. *IEEE Software*, 26(5):2–4, 2009.

[20] J. S. Grace Jansen. Advantages of the event-driven architecture pattern. 2020.

[21] A. Josey, M. Lankhorst, I. Band, H. Jonkers, and D. Quartel. An introduction to the archimate® 3.0 specification. *White Paper from The Open Group*, 2016.

[22] R. Kazman, S. S. Woods, and S. J. Carriere. Requirements for integrating software architecture and reengineering models: Corum ii. *Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261)*, pages 154–163, 1998.

[23] S. Khriji, Y. Benbelgacem, R. Chéour, D. E. Houssaini, and O. Kanoun. Design and implementation of a cloud-based event-driven architecture for real-time data processing in wireless sensor networks. *J. Supercomput.*, 78(3):3374–3401, feb 2022.

[24] M. Kuhn and J. Franke. Smart manufacturing traceability for automotive e/e systems enabled by event-driven microservice architecture. In *2020 IEEE 11th International Conference on Mechanical and Intelligent Manufacturing Technologies (ICMIMT)*, pages 142–148, 2020.

[25] R. Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[26] R. C. Martin. Clean architecture : a craftsman's guide to software structure and design.

[27] A. Mavin, P. Wilkinson, A. Harwood, and M. Novak. Easy approach to requirements syntax (ears). In *2009 17th IEEE International Requirements Engineering Conference*, pages 317–322, 2009.

[28] B. Meyer. *Object-oriented software construction*, volume 2. Prentice hall Englewood Cliffs, 1997.

[29] J. Münch, F. Fagerholm, P. Johnson, J. Pirttilahti, J. Torkkel, and J. Järvinen. Creating minimum viable products in industry-academia collaborations. In *Lean Enterprise Software and Systems*, 2013.

[30] R. L. Nord, I. Ozkaya, and R. Sangwan. Making architecture visible to improve flow management in lean software development. *IEEE Softw.*, 29(5):33–39, sep 2012.

[31] E. T. Nordli, S. G. Haugeland, P. H. Nguyen, H. Song, and F. Chauvel. Migrating monoliths to cloud-native microservices for customizable saas. *Inf. Softw. Technol.*, 160(C), jun 2023.

[32] S. Paz and J. Bernardino. Comparative analysis of web platform assessment tools. In *WEBIST*, pages 116–125, 2017.

[33] A. Rahmatulloh, F. Nugraha, R. Gunawan, and I. Darmawan. Event-driven architecture to improve performance and scalability in microservices-based systems. In *2022 International Conference Advancement in Data Science, E-learning and Information Systems (ICADEIS)*, pages 01–06, 2022.

[34] M. Richards. Software architecture patterns. 2015.

[35] R. Sánchez-Reolid, D. Sánchez-Reolid, A. Pereira, and A. Fernández-Caballero. Acquisition and synchronisation of multi-source physiological data using microservices and event-driven architecture. In V. Julián, J. Carneiro, R. S. Alonso, P. Chamoso, and P. Novais, editors, *Ambient Intelligence—Software and Applications—13th International Symposium on Ambient Intelligence*, pages 13–23, Cham, 2023. Springer International Publishing.

[36] K. Schwaber. Scrum development process. 1997.

[37] V. Singh, A. Singh, A. Aggarwal, and S. Aggarwal. A digital transformation approach for event driven micro-services architecture residing within advanced vcs. In *2021 International Conference on Disruptive Technologies for Multi-Disciplinary Research and Applications (CENTCON)*, volume 1, pages 100–105, 2021.

[38] D. Taibi, V. Lenarduzzi, and C. Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017.

[39] D. Thomas and A. Hunt. *The pragmatic programmer.* Addison-Wesley Professional, 2019.

[40] R. Wieringa. Design science methodology for information systems and software engineering. In *Springer Berlin Heidelberg*, 2014.

[41] S. Zhelev and A. Rozeva. Using microservices and event driven architecture for big data stream processing. volume 2172, page 090010, 11 2019.