
Department of Mathematics and Computer Science
Interconnected Resource-aware Intelligent Systems Research Group

*Master Thesis*

# Designing an Evolvable Clinical Workflow System

Konstantinos Chanioglou
ID: 2053705

Supervisors:
Prof. Dr. J.J. Lukkien
Dr. Aly A. Syed

This thesis project was funded by Koninklijke Philips N.V.

Eindhoven, July, 2025

# Abstract

Nowadays, Clinical Workflow Systems (CWS) are key enablers for standardization and execution of patient care procedures across healthcare institutions. However, existing CWS designs often lack the ability to cope with new (clinical) requirements, while requiring code-level customization to adapt to different hospitals' unique workflows and environments. This thesis aims to understand how to prepare CWS architecture for future changes and it proposes a CWS architecture designed for evolvability, which mainly requires the CWS to support workflow configurability and environmental adaptability without code changes. Based on this analysis, a system architecture is proposed that emphasizes evolvability. A key design element is the use of modular Unit Functions (UFs), which encapsulate clinical tasks and allow workflows to be reconfigured through external configuration files, without modifying system code. The architecture also incorporates interoperability mechanisms (adaptors and a message broker) that support integration with external systems, ensuring the system remains adaptable across diverse clinical environments. For validating the proposed system design, a proof of concept implementation is developed and demonstrates the successful execution of configurable clinical workflows as well as system adaptability in different environments across various defined scenarios. Evaluation results confirm that the proposed system satisfies critical qualities related to CWS evolvability like configurability, adaptability, flexibility, integrability, modularity, maintainability, and others. In summary, this work presents a practical solution to the challenge of designing future-proof CWS architectures for diverse and evolving healthcare environments as well as a clear design methodology, key concepts and insights that can guide the development of evolvable systems.

# Contents

# Chapter 1

# Introduction

## 1.1 Problem Description and Motivation

Workflows are used in different industry sectors to standardize and optimize processes that must be executed repeatedly. They are used to define, in a clear and non-ambiguous way, the sequence of tasks that must be executed for a process to be completed. They are then executed each time the corresponding process needs to be carried out, achieving standardization. For example, a workflow in the customer care sector can define how a customer should be handled when contacting the customer care department. Such a workflow definition, for example, can describe that at the start of the customer-employee interaction, specific customer's personal identification details need to be verified, and then the employee asks for the customer's inquiry. Based on the inquiry, a specific series of tasks is defined and followed. So, by defining such a workflow and executing it each time a customer contacts customer care, the unit can have a standardized and optimized way of serving customers.

In healthcare, a workflow is called clinical because it involves a sequence of tasks that healthcare professionals or systems perform to deliver patient care within a medical setting. *Clinical workflows*, as Tanzini et al. [1] described, *refer to the sequence and interdependencies of tasks involved in delivering healthcare, encompassing the roles, actions, timing, and coordination of activities across clinicians and systems.* For example, a typical inpatient admission workflow involves a sequence of interdependent tasks, which can be the following. A patient is first assessed by a nurse, then evaluated by a physician, diagnostic tests are ordered, treatment is administered, and all action results are documented in the electronic health record. Clinical workflows are inherently dynamic and complex, influenced by contextual factors, interrelated processes, and adaptations to ensure safe and effective patient care. All these aspects of a clinical workflow are depicted in Figure 1.1.

The context shapes the environment, such as a hospital or clinic, by influencing interactions and processes. Temporality addresses the timing and sequence of tasks, impacting coordination, while aggregation highlights the interaction between workflow tasks and actors, revealing patterns. Next, actors (people or systems) execute tasks by performing actions and using artifacts (tools and technologies) that enable task execution. Actions refer to the execution of specific tasks by actors and can be described by their nature, for example, whether they are automated, or require manual effort. Finally, outcomes are the individual actions' results, providing insights into their effectiveness and impact. Combining the results of individual actions can assess the overall workflow effectiveness and impact. These elements highlight the complexity and dynamic nature of clinical workflows across various clinical settings, shaped by different organizational and environmental contexts.

Healthcare providers (hospitals) design their clinical workflows by consulting clinical workflow guidelines that are periodically published by various healthcare organizations, such as the WHO or other specialized communities, for example, the BC Sepsis Network [3], which publishes

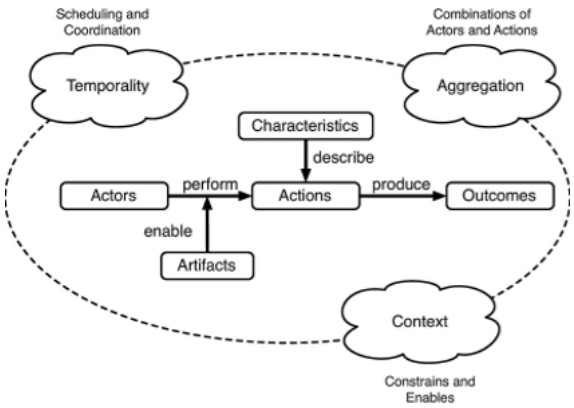



Figure 1.1: Clinical Workflow Description, Ozkaynak et al.[2]

guidelines specifically addressing the sepsis condition. For example, in Figure 1.2 is depicted a Sepsis Guideline Algorithm from the specific organization [4], that shows their suggestion on how caregivers should face the sepsis condition. This is an algorithm describing a sequence of actions (e.g. Check Heart Rate, Respiration Rate etc.) and conditions that need to be done. However, caregivers customize these proposed workflows based on their capabilities, regulations, best practices, and other factors that will be analyzed in section 2.2. As a result, workflow implementations vary across hospitals. For instance, suppose that a workflow measures a patient's vital signs and takes appropriate actions based on the results. In the case of a patient's vitals crossing a certain threshold (e.g., heart rate or blood pressure), the suggested workflow guidline implementation might be designed to trigger an alert for a nurse to inspect the patient's mental state. However, in academic hospitals, where professors and doctors want to evaluate new methods for better patient care and diagnostics, they are might in need of modifying the suggested implementation and add a different task, e.g. blood culture, to be executed by a nurse or another actor.

So far, it has been briefly explained what a clinical workflow is, that hospitals implement their workflows based on published guidelines, and that workflow implementation varies across hospitals due to different reasons. What has not been explained is that there are dedicated systems called Clinical Workflow Systems (CWS), which are responsible for implementing such workflow implementations, executing, managing, and monitoring them. Key stakeholders of such a system are the clinicians who monitor them and execute manual tasks of the workflows, hospital administration group which creates the new (custom workflow logic) requirements that need to be accommodated by the system, hospital IT administrations who monitor and manage, part of the system deployment and behavior, and finally vendors' engineers who are related with the system overall functionality, maintenance and customization. For the scope of this project, the relevant stakeholders were a clinical systems architect, acting as the problem owner, and a system designer. Such a system can exist in a patient monitor, which is a device responsible for continuously monitoring a specific patient using sensors attached to the body. In such a case, the CWS can be set to execute workflows that, under specific conditions (e.g., abnormal vital signs), raise alerts or automatically trigger other workflows for the specific patient, such as a sepsis workflow. In other cases, a CWS can exist in a centralized configuration to run multiple workflows for different patients at the same time. In either case, during workflow execution, clinicians interact with the system to monitor the state of workflows for different patients or to manually provide needed information (e.g. patient personal details) for the workflow execution to continue.

During workflow execution, for tasks to be completed, a CWS must communicate with various systems from the hospital environment to retrieve patient and other necessary information from different sources. A hospital environment can be described as a System of Systems, as it functions as a large system composed of multiple smaller subsystems. Based on Mehdipour and Zerehkafi[5], there are multiple healthcare information systems (HIS) and some of them are Clinical Informa-

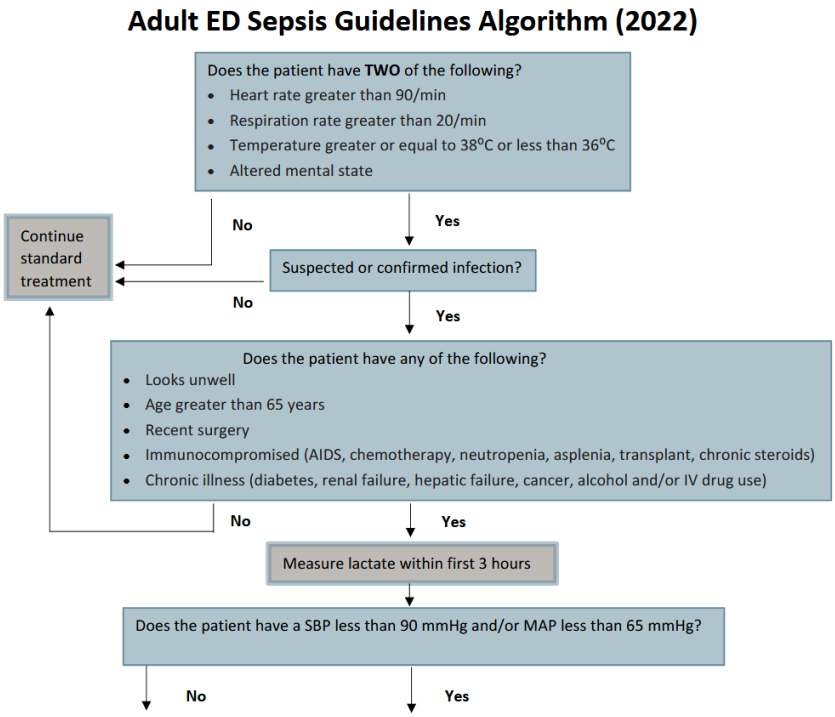



Figure 1.2: Part of Adult ED Sepsis Guidlines Algorithm 2022 [4]

tion Systems (CIS), Physician Information Systems (PIS), and Laboratory Information Systems (LIS). A CIS is a collection of computer based tools used to collect, store, and manage clinical data, helping clinicians make informed decisions and improve patient care. PIS specifically supports doctors by managing patient records, streamlining documentation, tracking treatments, and improving access to clinical data. These systems often use Electronic Medical Records (EMRs) and Electronic Health Records (EHRs). EMRs are digital versions of a patient's paper chart used mainly within a single healthcare provider's office or clinic, while EHRs are more comprehensive and can be shared across different providers. LIS helps manage lab operations by tracking samples, recording test results, and ensuring accurate and efficient handling of laboratory data. Next to these systems there are multiple others, and in various workflows tasks completion, the CWS must interact with these systems to retrieve clinical data, patient records, laboratory results and other needed pieces of information.

However, the set of Healthcare Information Systems (HIS) is far from standardized across caregivers' institutions. A characteristic example is the common assumption, by different studies proposing CWSs, that all hospitals have digital systems for recording patient data and health history (EMRs), or systems for sharing these records with other hospitals (EHRs). In reality, this is not always the case. According to several official reports and papers (ONC 2022, OECD 2023, [6, 7, 8]), while some hospitals in certain countries have transitioned to digital systems for recording or sharing patient data, this is still not the norm.

Considering the previously introduced points, workflow implementations differ across hospitals, and each hospital environment varies, we can now explain how these factors impact companies that develop medical systems. One such company can provide various healthcare technologies, such as patient monitors and other health information systems like centralized monitoring platforms. In these products, clinical workflows are executed. The CWS, as described, exists as a subsystem within these products to undertake all responsibilities related to workflow implementation, execution, management, and monitoring. Based on expert interviews, existing CWS implementations hard-code clinical workflows into the system, which limits adaptability and increases maintenance complexity. When a customer acquires such products, the part of the CWS responsible for work-

flow implementation must be customized to meet the specific needs of that hospital. This results in workflow implementation code re-engineering, extensive testing, significant time and financial investment, and most importantly, an inability of the CWS to easily adapt to each customer's unique workflow and system requirements at the time of delivery, leading companies to develop different releases of CWS for each customer. Additionally, healthcare organizations frequently update clinical workflows. As a result, hospitals often want to evolve the implementation accordingly, which creates the need for further CWS's specific releases maintenance and re-certification even after initial customization. The same problem of requiring CWS customization comes from the fact that CWS at each hospital needs to be customized to adapt to the specific hospital available systems.

Although current CWSs can address the individual needs of caregivers, each time a customization request arises, the company needs to collect the new requirements, analyze them in code terms, implement the changes, and test the entire system again before delivering the updated version of the product. All these factors lead to two identifiable problems. First, the CWS implementation must be customized to adapt to the workflow and environment of each hospital, which triggers an extensive verification and validation process. Second, the company must maintain a separate CWS release per hospital. These problems arise from limitations in the architectural design of existing systems, and highlight the need for an evolvable CWS that can adapt to varying customer workflow and environment requirements and changing demands over time without code customization per hospital. This study aims to propose a new evolvable CWS design that allows workflows to be changed and CWS to adapt to environmental variability without requiring system code-level customization, that is, changes made directly to the system's source code, such as modifying or recompiling components, thus avoiding the need to re-certify or maintain different system releases for each hospital. This will be achieved by collecting requirements, designing the system architecture, developing a prototype, and testing its evolvability to ensure it meets the diverse and changing needs of healthcare environments.

Building on this objective, the current study focuses on utilizing the configurability notion to shift the implementation of clinical workflows from the code level to the configuration level, allowing system adaptation to new requirements without modifying the underlying code. In doing so, it investigates how architectural design can support system evolvability. A system architecture will be designed to execute these workflow definitions while managing and monitoring their execution and accommodating environmental variations across caregivers. In other words, the proposed system should enable configurable workflows, adaptation to different hospital environments, including integration with diverse clinical technologies, tools, and infrastructure.

# Chapter 2

# Literature Review

To address this problem, it is essential to gain a deep understanding of both the context and the problem itself, and so the problem has been analyzed from two main points of view. Firstly, we must explore what it means and what is required to build an evolvable system. Secondly, we must dive into the problem's specific context, including clinical workflows and workflow systems, and examine whether solutions already exist. To achieve this, a literature review has been conducted on system evolvability, followed by a more focused review of clinical workflows and the configurability of clinical workflow systems. In the following two sections (section 2.1, section 2.2), two tables (Table 2.1, Table 2.3) presenting key studies in these fields are provided and explained. In these tables, each row represents a study from the literature, while the columns represent the study's primary focus. So, the symbols "+" indicate the focus of each study.

## 2.1 System Evolvability

Considering the first of the two topics, a taxonomy of different approaches studying system evolvability has been created and depicted in the Table 2.1.

| System Evolvability Papers | Evolvability Assessment Performed | System Architecture Considered | Requirements/ Infusion Effort Considered | Case Study Included | Literature Review Conducted | Empirical Study Conducted |
|---|---|---|---|---|---|---|
| Borches & Bonnema 2010 [9] | | + | | + | | |
| Borches & Bonnema 2009 [10] | | + | | + | | |
| Sullivan et al. 2018 [11] | | | | | + | |
| Suh et al. 2008 [12] | | | + | | | |
| Halilovic 2024 [13] | | | + | | | |
| Boyang 2024 [14] | + | + | + | + | | |
| America et al. 2010 [15] | | + | | + | | |
| Ross And Rhodes 2008 [16] | + | | + | | | |
| Anda 2007 [17] | + | | | | | |
| Colombo et al. 2016 [18] | | | | | + | |
| Fricke & Schulz 2005 [19] | + | + | | | | |
| Urken 2012 [20] | | | | | + | |
| Steiner 1998 [21] | | + | | | | |
| Luo 2015 [22] | + | | | | | |
| Wang et al. 2023 [23] | | | + | | | |
| Khan and West 2018 [24] | | | + | | + | |
| Selberg and Austin 2008 [25] | | + | | | | |
| Bogner et al. 2019[26] | | | | + | | + |
| Bogner et al. 2018[27] | | | | + | | + |
| Bochres & Bonnema 2008[28] | + | | | + | + | |
| Lowe et al. 1998 [29] | | + | | | + | |
| Falclarin et al 2003 [30] | | + | | | | |
| Krikhaar 1997 [31] | | + | | + | | |

Table 2.1: Literature in System Evolvability

### 2.1.1 Evolvability Literature Review Studies

Firstly, some studies conduct literature reviews (fifth column in Table 2.1) to provide clearer definitions and offer a more structured understanding of the term and revealing insights from the literature in this topic. While evolvability is still not clearly defined, it is generally considered as the ability of a system to accommodate changes with minimal cost while maintaining its architectural integrity. Urken et al. [20] reveals general insights into the integration of robustness, resilience, and sustainability to enhance the evolvability of engineered systems, and they emphasize the need for a new modeling approach that allows for adaptive behavior and emergent properties. By this framework architects can be assisted to design systems that are not only functional but also capable of evolving over time in response to changing environments and requirements. Khan and Wuest [24] propose a framework for upgradable systems and identifies key attributes that contribute to a system's evolvability focusing on upgradability. Additionally, Sullivan et al. [11] classify changeability as a crucial component of evolvability and analyze it through this prism in order to make the system able to host changes that increase the value of the system. It can be inferred from the previous studies that each one of them explore evolvability through the prism of different "ilities". For example, Urken et al. [20] explore evolvability through robustness, resilience, and sustainability, Khan and Wuest [24] through upgradability and Sullivan et al. [11] through changability. While these "ilities" are interesting, the current study differs from all of these while the main focus is to build an evolvable system by making it configurable and adaptable to the new workflow and system environment requirements.

Colombo et al.[18] synthesize and compare nine prior works that define change-related system properties ("ilities"), such as flexibility, adaptability, and evolvability. While most works provide definitions and informal groupings, they do not offer a consistent method to classify "ilities". Colombo et al. address this gap by categorizing "ilities" based on engineering change features, such as the lifecycle phase in which changes occur, the object being changed, and the type of change initiator. This unified perspective clarifies semantic overlaps and supports the precise application of "ilities" in system design. However, while Colombo et al. define evolvability strictly as changes to the system's abstract architecture during the (re)design phase, the present work adopts a broader and more pragmatic definition, as mentioned earlier. Evolvability is considered the system's ability to accommodate changes with minimal cost while preserving architectural integrity. This perspective better reflects the design goals of the proposed CWS architecture, which must support new requirements without significant architectural changes.

Also, Borches and Bonnema [9] present another aspect of system evolvability as they support that the biggest obstacle to system evolvability is the lack of shared understanding while systems are designed and developed by different teams. These papers propose methods to document system architecture to enable system evolvability. Lastly, Lowe et al. [29] constructs a comprehensive taxonomy on system evolvability and its relationship with system architecture. They identify three key categories within the taxonomy of evolvable systems. In the best case, the category of evolvable systems can handle the changes without changes in the current system architecture. Next to this is the assimilation category, where modifications need to be made to specific components within the existing architecture. In the worst case, it is the architectural changes category, in which practically the system cannot handle the required changes without changing the system architecture. Additionally, the study highlights four key attributes (generality, adaptability, scalability, and extensibility) that directly influence system evolvability. These insights clarify that the proposed definition of evolvability for the CWS aligns well with the literature and that the associated attributes can be used as criteria for system evaluation.

### 2.1.2 Evolvability on System Architecture Studies

Other studies focus on exploring evolvability from the system architecture point of view (second column in Table 2.1). These studies focus both on how to build architectures that support change, and on the underlying principles that explain what makes an architecture evolvable. To start with, Borches and Bonnema's works [9, 10] identify that one important barrier in system evolvability

is the lack of formal ways of knowledge sharing about the system design and thus they focus on the challenges of designing systems that can evolve over time, with a particular emphasis on communication and knowledge sharing within complex systems. Borches and Bonnema's in 2009 publication [10], and Krikhaar [31] as well, they introduce the concept of reverse architecting to support system evolution through the reconstruction of the system's architectural representation when it is undocumented or unclear. This process enables teams to understand the implications of design changes and manage better system evolution. Additionally, Borches and Bonnema's in their 2010 publication [9] propose the A3 Architecture Overview method. This aims to capture and share of architectural knowledge and they use the Philips MRI system as a case study. More specifically, the method aims to create manageable architectural representations that enhance communication and collaboration across multidisciplinary teams. America et al. [15] propose a solution to create a reference architecture that is capable of sharing understanding to support the evolution of systems. The authors emphasize the need for cost-benefit analysis and the incremental application of architectural changes to ensure that the system remains adaptable over time. These studies highlight the importance of modularity, effective communication, and architectural understanding, often achieved through documenting or reconstructing undocumented systems, as critical enablers of knowledge transfer and long-term system evolvability, particularly in healthcare technologies. While the current work does not focus on reverse architecting, it takes into account these insights to emphasize the role of intentional architectural documentation and modular design in enabling the CWS that will be proposed to support knowledge sharing and sustain system evolution over time.

From another point of view, Fricke and Schulz [19] introduce the Design for Changeability (DfC) concept. The concept focuses on creating systems capable of evolving throughout their lifecycle. The authors define four key changeability aspects which are robustness, flexibility, agility, and adaptability. To support this, they propose a set of architectural principles such as modularity, independence, and simplicity. Based on the key aspects the system being developed needs to focus on, they propose which principles the design focus should be on. Fricke and Schulz's study is directly related to the current study and provides principles that can be used during the designing and evaluating phase. For example, as the current study aims to propose a system that should be capable of adapting to different hospital environments, some of the principles that should be taken into account during the designing phase are simplicity, modularity, decentralization, and others.

Steiner [21], like Lowe et al. [29], creates a taxonomy of system architectures which includes the categories enduring, evolutionary, growth, and specific architectures. He supports that enduring architectures maintain key system characteristics over time, while evolutionary architectures manage changes across product generations. Next, growth architectures focus on specific designs within a generation, and specific architectures detail the implementation. A suggestion from this paper is that when evolutionary architecture is selected to be used, it is important to include characteristics such as modularity, flexibility, and scalability to ensure the system can adapt to future changes while maintaining core functionality. Also, Selberg and Austin [25] explore the complexities of Systems of Systems (SoS), proposing that architectural frameworks must evolve to handle increasing complexity and decentralization.

Boyang Zhang's master's thesis [14] proposes a Stepwise Architecting for System Evolvability (SASE) method. This is a framework that helps on evaluation and implementation of architectural changes to enhance system evolvability. Boyang built on Edin Halilovic's earlier work [13], and the SASE method extends the framework to offer a stepwise approach to anticipate future requirements and identify which, where, and how architectural modification should take place. Also, the SASE method helps system architects balance flexibility with the risks of over-design and ensures that systems can adapt to evolving demands and challenges throughout their lifecycle, a characteristic that leads to more resilient and adaptable system architectures. Another interesting approach by Falcarin et al. [30] presents a methodology for dynamic architectural changes in distributed services using the JADDA component. This component allows systems to adjust their architecture at runtime without requiring changes to the application code, enabling more agile and adaptable system development. The paper demonstrates the application of this approach using xADL (XML Architecture Description Language) to define and modify system architectures, showcasing how

systems can evolve and adapt to changing requirements with minimal disruption.

Overall, these studies explore evolvability from the system architecture perspective, emphasizing the need for modular, flexible, and adaptable architectures to support evolving systems. Papers by Borches and Bonnema [10, 9], America et al. [15], Krikhaar [31], Fricke and Schulz [19], and Steiner [21] introduce methods like reverse architecting, A3 Architecture Overview, Design for Changeability (DfC), and a taxonomy of architectures to improve system evolvability, while Boyang Zhang's thesis [14] presents a stepwise framework for anticipating future requirements, and Falcarin et al. [30] showcase dynamic architectural changes using JADDA to enhance agility and adaptability in distributed services. Even though all these studies focus on system evolvability and offer a diverse overview of relevant challenges, Fricke and Schulz's work provides valuable insights into what aspects (flexibility, adaptability) and design principles (modularity, integrability, autonomy) the current study should focus on. This aligns well with the goal of the current work, which is to design a CWS that can adapt across diverse hospital environments by supporting configurable workflows and environmental adaptability. The principles proposed by Fricke and Schulz directly support this aim by guiding architectural choices that enable adaptation and long-term system evolvability.

### 2.1.3 Evolvability Empirical Studies

System evolvability is also analyzed by empirical studies point of view (sixth column in Table 2.1). Specifically, studies by Bogner et al. [27, 26] focus on service-oriented architectures while examining industry practices to ensure their evolvability. Bogner et al. identify challenges such as ensuring low coupling and high cohesion among services and the lack of practical tools for measuring evolvability. Additionally, in their second study [26], they propose a method for continuous evolvability assurance in Service-Based Systems (SBS) using service-oriented metrics and evolution scenarios. This method aims to ensure that microservices are capable of evolving through the use of design patterns and tool support and emphasizes the need for improved evaluation methods to address evolvability in this category of systems. However, while some aspects of Bogner et al.'s research are related to higher-level system notions like architectural design, on which the current study focuses, their main emphasis is primarily on software implementation practices, indicating a gap in addressing how these practices fit into the overall architectural frameworks that govern complex systems.

### 2.1.4 System Evolvability Requirements Studies

As requirements analysis is one of the most important steps for building evolvable systems, several studies focus on system requirements analysis to ensure the evolvability of complex systems (third column in Table 2.1). Wang et al. [23] emphasize the importance of understanding customer needs for system upgrades by developing a four-step method that includes scenario building and requirements transformation. Their method involves the identification of the core components of the product service system (PSS), analysis of future trends with PEST-OT matrices, and capturing customer requirements through persona-based scenarios. Similarly, Khan and Wuest [24] propose a framework for designing upgradable PSS, and the need for modularity, continuous validation of customer requirements, and integration of product and service upgrades are highlighted. Additionally, Boyang and Halilovoc [14, 13] use the ESCE Method to predict future changes and PESTEL to identify external factors that might influence how the requirements of the system could change over time and discuss methodologies for evaluating system changeability, offering methods for incorporating future requirements into existing system architectures. Also, an important aspect of requirement analysis is the infusion effort. Suh et al. [12] propose a technology infusion framework to assess the impact and implications of infusing new technologies into existing systems. Their methodology uses a Design Structure Matrix (DSM) and a Delta DSM to quantify the engineering changes caused by the new technology, estimate infusion costs, and evaluate the

market impact. The study also demonstrates the framework through a case study of the printing industry. It shows how the methodology can be used to evaluate technology infusion efforts and ensure that technological upgrades can be successfully integrated into existing systems.

The above studies provide a straightforward way of working in requirement collection and analysis before the system design phase to ensure that all the foreseen requirements are identified and analyzed. So, in the current study, such techniques will be used. More specifically, after defining the core system requirements, PESTEL analysis will be used to identify potential future requirement changes on the system, as well as scenario generation for constructing a clear understanding of how such requirements impact the system. Finally, continuous customer validation will ensure alignment with the customer's needs.

### 2.1.5 Evolvability Assessment Studies

Other studies have tried to provide frameworks for assessing system's evolvability and applied approaches for what should be done to assess and quantify this property (first column in Table 2.1). Anda [17] focuses on the combination of structural code measures and expert assessments. Their paper emphasizes the challenges in assessing evolvability for complete systems, as opposed to individual software components, and illustrates this through a case study involving four functionally equivalent systems. The authors underline that structural code measures alone may not fully capture evolvability and suggest to combine them with expert opinions to better predict the future effort required for system evolution. The findings suggest that a comprehensive assessment of evolvability should include both quantitative metrics and qualitative insights from experienced professionals. Additionally, Luo [22] offers a simulation-based method to assess the impact of product architecture on evolvability, using a design structure matrix (DSM) to model and assess the influence of component interdependencies on future design changes. Similarly, Fricke and Schulz [19] use the DSM approach to assess evolvability as well. Also Borches and Bonemma [28] mention that they use a pro/con list to approach a formal framework of evolvability assessment while a formal one doesn't exist. Ross and Rhodes [16] focus on conceptualizing system timelines and value delivery using Epoch-Era Analysis. This method examines how systems evolve over time by considering shifting contexts and expectations by using natural value-centric time scales to manage system change. The paper highlights the importance of understanding dynamic value and how systems must adapt to meet evolving needs and challenges across multiple epochs. Lastly, previously mentioned Boyang Zhang's [14] work on the SASE Method, which includes a step for architectural concepts evaluation to determine which design approach will best support the system's evolvability. The method compares multiple architectural concepts against criteria such as modularity, flexibility, and scalability and with a score based assessment choose the best one to be applied on the existing system architecture and thus ensures that the system will be capable of accommodating future changes.

### 2.1.6 Case Studies

Finally, case studies play a critical role in providing practical, real-world solutions for constructing evolvable systems by focusing on specific systems and apply methods to enhance their evolvability (fourth column in Table 2.1). Some of the already mentioned papers showcase and verify their proposals on specific systems. Borches and Bonnema [10, 9] demonstrate the use of architectural overviews to communicate and manage system changes, specifically in a complex MRI system at Philips Healthcare. Also, in [28] the authors present a case study on reverse architecting applied to the Philips MRI system, to demonstrate how the architectural knowledge of complex systems can be captured and used to support their evolution over time. Krikhaar [31] provides a case study of reverse architecting applied to complex systems, using examples from Philips' medical and communication systems to show how existing system architectures can be re-engineered to improve their adaptability to future requirements. Bogner et al. [27, 26] focus on the evolvability of microservice-based systems. They explore industry practices and challenges and present insights

through service-oriented metrics and evolution scenarios for microservices. Boyang Zhang [14] applies his proposed SASE method in a case study to demonstrate how the framework can guide system architects in anticipating future requirements and modify the architecture to enhance evolvability.

### 2.1.7 Key Takeaways

To summarize the insights from these studies, Table 2.2 presents various system qualities, commonly referred to as "ilities" in the literature, that are closely associated with system evolvability. Although evolvability itself is described in the table, it is often considered an emergent property arising from a combination of other qualities. Given the specific aim of the system to be designed, this study focuses on a subset of these qualities. The literature suggests that the most relevant to evolvability include adaptability, scalability, extensibility, modularity, robustness, flexibility, and agility. However, the final selection will depend on the problem context and will be determined after the system requirements have been identified.

| **Ility** | **Description** |
|---|---|
| Adaptability | Ability of a system to adapt to a changing environment. [18], [19], [29] |
| Flexibility | Ability of the system to be changed easily. [19] [18], [29] |
| Scalability | Ability of the system to change in size to support something. [18] |
| Extensibility | Ability of the system to fulfill new (set of) functions. [18], [29] |
| Modularity | Degree to which a system's components can be separated and recombined. [19], [9] |
| Robustness | Ability of the system to be insensitive to changing environment. [19], [20] |
| Agility | Ability of the system to be changed rapidly. [19] |
| Upgradability | Ease of improving the system with newer components or versions. [24] |
| Sustainability | Ability of the system to remain maintainable and valuable over time. [20] |
| Resilience | Capacity of the system to recover from disruptions or failures. [20] |
| Configurability | Ability of the system to be customized via settings or parameters. |
| Changeability | Umbrela of different qualities related to system change ability. (Robustness, Flexibility, Agility, Adaptability). [19] |
| Integrability | Ability of the system to integrate new external components or systems. Related to compatibility and interoperability. [19] |
| Autonomy | Ability of system's components to operate independently and manage themselves. [19] |
| Generality | Ability of the system to accommodate changes within the existing design. [29] |
| Evolvability | Ability of a system to accommodate changes with minimal cost while maintaining its architectural integrity. [29] |

Table 2.2: System Qualities Related to System Evolvability

This study will utilize methods from the previous literature review. To be more specific, starting from the requirement collection methods, such as PESTEL and scenario generation, as proposed by the above studies, will be used to ensure that both current and future requirements of the system are identified and included in the system design. Considering the system design phase and evaluation, different aspects of the stated system qualities will be selected for use according to the requirements that will be identified. Based on the chosen qualities, the correct architectural concepts will be applied to the system design proposal. Finally, a scenario-based evaluation will be conducted for the system, as proposed in previous studies.

## 2.2 Clinical Workflows and System Architecture for Clinical Workflows

Next to the literature review on the system evolvability, the second important part that needs to be covered to understand the problem correctly is why the clinical workflows differ among care providers and why the current systems cannot cover the need for easy customization of CWS. Thus, a literature review has been conducted on the topic of clinical workflow and previous propositions for solving this problem as well as for the system structure of hospitals in order to understand how CWS interacts with other systems from the hospital environment. The papers that have been reviewed in this context are included in Table 2.3, on the same way as in the previous section, and a taxonomy of studies has been created as shown.

| Health Care Systems / Workflows Studies | Workflow Modeling and Formalization Discussed | CWS Architecture or Components Discussed | Causes of Clinical Workflow Variation Identified | Real-World Clinical Workflow Guidelines Examples |
|---|---|---|---|---|
| Huser et al. 2011 [32] | + | + | | |
| Quaglini et al. 2000 [33] | + | | | |
| Ferrante et al. 2016 [34] | | | + | |
| Kennedy et al. 2010 [35] | | | + | |
| Quaglini et al. 2001 [36] | | | + | |
| Tummers et al. 2021 [37] | | + | | |
| Mehdipour and Zerehkafi 2013 [5] | | + | | |
| Dadam et al. 2000 [38] | + | + | | |
| Carvalho et al. 2010[39] | + | | | |
| Peleg and Haug 2023 [40] | + | | | + |
| Peleg 2013 [41] | + | | | |
| Ozkaynak et al. 2016 [2] | + | | | |
| Carter and McGill 2022 [42] | | | | + |
| Habib et al. 2015 [43] | | | | + |
| Fitch and van De Beek 2007 [44] | | | | + |
| Konstantinides et al. 2020 [45] | | | | + |
| Mustafa et al. 2023 [46] | | | | + |
| KDIGO 2024 [47] | | | | + |
| HealthQualityBC 2022 [4] | | | | + |
| Byrne et al. 2023 [48] | | | | + |
| SWAB 2024 [49] | | | | + |

Table 2.3: Literature in Clinical workflows and CWS

### 2.2.1 Clinical Workflow Variations

The first category of studies focuses on why clinical workflows vary from the proposed guidelines (third column in Table 2.3). The reasons can be categorized into four main categories: medical knowledge, practices and regulations, organizational factors, decision-making, and human factors, and lastly, system and social factors. To begin with, one factor is the local adaptation of national or international guidelines to better suit the hospital's specific context, with hospitals modifying procedures based on local practices and regulations [34], knowledge [33], and clinical leadership and expertise [34, 35]. For example, a specialized cardiac center may design workflows based on local expertise to reflect their unique experience. A second factor is organizational structures [33], resource availability [33, 35], and infrastructure and support systems variances [35]. For example, a large academic hospital with complex hierarchies may design more intricate approval workflows compared to a smaller clinic that would structure their workflows in a more straightforward way and, of course, adapt it to their own availability of diagnostic machines or other mechanisms. The third factor is human factors and decision-making [33], which produce different workflows. For example, in hospitals where shared decision-making is emphasized, workflows may be modified to include additional steps for patient and family consultations. Last but not least, a fourth factor is the economic, political, and social aspects [35]. For example, budget constraints may lead hospitals to adjust workflows, such as adding cost-approval steps when prescribing medications, to comply with national healthcare policies. Moreover, new clinical insights may lead to workflow modifications driven by multiple factors such as cost efficiency, clinical outcomes, the patient

population, or the patient's clinical condition. These insights could be customer-specific or global, such as those enforced through regulations.

So, in Figure 2.1, Figure 2.2, it is depicted that caregivers consult proposed workflow guidelines but also consider various other reasons that lead to customizing and producing new workflow versions, supporting the fact that workflow varies across hospitals. More specifically, in Figure 2.1 is depicted that formalized guidelines are constructed based on medical knowledge. The main point that workflows vary across hospitals is that hospitals with specific organizational structures combine formalized guidelines and their organizational structures to produce custom workflow models. Then, the workflow model is used for simulations, whose results allow for the finding of optimal resource allocation. Eventually, the workflow may be enacted in the field. In Figure 2.2, it depicted a different iterative approach, highlighting the workflow variation across hospitals. Firstly, hospitals gather knowledge and, based on that, create workflow models. Step three shows that created workflow models are combined with guidelines workflow knowledge as well as with the local hospital environment on which workflows may interact (in this case, EHR is mentioned). So again, it is clear that guidelines are combined with the local customized needs of the hospitals. After validation and verification, workflows are deployed, executed, and optimized. Another interesting point is that at step seven, insights from the workflow model and execution are shared with the community, and this creates an iterative process because it gives input to the workflow modeling process again. Based on these points, it is clear that hospitals consult shared information and guidelines from the community and, combined with their own expertise, knowledge, and organizational structure, produce the final workflows.

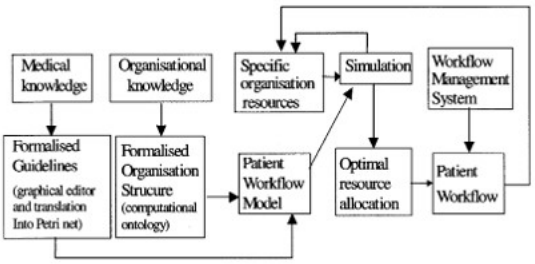



Figure 2.1: Clinical Workflow Variation [33]

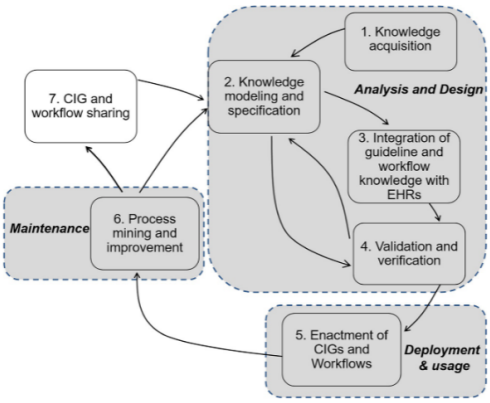



Figure 2.2: How Clinical Workflows are constructed and source of variation [40]

### 2.2.2 Clinical Workflow Guidelines

Other studies from health organizations focus more on proposing clinical workflow guidelines (fourth column in Table 2.3), a representative example being the guideline shown in Figure 1.2. While this is not the primary focus of this study, these guidelines still need to be examined to understand their structure and how they can be modeled and executed using a CWS, which is one of the goals of this study. Clinical workflow analysis and relevant conclusions will be presented in detail in chapter 3, as this is one of the key requirements of the current study.

### 2.2.3 CWS Architecture Proposals

Now, let's focus on studies that proposed CWS architectures (second column in Table 2.3) to understand the already existing solutions in this field and whether parts of them could be utilized to be applied in our specific case to enable configurable workflows and, at the same time,

evolvable system design. Huser et al. [32] investigate workflow engine technology for clinical decision support and demonstrate how workflows can be modeled using standard modeling languages such as XPDL and executed with the integration of EHR systems. Their approach allows for dynamic adjustments to clinical workflows without the need to interrupt the system operation, a characteristic that makes the system adaptable to changes in patient care scenarios. While the authors acknowledge that some hospitals may not have EHR systems and propose alternative data sources, their primary focus remains on EHR-integrated environments without fully generalizing the architecture for all potential healthcare system variations. Similarly, Dadam et al. [38] focus on process-oriented information systems (WfMS) for healthcare and on reconfigurability and evolvability of clinical workflows. Their work highlights the use of ADEPT, a workflow management system, to enable dynamic adjustments and ensure workflows remain valid and functional when clinical requirements change. It also shows the importance of runtime workflow configurability and system scalability, which occur during system operation. This enables workflows to adapt over time without compromising the precision and reliability required in hospital environments.

Also, the primary focus of some papers is on the hospital system's components and architecture [37, 5]. This is very insightful for this study because it gives an idea of how different components/sub-systems of the hospital information systems cooperate in providing patient care, management, and other services. A very interesting part of this category's literature review revealed that the hospital information system components are far from standardized, as mentioned in the work of Huser et al. [32] as well. This shows the urgent need for this study that components like EHR, a central component in other proposed architectures [32], are not systems that should be thought to exist in every hospital. Thus, the current study's system should be designed so that if such an EHR or any other system does not exist in a specific hospital, it should still be able to function as it is supposed to function. In a few words, the system design needs to be generalized and not assume a specific hospital system exists. These studies offer critical insights into the proposed solutions for designing systems that support workflow configurability. However, there is an obvious drawback to the proposed systems' adaptability to different hospital environments and their ability to accommodate changing environmental requirements. Also, the workflow engine is identified as a widely used component in the above proposals. It is responsible for interpreting configurable workflow definitions and acting as an orchestrator for executing them. Such a component will be used in this study as well.

### 2.2.4 Clinical Workflows Modeling

The last category in this taxonomy includes studies that identify clinical workflows often lack formal representations (first column in Table 2.3), which poses significant barriers to research in this field. These studies focus primarily on the standardization of institutional workflow modeling. In this context, Carvalho et al. [39], Quaglini et al. [33], and Dadam et al. [38] propose the use of Petri Nets and UML to formalize and standardize clinical workflow representations. Additionally, Peleg and Haug [40] and Peleg [41] review various methodologies for creating computer-interpretable workflows and focus on using different modeling languages and tools to standardize clinical workflows. Lastly, Huser et al. [32] propose using XPDL (XML Process Definition Language) to model clinical workflows, enabling dynamic adjustments based on clinical needs. However, this approach requires specific architectural support, which will be considered by the current study.

In summary, despite differences in the modeling methods (e.g., Petri Nets, UML, XPDL), all the reviewed studies share a common approach, which is implementing workflow modeling through configuration mechanisms rather than hardcoded logic. This is a key insight for the current study, as it enables decoupling workflow modeling from system code, allowing workflows to be reconfigured without modifying the underlying system software. These workflow configurations can be interpreted by workflow engine as described in the previous section. Additionally, it has been shown that Petri Nets is a valuable tool for representing and explaining the workflow model and its behavior of a workflow, which will also be used in this study.

### 2.2.5 Software Product Lines

In addition to all the previous fields studied, a related field that might introduce new insights into the current study is the field of software product lines (SPLs), which is used to create configurable systems. Multiple studies focus on creating configurable systems using this approach, some of which are mentioned below. To begin with, Software product line engineering (SPLE) provides methods and tools to systematically reuse software assets and establish software product lines or portfolios of software products (a.k.a., variants) in an application domain as Lindohf et al. cites Lindohf2021 explains. Laguna et al. [50] define software product lines as a family of related software systems/products sharing a set of core assets, and they are constructed as explained next. A core system is defined with a list of mandatory functionalities that all product variants will have. These features are essential and present in every product variant. Based on this, a feature model (or other variability models) is created that includes mandatory features as well as optional and alternative ones. This model allows for specifying all the possible configurations of features that a variant can have. Once the requirements are known, the optional and alternative features from the feature model are selected to create a specific variant. These features could be added or removed in the configuration before the system's build process (compilation time). Usually, an automation tool is used to build the system with the selected features configuration and derive the product variant. The product is then automatically assembled based on the configuration. Lastly, dynamic software product lines (DSPL) can be used in cases of runtime variability needed so the system can be configured dynamically at runtime (e.g., feature toggling) to adapt to different needs.

The reviewed literature on SPLs can be categorized into four key areas: architecture modularity, legacy system evolution, process maturity, and runtime adaptability. Gomes et al. [51] focus on modularity and reuse in healthcare systems through SPliCE, an MDE method that separates clinical and architectural models to support adaptability and scalability. Laguna and Crespo [50] address evolution from legacy systems by emphasizing feature models and refactoring to improve modularity and runtime configurability. Lindohf et al. [52] assess the maturity of SPLE using the Family Evaluation Framework, highlighting organizational and architectural concerns that impact scalability and efficiency. Marques et al. (2019) and Quinton et al. ([53] & [54]) explore SPL and DSPL evolution, respectively, stressing the role of feature models, co-evolution of system components and variability models and the importance of maintaining consistency during runtime adaptations.

To summarize, the topic of SPL generally supports compile-time variability, where product variants are created during the design and build phase by selecting features and components at compile time ([54], [55], [56]). DSPL extends SPL principles by enabling runtime variability, where the system can adapt and reconfigure dynamically at runtime based on context, environmental changes, or user input. Runtime Customization in DSPL allows systems to evolve continuously after deployment, enabling automatic adjustments and new configurations without needing manual code changes. Both SPL and DSPL systems may undergo post-deployment evolution, with DSPL specifically focusing on runtime adaptation to ensure system consistency across varying states and requirements.

Although software product lines are a powerful approach for managing variability in systems and avoiding manual code customization, they are not well aligned with the goals of this specific project. As will be explained in more detail in the following chapters (particularly in chapter 3), the primary requirement of this project is to design a software system (CWS) capable of reconfiguring workflow definitions without requiring code-level changes. The goal is to adapt workflows to the specific requirements of different hospitals without creating new releases of the systems for each case. In other words, the software system release will remain the same across all hospitals, but it will support the dynamic reconfiguration and execution of workflows based on varying requirements. Software product lines in contrast generate a different variants of the system for each hospital which must be avoided.

## 2.3 Literature Review Summary

While the literature review was quite long, in Table 2.4, it is depicted as a summary of techniques and information related to the current study and which of them will be used and how.

| Viewpoint | Methods/ Ideas Used in Literature (with References) | Used in Current Study |
|---|---|---|
| **System Requirements** | PESTEL analysis [14, 13], scenario building /persona-based requirements [14, 23, 23], DSM [12] | PESTEL and scenario-based analysis are used to generate system requirements |
| **System Qualities Related to Evolvability** | Qualities taxonomy harmonization [18], changeability frameworks [20, 11] | Main system qualities used for this study is configurability, adaptability, flexibility and integrability. However other related qualities to these, like modularity, scalability and extensibility will be used for the system design and evaluation. |
| **System Architecture** | Design for Changeability (DfC) [19], dynamic reconfiguration via JADDA [30], Modular architectures [19, 14], reverse architecting [31, 10], A3 Architecture Overview [9] | The core principle of Design for Changeability (DfC) will be applied to the system design. Specifically, the appropriate system qualities will be selected based on the problem context and used as guiding principles during the architectural design phase to ensure the system's evolvability. |
| **System Architecture for CWS** | Workflow engines [38, 32], decentralized Hospital Information Systems [37, 5], EHR-integrated vs flexible design [32] | Workflow engine idea is used to enable configurable workflows during system operation. Also, architecture generalization is applied to support varying CWS environment. |
| **Workflow Variation Causes** | Medical expertise [33, 34], organizational and system context [35], decision-making and regulations [34, 35] and others | Key motivation of the study; Architecture should support workflow configurability without code level changes. |
| **Workflow Modeling** | Petri Nets & UML[38], XPDL/XML [32], computer-interpretable models [41, 40] | Ideas of use Petri Nets and configurable workflows will be used. The first for describing workflow behavior and the second to enable configurable workflows that do not need the system software to be changed or have downtime. |
| **Evolvability Assessment** | Scenario-based evaluation [26], expert evaluation [17], Delta DSM [12][22, 19], Epoch-Era Analysis [16] | Scenario-based evaluation will be used as requirements will be expressed in form of scenarios. |
| **Case Studies** | MRI systems [10, 9], reverse architecting in Philips systems [31], microservice evolvability [27, 26], SASE framework application [14] | A case study approach was followed to provide a structured method for deriving requirements for the evolvable CWS, as well as guiding its design, proof-of-concept implementation, and evaluation. |

Table 2.4: Summary of literature viewpoints and their relevance to the current study

## 2.4 Research Gaps

In the taxonomies presented in this chapter, multiple research gaps have been identified across both system evolvability and CWS domains related to the goal of the current study. From the system evolvability perspective, there is a lack of studies that explicitly define evolvability in the context of the problem addressed in this study. Moreover, few stepwise approaches exist for integrating evolvability into the design and assessment of CWS. Specifically, mainly configurability but adaptability as well remain insufficiently explored as qualities contributing to CWS evolvability, and there are no case studies combining these elements comprehensively. From the CWS-focused literature, no existing solutions support workflow configurability and environmental variation adaptability in the same system design. Many proposed systems make the critical and unrealistic assumption of standardized hospital environments. Furthermore, detailed architectural designs and comprehensive evaluations demonstrating the practical functionality of these systems are often missing. In conclusion, there is a significant gap in case studies that investigate the design of CWS from an evolvability perspective, spanning requirements elicitation, architectural design, system implementation, and evolvability-focused assessment.

This study aims to address these gaps by demonstrating how to enable both workflow configurability and environmental adaptability. Unlike existing approaches, the proposed system will support workflow configurability and system adaptation to hospital-specific settings and infrastructure without requiring code-level changes. This eliminates the need to maintain a separate CWS release per hospital and reduces the burden of extensive testing and verification for each deployment.

## 2.5 Research Questions & Methodology

To cover the mentioned identified research gap a set of research questions with sub-questions have been formed to be answered through this study. Below are the research questions, the motivation and a short methodology that has been followed for each of them.

**RQ1: What are the key characteristics and evolving requirements of clinical workflows & CWS that should be supported by the proposed system design?**

1. What are the current requirements from the system stakeholders that clinical workflows demand from the system to enable workflow configurability and system adaptability to external system variability?

2. What are the predictable future requirements of CWS that the system design/architecture should accommodate?

This question and its subquestions aim to reveal the stakeholders' requirements for the system that will be designed. To be more precise, the requirements will focus on what is required concerning the configurable workflow functionality and how the system should interact with other systems in different hospital settings. By answering this research question, this case study will construct a solid set of requirements for the system. Also, an analysis of the future requirements will reveal the possible factors that might change the system requirements and how these factors will change them. This will reveal some possible requirements variations that the proposed design should be capable of handling.

To identify the workflow requirements, the analysis of clinical workflows and CWSs is continued through a literature review and knowledge sharing from Philips domain experts. For the clinical workflow, a set of actions and configuration points that will enable the customization of the workflows will be identified so that the system can accommodate the wanted changes using configuration. After identifying the clinical workflow requirements, the method with which the clinical workflows can be expressed will be chosen. Considering the hospital's environmental variability, the requirements for system adaptability will be analyzed. Lastly, a PESTEL analysis will be conducted to identify possible future variations of the requirements.

**RQ2: How can the CWS architecture be designed to support workflow configurability and system adaptability without requiring code-level modification and satisfy the identified current and future system requirements?**

1. What are the required components for such an architectural design?

2. How should workflows be defined to allow configurability?

3. What are the trade-offs that when building such a configurable and adaptable system should be considered?

RQ2 will focus on designing the architecture that enables configurable workflows. This set of research questions aims to lead to a system architecture proposal for the identified requirements in the previous research question. More specifically, the necessary views of the architectural design will be constructed, describing the proposed system and a set of diagrams that describe how the

proposed design behaves and meets the identified requirements. This involves selecting appropriate components for designing the required system. Also, there is a focus on how the clinical workflows should be defined (configurable clinical workflow description) in the system to enable configurability. Finally, answering the third subquestion identifies all the aspects that need to be considered when configurability and adaptability are added to the system.

**RQ3: What are appropriate criteria/metrics/questions for assessing evolvability of CWS?**

1. How can these criteria be applied to evaluate the proposed architecture's ability to adapt to future workflow/system changes?
2. What change scenarios could be used to test the system's configurability and adaptability to the different hospital environments?
3. What future workflow/system requirements are unlikely to be handled by the proposed system, and what limitations do these requirements pose?

RQ3 is concerned with evaluating the system's evolvability, primarily through the prism of configurability and adaptability, as well as other selected system qualities. To do so, evaluation criteria will be developed and applied to assess how effectively the system accommodates evolving workflows and requirements. Also, for this research question, a simulation of the proposed system will be developed and, based on identified scenarios of workflow changes, will be tested and demonstrated. Finally, an analysis will be conducted on the proposed system design boundaries and will be reported.

The current study adopts a Design Science Research (DSR) methodology, which is appropriate when the goal is to create and evaluate an artifact that addresses a real-world problem. In this case, the artifact is a CWS architecture that supports evolvability. Within this research approach, the design methodology used to guide system development is depicted in Figure 2.3. Firstly, system requirements will be collected, and based on these, scenarios that the system must fulfill will be generated. Then, appropriate system qualities, following the key ones (configurability and adaptability), will be selected to be incorporated in the design. During system design, all the requirements and system qualities that the system should fulfill will be considered. Based on the generated system design, a proof of concept will be implemented to demonstrate the design capabilities. Finally, the system will be evaluated through the selected qualities and generated scenarios that the system should support.

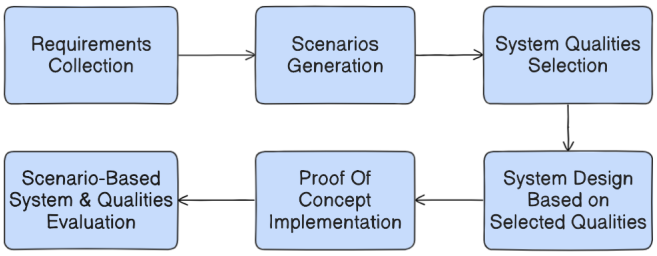



Figure 2.3: Design Methodology

## 2.6 Expected Results and Deliverables

This research will result in a comprehensive investigation of requirements, system design, proof-of-concept implementation, and evaluation of a CWS that enables workflow configurability and adaptability across diverse clinical settings without requiring code-level changes. The study will

provide a detailed analysis of workflow and system requirements, which will act as the foundation for the system architectural design proposal. A workflow definition method will be applied, demonstrating how clinical workflows can be modeled in a way that supports configurability without modifying the system's underlying software. Based on the identified requirements and selected system qualities, a system architecture will be designed and verified through analytical reasoning and initial evaluation. To validate the proposed approach, a working prototype will be implemented, demonstrating the identified requirements. A series of predefined change scenarios will be used to test this prototype in order to establish how well the system can handle changing requirements. Finally, a set of criteria will be selected to provide a valuable framework for analyzing the CWS's evolvability.

# Chapter 3

# Requirements Analysis

## 3.1 Clinical Workflow Analysis

To understand how clinical workflows can be configured and adapted in healthcare systems, it is first important to define clinical workflows: a clear and shared definition ensures that the proposed system design can correctly address the requirements for configurability, standardization, and adaptability in care delivery. The following section provides a high-level definition of clinical workflows, followed by a detailed explanation. A more technical definition, utilizing Petri Nets, is also provided. Examples demonstrate how clinical processes can be modeled according to this definition. The section will then continue by identifying the system requirements.

### 3.1.1 Clinical Workflow Definition

There are multiple clinical workflow definitions in previous works, but as the literature points out, for example, in an extended review conducted by Ozkaynak et al.[2], many ambiguities remain in the proposed definitions. One such representative definition is given by Zheng et al.[57]: "A set of tasks, grouped chronologically into processes, and the set of people or resources needed for those tasks that are necessary to accomplish a given goal". Another, by Davis et al. [58], defines clinical workflows as "a process involving a series of tasks performed by various people within and between work environments to deliver care". These definitions provide a high-level understanding of what a clinical workflow is. However, they do not offer a holistic explanation of workflow components, its functioning, and execution.

Building on these high-level definitions, the current study proposes a more detailed and operational explanation. A clinical workflow is a procedure consisting of a structured, predefined series of tasks (executed in sequence or parallel), decision points, transitions, and events whose execution achieves a defined outcome, standardizing care delivery. The execution of each task, or the procedure as a whole, must comply with any time constraints and may involve healthcare staff, tools, and systems from one or more organizations working together to provide efficient patient care. A workflow is executed when triggered and may also trigger other workflows. Overall, clinical workflows ensure that care processes are carried out in a coordinated manner, promoting standardization, efficiency, compliance, and patient safety. Tasks represent executable actions, transitions, and conditions control their execution flow, and events can be used to specify triggers that initiate, continue, or terminate the workflow.

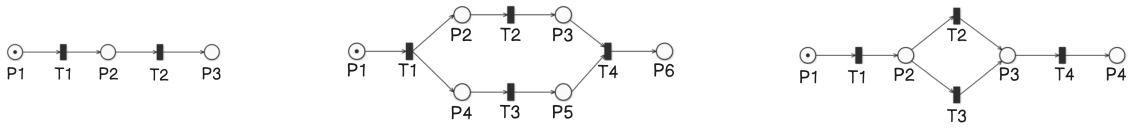
To formalize and model workflow behavior, this study adopts Petri nets as a technical representation. Petri nets provide a formal and executable foundation for describing how workflows are structured, triggered, and executed. Below is the formal definition of a Petri net, based on Wang et al. [59]. Following Wang et al. [59], a Petri net is formally defined as a 5-tuple $N = (P, T, I, O, M_0)$, where:

1. $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of places.
2. $T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of transitions, such that $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$.
   *To enable at least one transition, it is required that* $|P| \geq 2$.
3. $I : T \times P \to \mathbb{N}$ is an *input function* that defines directed arcs from places to transitions, where $\mathbb{N}$ is the set of non-negative integers.
4. $O : T \times P \to \mathbb{N}$ is an *output function* that defines directed arcs from transitions to places.
5. $M_0 : P \to \mathbb{N}$ is the *initial marking*, assigning a number of tokens to each place. Tokens represent information, conditions, or resources in the system. The system's state is determined by the distribution of tokens across places, and transitions operate by consuming tokens from input places and producing tokens in output places.

A *marking* in a Petri net refers to the specific allocation of tokens across places, which defines the current state of the system. If there is an arc from a place $p_j$ to a transition $t_i$, then $p_j$ is an *input place* of $t_i$, formally expressed as $I(t_i, p_j) = 1$. Conversely, if an arc leads from $t_i$ to $p_j$, then $p_j$ is an *output place* of $t_i$, denoted $O(t_i, p_j) = 1$. A place $p_i$ is a passive component that represents a condition, state, or resource in the system. A transition $t_i$ is an active component that models an event or action that changes the system's state. An arrow (directed arc) connects places to transitions or transitions to places, defining the direction of flow and dependencies between conditions and actions. Finally, a token represents an object, piece of information, or condition. The distribution of tokens defines the system's state, referred to as the marking, and transitions change this state by moving tokens through the net.

To illustrate the formal definition of a Petri net, consider the simple example shown in Figure 3.1 (a). It consists of three places $P = \{p_1, p_2, p_3\}$ and two transitions $T = \{t_1, t_2\}$. The input function $I$ and output function $O$ define the arcs: $I(t_1, p_1) = 1$, $O(t_1, p_2) = 1$, $I(t_2, p_2) = 1$, and $O(t_2, p_3) = 1$. The value assigned by the input/output functions (such as 1 in this case) is called the arc weight, indicating how many tokens are required or produced. The initial marking $M_0$ assigns one token to $p_1$ and zero to all other places. A transition is said to be *enabled* when all its input places contain at least the number of tokens required by the arc weights. In this case, $t_1$ is initially enabled because $p_1$ has one token. When $t_1$ fires, it consumes one token from $p_1$ and produces one in $p_2$, enabling $t_2$. Firing $t_2$ then moves the token to $p_3$, completing the process. This minimal example demonstrates the fundamental mechanics of token flow, enabling, and transition firing in a Petri net.

Other functionalities can be modeled by Petri nets, such as parallelism, conditional execution, and loops. In Figure 3.1 (b), parallelism is demonstrated. After the initial transition fires, one token is produced for each parallel branch, enabling both transitions $T_2$ and $T_3$. Each branch proceeds independently, and the process continues only when both branches have completed, as tokens in both $P_3$ and $P_5$ are required to enable $T_4$. Also, conditional execution is illustrated in Figure 3.1 (c). When transition $T_1$ fires, a token is placed in $P_2$, enabling both transitions $T_2$ and $T_3$. However, only one of these transitions can fire (consume the only available token), thus modeling a decision point. The token proceeds through either the upper or lower path, and the process continues enabling $T_4$.




(a) Sequencial (b) Parallel (c) Conditional

Figure 3.1: Petri Net Examples

To apply this modeling approach to the current study context, part of the sepsis workflow guideline algorithm shown in Figure 1.2 is modeled as a Petri net and shown in Figure 3.2. Initially, the transition *Start Vitals Measurement* is enabled, and after firing, four tokens are distributed across the four parallel branches and enable the transitions *Measure Heart Rate*, *Measure Temperature*, *Measure Respiration Rate* and *Evaluate Mental State*. These transitions can fire independently, simulating the execution of the respective action. Once all the transitions have fired, indicating that the actions have been executed, the transition *Evaluate Vital Signs* becomes enabled. At this point, the workflow models that, based on the outcome of the measurements, conditionally proceed to either initiate an infection check or continue with the standard treatment and finally, the workflow is complete. During the workflow execution, the distribution of the tokens in the Petri net shows its state, giving information on which transitions are enabled and indicating which actions can be executed. It should be noted that the final conditional part of Figure 3.2 does not strictly align with the logic depicted in the original sepsis workflow guideline. In the initial proposed workflow, other actions need to be executed after the *Checking For Infection* transition. This part has been simplified for illustrative purposes to demonstrate how Petri nets can be used to model and explain clinical workflow execution.

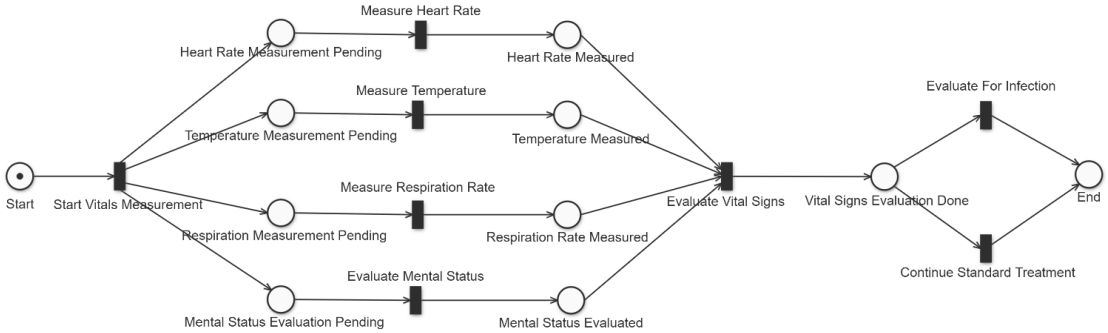



Figure 3.2: Part of Sepsis Workflow expressed in Petri Net

Now that the modeling and execution of clinical workflows using Petri nets have been thoroughly explained, a simpler and more user-friendly notation will be adopted for the remainder of this study. As outlined in the preceding explanation, the clinical workflow, when expressed in simplified notation, comprises tasks, transitions, conditions, and events. Thus, a mapping is provided between these simplified notation components and Petri net components. In this mapping, tasks in the simplified notation are represented as transitions in the Petri net, corresponding to discrete, executable actions. Conditions are implemented as places that hold a token and can enable the corresponding following task (Petri net transition) to fire. Events, such as start or end events, are modeled contextually as places in the Petri net, where a start event corresponds to a place with an initial token, and an end event to a place that holds a token upon workflow completion. In the simplified notation, directional flows(transitions) between components represent the control flow of the process. In the Petri net mapping, these flows are not implemented as standalone elements but are instead realized through a combination of places and transitions that model sequencing, parallelism, and synchronization. Some places and transitions in the Petri net model, such as a place labeled Measurement Pending or a transition named Start Vital Measurement, serve as structural components. These are added not to represent functional components of the simplified workflow, but to support sequencing, synchronization, or parallel execution. While these structural elements do not correspond to any specific workflow component in the simplified notation, they are essential for preserving the intended execution logic in the Petri net.

Thus, in this notation, tasks are units of work that consume a token when executed and produce a token for each of their outgoing transitions. A task is enabled when at least one of its input components can provide a token. Transitions in the simplified notation represent the directional flow between components; they are not active components (unlike Petri net transitions) but passive

connectors that transfer tokens between components. Conditions are decision points that can hold a token and evaluate by which next component it should be consumed. In this simplified notation, conditions require all previous components to be able to provide a token before being enabled, functioning as synchronization points. Events are specialized constructs that mark the start or end of the workflow: a start event produces the first token to initiate execution, while an end event consumes the final token to mark completion. Although this simplified notation abstracts away the structural details of Petri nets (e.g., intermediate places and auxiliary transitions), it retains the core execution logic. As such, it offers a more intuitive view while remaining compatible with Petri net semantics.

Using the simplified notation explained above, part of the sepsis workflow guideline algorithm, shown in Figure 1.2, has been modeled and shown in Figure 3.3. At the beginning of the workflow, the start event can activate the workflow and produce four tokens, one for each outgoing transition. Thus, the next four tasks (modeled by rounded corners rectangles) are enabled and can be executed in parallel (*Measure Heart Rate (H)*, *Measure Temperature (T)*, *Measure Respiration Rate (R)*, and *Evaluate Mental status(M)*). Their arrangement allows their execution to be in parallel, meaning that their execution is independent. Similarly in the Petri net, once all four tasks are completed, each one produces a token. These tokens are merged in a place representing the condition (modeled by a diamond shape), enabling the workflow to proceed through a subsequent transition. As shown in Figure 1.2, for each measurement, there are specific limits, e.g., the temperature should be between 36 and 38 degrees Celsius. So each task that measures one vital sign produces a related value ($H,T,R,M$) which is passed to the condition component to be evaluated. For simplicity, in Figure 3.3, the case where each one of them is out of limits has been marked, as $c(H)$, $c(T)$, $c(R)$, or $c(M)$. Thus, the condition $(c(H)+c(T)+c(R)+c(M)) \geq 2$ is evaluated using the provided values from the previous tasks, and based on its result, a token is produced for the corresponding outgoing transition, either towards the *Evaluate for Infection* task if the condition is met, or towards the *Continue Standard Treatment* task otherwise. This illustrates how the result of the condition evaluation is explicitly used to guide the flow, ensuring that the decision logic is clearly represented in the workflow. In the same way, the rest of the workflow is executed.

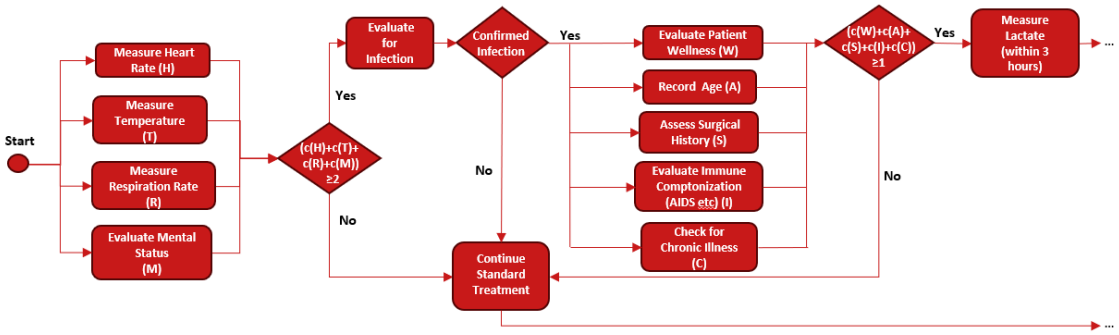



Figure 3.3: Part of Sepsis Workflow from Adult ED Sepsis Guidelines Algorithm 2022

It is also helpful to note that the tasks in the workflow can be either automated tasks executed by a system or tool or manual tasks executed by a medical staff member. For example, the *Evaluate Temperature* task may need to be executed either automatically by a patient monitor or manually by a nurse, which serves as a good example of why configurability in workflows is necessary. This would not change the workflow's structure in the above diagram, but in a more detailed model, such configurability would be reflected as a task parameter. Regarding time-related execution constraints, as shown in the last visible task in Figure 3.3, *Measure Lactate within 3 hours*, it is specified that it must be completed within a three-hour timeframe. Thus, it is clear that time limits should be met. This diagram does not show how this is secured, but this will be clarified in chapter 5, where an actual workflow implementation is shown. Finally, for this workflow to be triggered, there are multiple possibilities. In some hospitals, the workflow needs to be triggered

immediately when a patient is hospitalized, or in others, it can be triggered dynamically by tools or clinicians when certain conditions are met.

### 3.1.2 Workflow Variability

In this section, all workflow variation points across hospitals, and the changes that the CWS should be able to handle without requiring code-level changes (i.e., being configurable) are identified. Following this, a PESTEL analysis is conducted to assess potential future requirements and their impact on the identified system requirements.

So far, and especially in section 2.2, it has been clearly explained why workflows need to be configurable. However, it is necessary to identify how workflows need to be reconfigured. To begin with, there are four identifiable workflow components: tasks, transitions, conditions, and events. Tasks represent the individual actions that must be performed within the workflow. Transitions define the flow between the components, dictating how the process moves from one component to another. Conditions are logical rules that determine the workflow's path based on specific metrics used within the defined conditions (e.g., vital sign thresholds). Lastly, events are components triggered by external or internal stimuli that can initiate, continue, or terminate a workflow. It is obvious that since workflows differ across hospitals, different arrangements of these components might be asked. Therefore, the system should be able to add, remove, or rearrange these elements within the workflows.

In addition to these core components of workflows, other internal component points are also of great importance. These internal properties are not part of the core Petri net mapping but represent configuration details at the internal level of each component. Specifically, each task can have an input, output, pre-and post-conditions, the task's executable actions, associated actors, and execution time constraints. A task can receive an input that, under a precondition, can execute an executable action using an associated actor or system within a time constraint and produce a specific output intended to satisfy the postcondition, marking it as completed. Some of these points should be configurable, while others should not. To simplify the analysis, we assume that each task performs only one executable action. For example, a task will only measure the patient's temperature, not all vital signs.

Considering the task component and the stated assumption, the input, precondition, post-condition, and output should be fixed and not configurable at a certain level because they are tightly coupled with the task's selected executable action. For example, to measure a patient's temperature (task), the patient ID is needed as input. The precondition can be that the patient is present on the bed. The measuring can then take place (executable action), producing the output as the patient's temperature in Celsius. A boolean expression checking if the value is reasonable for a human being can be the postcondition to avoid measurement from a thermometer that malfunctions. Even though the task assumes some core preconditions to be satisfied (e.g., presence of the patient) in order to be executed, a hospital might need to have and manage some additional customized pre-postconditions because those define under which circumstances a task should start or complete, and this might vary from hospital to hospital. Similarly, the actors assigned to a task may need to be adjusted based on organizational roles or systems availability. Timing constraints, such as execution deadlines or acceptable delays, must be configurable. At the same time, timing constraints ensure that clinical workflows adhere to necessary medical protocols and safety standards, which may differ across hospitals.

Regarding the actor variation point of a task, further clarification is required. As identified during the analysis and discussed in the literature review (section 2.2), hospitals operate in diverse environments with different organizational structures and system infrastructures. As a result, the same task may be executed by different types of actors across hospitals. This highlights the need to model a more generic type of environmental variation within workflows. If a task must be performed manually by hospital staff, the actor can be specified based on the hospital's internal roles and responsibilities. Conversely, if the task is executed automatically by another system, the actor should indicate which system is responsible. Hospitals may vary significantly in terms of available infrastructure. For instance, some may use Electronic Medical Record (EMR) or

Electronic Health Record (EHR) systems, while others may rely on manual data entry in the absence of such systems. These differences must be captured when modeling both manual and automated tasks to ensure accurate representation and integration with the CWS.

In addition to individual tasks, the notion of sub-workflows is used. Conceptually, a sub-workflow can be considered a specialized task that, instead of executing a single executable action, triggers the execution of an entire embedded workflow. Formally, a sub-workflow is itself a workflow composed of the same component types (tasks, transitions, conditions, and events) and can be invoked as a modular unit within a parent workflow. Configuring such a sub-workflow should also be possible, as they are treated as modular units within a larger workflow. Sub-workflows offer an additional layer of flexibility in workflow modeling and simplifies the modeling of a bigger workflow. Sub-workflows may need to be added, removed or rearranged to adapt complex clinical procedures without affecting the model of the main workflow. However, as sub-workflows represent a workflow, they are not considered a unique configuration point, and their configuration will be enabled by the configurability of all the other components.

Similarly to tasks, conditions serve as decision points within the workflow, and they must also be configurable by allowing the condition logic, metrics, and thresholds to be adjusted to reflect changes in clinical guidelines or patient management protocols. It is also essential that the workflow component's arrangement can be adapted to meet the individual hospital needs. Specifically, tasks and conditions might need to be added, removed, or be rearranged to adjust the workflow according to evolving clinical requirements or hospital-specific practices. For that to be possible, transitions need to be configurable. Particularly they must support the ability to be added, removed, or modified by changing their source and destination points to introduce new pathways between tasks, conditions or events. Events must also be able to be added and removed to ensure the workflow can react appropriately to different external or internal stimuli. This structural adaptability complements the fine-grained internal configurability of individual workflow components and ensures that small and large workflow changes can be efficiently managed. To summarize the variation points discussed in the previous paragraphs, Table 3.1 presents all the workflow configuration points that the CWS should support.

| Workflow Component | Variation Point | Explanation |
|---|---|---|
| **Task** | Precondition/Postcondition | Define the condition under which the task's executable action can start or complete. |
| | Execution Time Constraint | Define execution deadlines, maximum durations, or acceptable delays for task completion. |
| | Actor | Assign different actors (e.g., nurse, doctor, automated system) responsible for executing the task's executable action. |
| | Entire Task | Add, remove, reorder tasks to adjust workflows. |
| **Transition** | Entire transition | Add, remove a transition or modify the source and destination points of a transition to adapt workflow's components arrangement. |
| **Condition** | Condition Expression | Adjust decision rules and threshold values (e.g., vital sign metrics). |
| | Entire Condition | Add, Remove or reorder a condition. |
| **Event** | Entire Event | Add, Remove. |

Table 3.1: Workflow Components and their Configuration Points

While Table 3.1 outlines the configurable points for each workflow component, it is important to pinpoint that these configurations, once they are reconfigured, must be validated to ensure that the generated workflow is still valid. This validation must also consider whether the changes are compliant with medical guidelines, authorized by clinical roles, and free from unintended dependencies (e.g., tasks that must not be combined or reordered due to safety constraints). Considering workflow validation, two main perspectives must be considered: syntactic validity and semantic correctness. Syntactic validity ensures that the workflow remains structurally sound and follows

the workflow model's formal rules after any modification. On the other hand, semantic correctness ensures that the resulting workflow continues to represent a clinically meaningful and acceptable process. This addresses the concern of preserving workflow identity, for instance, ensuring that a reconfigured workflow still qualifies as a *sepsis workflow*. Semantic correctness would define and enforce such boundaries, ensuring that the intent and clinical purpose of the workflow remain valid. For example, reassigning a medical task to a non-authorized actor (e.g., assigning a diagnostic task to an automated tool instead of a doctor) could violate clinical constraints, and such changes must be restricted. Therefore, while configurability is essential, it must be guided and constrained by both syntax and domain-specific semantics. While both of these aspects will be taken into account for the system design, in the current study, only syntactic validity will be actually implemented, while semantic validity can constitute a study focus on its own.

### 3.1.3 PESTEL Analysis

One of the most important aspects of designing an evolvable system is predicting, when possible, future sources of change and requirements and their impact on the system being designed. By analyzing these changes during the design phase, it becomes possible to anticipate what the system should be able to handle in the future. Even if these requirements are not implemented from the beginning, the system design should allow their integration without design level changes later, making the system design evolvable. Thus, in addition to the currently identified requirements, namely, the variation points of workflow components summarized in Table 3.1 which must be configurable in order to support changes without code-level modifications, the next step is to identify potential future changes and their impact on the system that will be proposed. For this purpose, PESTEL analysis is used. The PESTEL framework is a structured approach for assessing the external factors that, for this case, influence healthcare workflow systems. By examining the six key areas, Political, Economic, Social, Technological, Environmental, and Legal, PESTEL analysis provides valuable insights, helping identify potential influences that could impact the system design and functionality.

The analysis is based on known changes likely to occur, drawn from the literature and stakeholders, as well as articles that focus on exploring the trends in this sector over the next few years, such as [60, 61]. The sources were first identified through a review of relevant literature, market analysis articles, and input from domain experts. They were then classified using the PESTEL framework according to the nature of the influencing factor. Each row includes the corresponding PESTEL category, a description of the change source, its expected impact on the system, and its classification into one or both of two impact categories: (1) workflow configurability and (2) system integration and deployment. These categories are further analyzed in the following paragraphs. So Table 3.2 presents the PESTEL analysis, which shows the sources of change that can occur and how these can influence the CWS. In the next paragraphs, these sources of change are analyzed. For this project's scope, eleven main sources of change were identified, related to the specific problem context this study focuses on. However, a more detailed analysis should be conducted to provide a broader overview of other problem contexts in the healthcare domain.

To begin with, one of the key political factors that affects the CWS is the *accessibility of healthcare guidelines* within different healthcare networks (*1*). Changes in healthcare policies and the availability of guidelines (for example, the potential withdrawal of a country from international health organizations such as the WHO) can significantly impact the design of the workflows. This is a critical aspect that the system must handle since workflow variations are prevalent across different institutions because of this influence. These changes influence the execution flow, the availability of decision rules, and in total the variation points in the workflows. As healthcare guidelines evolve or become more/less accessible in different regions, the system must be flexible enough to adapt to these changes.

*Availability of resources* (*2*) is one of the main economic factors influencing the CWS, from both human and technological perspective. Resource availability changes, like budget constraints, staff shortages, or from the other side the introduction of new technologies/systems, can significantly impact how workflows are designed and adapted. For example, hospitals with limited staff or

| NR. | PESTEL Category | Source of Change | Impact on the System | Category |
|---|---|---|---|---|
| 1 | Political | **Guideline Accessibility:** Influence of guideline access within healthcare networks [35]. *(e.g. USA dropped from WHO)* | Might lead to workflow changes because updated rules are not available, and thus, variability exists across institutions. | 1 |
| 2 | Economic | **Resource Availability:** Impact of limited/excessive resources (human or technological) [33, 35]. | Impacts environment systems availability, timing, and staffing dependencies in workflows. | 1,2 |
| 3 | Social | **Patient Centric Care:** Role of collaborative decision-making between clinicians, patients, and families [33]. | Requires adaptation of the workflow model and support for custom data inputs and decision-making. | 1 |
| 4 | | **Clinical Leadership and Expertise:** Influence of clinicians' experiences and preferences on workflow design [34, 35]. | Introduces local deviations or modifications to workflow guidelines. | 1 |
| 5 | Technological | **Infrastructure and Support Systems:** New hospitals with advanced electronic systems, such as EHR and CDSS, integrate digital solutions and AI, while older hospitals still rely on paper-based systems [35]. | Determines which systems need to be integrated into the system and the required interfaces. | 2 |
| 6 | | **Telemedicine Advancements:** With this approach patients can be monitored remotely using wearable sensors and distant consultations. | The system should be able to be deployed in different machines in order to support both centralized and decentralized patient care as well as integrate new technologies/systems. | 2 |
| 7 | | **Interoperability & Security Standards Change**: Interoperability standards continue to improve, and new standards may need to be adopted. | Impacts how the system communicates with and integrates data from other systems. | 2 |
| 8 | Environmental | **Local Knowledge:** Customization to reflect local expertise, knowledge, and effective practices [33]. | Leads to localized workflow variants and condition-specific adaptations. | 1 |
| 9 | | **Organizational Structures:** Influence of healthcare organization setup on workflow processes [33]. | Determines who owns processes, which actors perform tasks, and how coordination is achieved, including deployment alternatives. | 1,2 |
| 10 | Legal | **Local Practices:** Adaptation of national or international guidelines to local contexts [34]. | Enforces rule-based constraints and adaptations in workflows or system integrations. | 1,2 |
| 11 | | **Data Privacy Regulations**: Ongoing updates to privacy laws protect patient data and uphold their rights over its use, storage, and sharing. | Impacts the data storage, encryption standards, access controls, data anonymization, advanced audit trails. | 2 |

Table 3.2: PESTEL Analysis - Sources of Change in Healthcare Workflows Management Systems

systems may need to implement simplified workflows ignoring not mandatory tasks to maintain efficiency. In contrast, others with more resources may have the flexibility to use more personalized or detailed workflows using more, both human and system, resources. The system must adapt to these variations in resource availability, to ensure workflow configurability independently of these constraints.

Additionally, through out the recent years there is a noticeable shift towards *patient-centric care* (*3*) that requires the system to support personalized workflows and collaborative decision-making between clinicians and patients. This shift emphasizes the need for flexibility, configurability and adaptability, as the system must flexible and configurable enough to adapt on hospital needs for making their workflows be more or less patient or system centric by adapting the workflow models. Next to this, *clinical leadership and expertise* (*4*) leads to local deviations from guidelines workflows, as clinicians customize processes to meet specific needs.

*Technological advancements* (*5,6*) in healthcare are driving significant changes in system design and this is one of the most important aspects to capture. New hospitals with advanced electronic systems, such as Electronic Health Records (EHR) and Clinical Decision Support Systems (CDSS), favor integrating digital solutions and AI to CWS. In contrast, older hospitals may still want to rely on paper based systems or more conservative technologies. Thus, the system should be capable of integrating with various infrastructures, independently of their interfaces, to ensure integrability and adaptability. From technology perspective, it would be rational to support that while systems evolve over time, CWS should be capable of operating in different platforms (e.g. Patient Monitor, On premise servers, Cloud). Also, such an approach requires flexibility in the system deployment and functioning because such a scenario might requires centralized monitoring of multiple patients (Server/Cloud) or more edge computing approach (Patient Monitor) to have a dedicated system for each patient. Additionally, the continuous evolution of *interoperability standards* (*7*) means the system must adapt to new standards in order to interact with other systems and also meet security and data consistency needs during these interactions. Thus, as healthcare technology evolves, the system must support integration of different systems with different interfaces and ensure data security and consistency, and ultimately remain interoperable with other healthcare systems. Finally, telemedicine is becoming increasingly prevalent in patients' lives, enabling the use of new monitoring systems such as wearable sensors for remote patient monitoring [62]. Consequently, the system should remain flexible enough to integrate such technologies and adapt the clinical workflow accordingly.

While all sources of change stem from the broader context in which the CWS operates, the PESTEL framework classifies Environmental factors more narrowly. In this analysis, environmental influences include elements such as *local knowledge* (*8*) and *organizational structure* (*9*), which shape how the CWS is deployed and used across hospitals. The system must adapt to regional differences in healthcare practices and ensure workflows are tailored to specific institutional needs. For example, hospitals in rural areas may require different workflows than urban hospitals. Additionally, the system must accommodate variations in organizational structures, such as centralized or decentralized patient management, to support the distribution of tasks and responsibilities.

Legal factors are critical to the CWS as healthcare systems must comply with various regulations and guidelines. *Changes in local practices* (*10*) and national or international healthcare guidelines often require the workflows to adapt in order to remain compliant. Also, the system must handle *data privacy* and other relevant regulations (*11*) such as GDPR or HIPAA, ensuring that patient information is stored, accessed, and shared securely. As new laws or regulations emerge, the system must be configurable enough to integrate these legal requirements without compromising functionality.

Based on the analysis, two main categories of impact on the CWS can be identified. The first category consists of sources *1, 2, 3, 4, 8, 9*, and *10*, and the second category consists of sources *2, 5, 6, 7, 9, 10* and *11*. The first category is directly related to the system's requirements for supporting the identified workflow variability and so workflows configurability. These sources reflect factors such as resource availability, clinical expertise, and organizational structures, all

of which influence how workflows are designed. They emphasize the system's ability to adapt to varying external conditions across healthcare settings. In contrast, the second category focuses on system integration with external technologies, deployment options and internal functionality. As the CWS needs to interact with other systems (e.g., EHRs, CDSS), this category emphasizes the importance of ensuring that the system can adapt to new interoperability standards, integrate with emerging technological platforms an handle regulations related to data privacy, consistency and security. Numbers *2, 9* and *10* are included in both categories, because they relate to workflow variability, system integration and deployment requirements.

PESTEL analysis is typically conducted during the early stages of system development, before the system exists, to evaluate how external factors might influence the system's design and requirements. In this study, even though the system has not yet been developed, the impact of these external changes can already be analyzed based on the identified requirements (workflow configurability and environmental adaptability), as these define what the system must be capable of addressing. Since all sources in the first category are perfectly aligned with the requirement that the CWS should handle workflow's variations and ensure that it enables configurable workflows (without code-level changes), this category is thought to have minimal impact on the system as this is the main system requirement and should be met for the system design to be considered as successful. However, if someone evaluates the impact of this category the other way around, it could be argued that even a single unmet requirement would likely cause the system to fail in supporting the desired functionality. Thus this category has very high impact to the system. Either way it is evident that the system should handle such changes. Regarding the uncertainty of these change factors, all of these are highly predictable because they stem from established trends or ways of working that are unlikely to change unpredictably. For instance, it is almost certain that different hospitals will face varying resource availability and expertise, making this an unavoidable design factor in CWS design.

For the second category, the focus is on ensuring flexibility for integration with other systems. Following the same reasoning as in the first category, the impact of this category to the system is high because without designing the system to be flexible and adapt to the external variability (interoperability protocols, external system variability, deployment alternatives), the system will fail to meet integration requirements and will compromise the ability to configure and execute workflows without code-level changes. These sources of change are also certain, as technological evolution and interoperability standards are continuously improving, thus the system will still need to be adaptable to future changes on these points. The need for supporting different deployment models remains uncertain, as no concrete guidelines or widely accepted standards have been identified. However, if this requirement is not considered during the design phase, it could lead to unavoidable system redesign in the future, a scenario regarded as the worst case. Therefore, both the impact and the uncertainty associated with this requirement are relatively high. That said, as demonstrated in the following sections, examples of such scenarios indicate that the need for different deployment models is both reasonable and foreseeable, and this requirement should be incorporated into the system's design.

An interesting observation is that one might expect varying levels of impact severity and uncertainty across the two categories. However, since each category includes at least one high-impact and certain source of change, both are considered critical for integration into the system design to ensure it can evolve and meet current and foreseeable future requirements.

### 3.1.4 Final Requirements

After doing this comprehensive PESTEL analysis, it can be concluded that three main requirements have been identified for the CWS and they should be taken into account for designing the system:

1. Workflow Configuration: The system must be capable of reconfiguring workflows without code-level changes as analyzed in detail in subsection 3.1.2 to adapt to changing workflow

requirements originating from regulations, clinical expertise and other sources.

2. System Adaptability, Flexibility & Integrability: The CWS must integrate seamlessly with existing healthcare systems, such as EHRs, CDSS, patient monitors, and other technological platforms and systems. As interoperability standards and external system interfaces continue to evolve, the system must be able to adapt to these changes, ensure secure and consistent data exchange between systems, and ultimately support the integration of emerging systems and other technologies.

3. Deployment Flexibility & Adaptability: The system must support both centralized and decentralized execution of patient workflows to accommodate variations in healthcare infrastructure. In a centralized deployment, the CWS may run on a central server managing multiple patients across different locations. In contrast, decentralized deployment allows the CWS to run locally, for example, directly on a patient monitor, to support isolated, patient-specific workflows. This flexibility is essential for deployment across diverse care settings such as hospitals, clinics, ambulances, and remote care environments. Without it, the system would be limited in applicability and unable to support key use cases with varying infrastructure constraints.

These high-level requirements are further detailed and made measurable through the change scenarios in the following section, which specify testable system behaviors in response to concrete sources of change.

The ultimate goal of the proposed system, above any specific requirement discussed below, is to ensure that it can cope with both current and future requirements, so it is evolvable. A very good example of a concept that allows a system to be evolvable is the concept known as Software-Defined Systems [63]. This concept is based on the fact that in systems or products with fixed hardware, which is rarely changed during their life-cycle, the range of functionalities they can offer to users is inherently limited. Software-Defined Systems introduced the idea that software can add new functionalities, correct errors, and meet future requirements since the hardware cannot be changed. The software defines the system's functionalities, which is why the concept is referred to as Software-Defined Systems (SDS). So, this concept is a good example of how a system can be built to be evolvable and meet future requirements, overcoming the constraints hardware puts on the system. However, for each system, the definition of evolvability can vary. In general as mentioned in the literature review, System evolvability is considered as the ability of a system to accommodate changes with minimal cost while maintaining its architectural integrity. That said, in this study's proposed CWS, on top of this mentioned definition, evolvability means that no-code customizations need to be made to adapt the system's workflows to different hospital requirements, and no-code or minimal-code maintenance is needed for the system to adapt to the continuously evolving system's environment. Code maintenance, is not referring to the CWS customization on a specific hospital's needs but to add new functionalities that globally are needed. So, one CWS release will exist for all the involved hospitals.

## 3.2 Change Scenarios

Based on the analysis of current and future requirements in subsection 3.1.2 and subsection 3.1.3, a set of scenarios to illustrate what the system should support in practice are presented in this section. The scenarios focus on three different perspectives. First, workflow variability scenarios will show practically how workflows can be customized. Next, environmental variability scenarios will demonstrate how the system should adapt to changes in its operating environment. Finally, deployment scenarios will illustrate how the system can be deployed to meet different infrastructure and organizational requirements.

### 3.2.1 Workflow Configurability Scenarios

To demonstrate what the CWS should support in practice for enabling configurable workflows, this section presents a set of illustrative, simplified but representative, scenarios. The components used in the following workflow diagrams follow the simplified notation, explained earlier, and are the next:

- Start Event: Represented by a circle, indicating the entry point of the workflow.
- End Event: Represented by a bold circle, indicating the termination of the process.
- Time Event: Represented by a doubled circle, which can be activated after a specified amount of time and trigger an outgoing transition.
- Task : Represented by a rounded rectangle, describing an task to be performed (e.g., "Measure Temperature").
- Condition: Represented by a diamond shape, which evaluates conditions to determine the path of execution, similar to the diamond shape condition shown in Figure 3.3. It's also referred as exclusive gateway as only one output pathway can be followed based on the condition.
- Transition: Represented by a directed arrow, used to connect the rest of the components and indicate the execution flow.

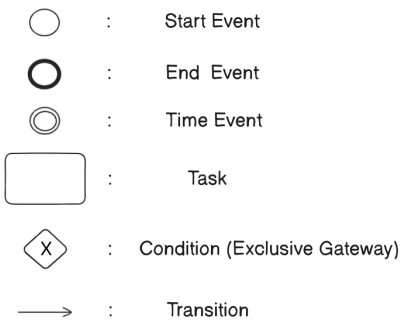



Figure 3.4: Workflow Components

Let us start by showcasing scenarios related to the task variation points, depicted in Table 3.1, using the context of the PESTEL analysis and, in general, the prior analysis is done. To begin with, a scenario caused by a political influence that affects hospitals' guideline accessibility. The decision made by the USA in 2025 to withdraw from the WHO [64] could potentially lead to limited access to clinical workflows published by this organization in USA hospitals. This change could result in differences between the updated proposed workflows that CWS may contain and the ones that hospitals have adopted since the last update received and based on how they update their workflows using local knowledge and protocols. If a hospital in the USA acquires the CWS, it will need to adapt the proposed workflows in CWS to comply with the specific practices and protocols the specific hospital follows. This scenario demonstrates the task addition, removal, and reordering variation points and is depicted in Figure 3.5. The white rectangles represent the tasks that remain in the workflow after reconfiguration, while the blue ones represent the tasks that are affected by the reconfiguration.

- **Task Addition**: In the *Addition Scenario*, the task *Evaluate Patient Mental Status* needs to be added between the existing tasks *Measure Temperature* and *Measure Systolic Blood Pressure*. The system must support inserting this new task into the workflow, without

needing of any code-level customization leading to a new release. The system should be aware of this change and execute the updated version of the workflow upon an execution trigger. This applies on all of the workflow reconfigurations cases.

- **Task Removal**: If a task, such as *Evaluate Patient Mental Status*, is deemed redundant or not aligned with the hospital's workflow needs, it can be removed from the workflow, as shown in *Removal Scenario*. Accordingly, the system should be able to let the administrative user remove the wanted task and update the workflow to proceed directly from *Measure Temperature* task to *Measure Systolic Blood Pressure*.
- **Task Reordering**: Tasks might need to be reordered (*Reorder Scenario*) based on clinical preferences. For example, the task *Evaluate Patient Mental Status* could be moved to the end of the series of the tasks to ensure, in cases that the other tasks are deemed more critical and need to be executed first.

Considering hospitals' organizational structure can similarly influence workflows, as explained above, allowing for the addition, removal, or reordering of tasks. However, an additional impact that this change can have on the system is that it requires actors or roles to be changed in tasks to specify who will execute the specific task. For example, due to changes in organizational structure, a task that a doctor previously performed may now be assigned to nurses after they receive relevant training to undertake the specific task. Having a version of a workflow used at a specific point in time, where the task *Evaluate Patient Mental Status* is assigned to doctors, the system should allow administrative staff to modify this assignment and designate new actors or roles for the task, such as nurses. In this scenario, it is demonstrated how an actor or role can be reconfigured.

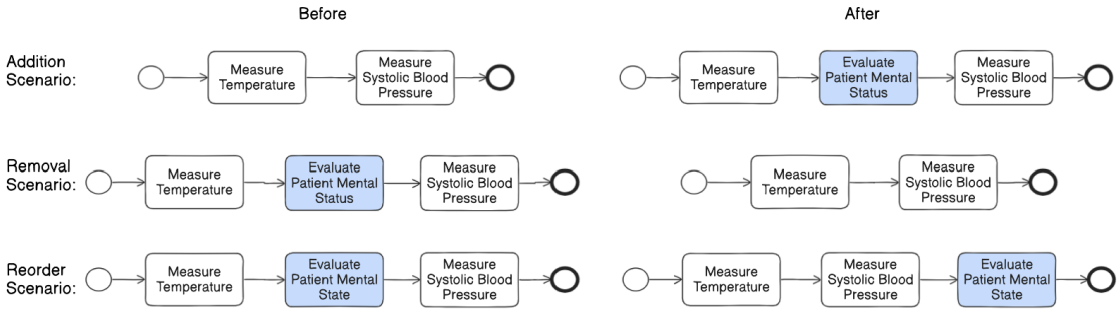



Figure 3.5: Task Reconfiguration

If we consider, hospitals' local knowledge and expertise in hospitals, several other scenarios can be generated. For example, research outcomes might be applied to test clinical workflows in teaching hospitals. In case it has been identified that some actions instead of being executed each time the workflow is being executed, by adding conditional logic, they can be executed only under a specific condition defined. The following scenarios focus on these variation points, specifically the addition, removal, or reordering of conditions and addition, removal and modification of transitions within the workflow.

- The system must support the **addition of conditions** to align with hospital practices or clinical guidelines. For example, as *Addition Scenario* shows in Figure 3.6, after the *Measure Temperature* task, a new condition can be inserted to add conditional functionality. If the temperature is normal (considered as 36.4°C-36.8°C), the workflow proceeds to *Evaluate Patient Mental Status*, otherwise, it proceeds to execute *Measure Systolic Blood Pressure*.
- The system must support the **removal of conditions**. For instance, if independently of the temperature checking task outcome the task *Measure Systolic Pressure* is deemed necessary, the system should allow the condition and its conditional branches to be removed, enabling a direct transition between the two steps.

- The system must be capable of **editing conditions**, as shown in the *Condition Modification Scenario*. For example, the condition initially indicated that if the temperature is between 36.4°C and 36.8°C , the workflow must continue to the *Evaluate Patient Mental Status* task. After modification, the condition has been changed to state that only if the temperature is between 36°C and 37°C, the workflow will proceed to *Evaluate Patient Mental Status*, and otherwise, it will follow the alternative transition.

- Lastly, when workflow components are added or removed, transitions between the workflow elements must also be able to be updated to preserve the workflow's integrity. The system must support **addition, removal, and origin/target editing of transitions**. For example, in the *Addition Scenario* in Figure 3.6, the left side initially shows three transitions, while the right side shows six transitions, indicating that transitions have been added to connect, as wanted, the workflow components. Similarly, transitions have been removed in the *Removal Scenario*. Also in *Removal Scenario*, the transition that starts from the exclusive gateway and ends at *Measure Systolic Blood Pressure*, after the condition is erased, it could be reconfigured to start from the *Measure Temperature* task, as shown in the right side of this scenario. In other words, if the order of workflow components changes, the system should enable transition reconfiguration to link the correct predecessor and/or successor.

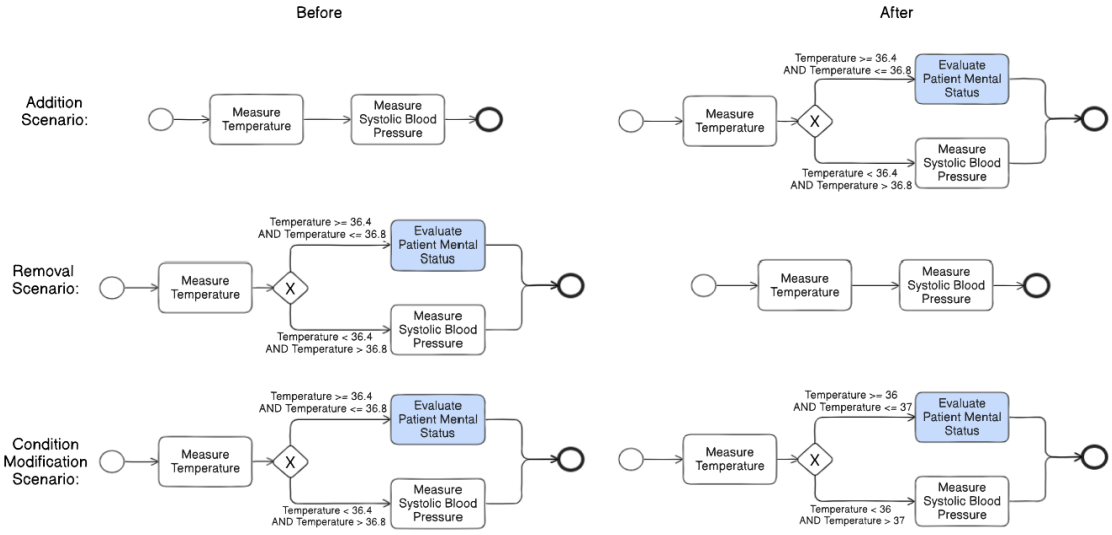



Figure 3.6: Condition & Transition Reconfiguration

Additionally, another relevant topic is the need for patient monitoring in a hospital's Intensive Care Unit (ICU). In ICUs, patients' vital signs are continuously monitored, and tasks such as *Measure Systolic Blood Pressure* must be executed under specific preconditions (e.g., proper sensor placement) and be completed after postconditions are satisfied (e.g., a valid measurement is generated, and no sensor malfunction is detected). Some of these pre/postconditions are essential and should not be configurable, as they are required for task execution in any hospital. For example, in the case of *Measure Temperature* task, sensors must be properly attached to the patient. However, beyond these basic pre/postconditions, hospitals may have additional requirements that vary depending on their internal protocols or infrastructure. Therefore, the system must support configurability of such additional pre/postconditions. Preconditions could be modeled in the workflow with conditional functionality placed before the task that it applies, in the same way as shown in the previous scenarios illustrating condition configuration. Similarly, conditional logic can be added after the task to simulate a post condition. The following scenarios are generated to illustrate this need.

- The system should support the ability to **add, edit, or remove a precondition** for a task. For example, *Perform Biopsy* task should only proceed if the patient has given consent. In this case, the system must allow adding a precondition to check whether consent has been obtained before executing the task. Similarly, it should enable the removal or modification of such preconditions when necessary. Since preconditions are modeled as conditional executions of tasks, the condition configuration scenarios presented earlier demonstrate how these preconditions can be adjusted. The same mechanisms apply to adding or removing a precondition.

- The system should be able to **add, edit, or remove a postcondition** that checks whether certain conditions are met after completing a task. For example, after the *Measure Temperature* task, the system should add a postcondition (conditional logic) to check if the temperature has been measured and falls within logical limits to avoid sensor malfunction. In the same way as preconditions, the post condition should be enabled to be modified and removed.

Finally, in a healthcare setting, events play a crucial role in triggering, continuing, or terminating workflows based on specific patient conditions, statuses, or external stimuli. For example, if a task need to be executed in a specific time frame and this does not happen, an event can be used to notify the related actors to take immediate action. The following scenarios illustrate the capabilities that the system should support in response to events within workflows.

- The system should be capable of **adding or remove an event** to the workflow. For example an event can be added to express a time constraint as shown in Figure 3.7. For example, the system should be able to apply a **time constraint** to the *Measure Systolic Blood* task and take appropriate actions when time limits are exceeded. The time constraint can be configured by changing the time needed for the event to be triggered and enabling the *Notify Doctor* task. In the same way as all the other workflow's components, the system should be able to remove an event from the workflow.

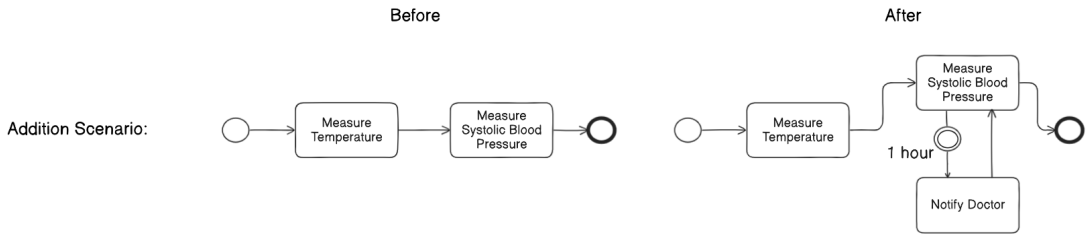



Figure 3.7: Event Reconfiguration

Making all these workflow variation points configurable enables the system to handle the diverse workflow requirements across hospitals without requiring any code-level changes. The system must also validate both the syntactic structure and semantic correctness of reconfigured workflows to ensure their integrity and correct execution. While this section has focused on what kinds of reconfiguration actions should be supported by the system, it has not addressed which actions should actually be permitted in a given context (semantically). These restrictions relate primarily to the semantic validity of the workflow, which was briefly discussed in subsection 3.1.2. Additionally, the system should provide maintainable and easily testable workflow structures to avoid requiring extensive testing and validation of the entire workflow each time a reconfiguration takes place. Finally, the system should offer a user-friendly reconfiguration interface to support users in implementing change requirements efficiently and accurately. In summary, the scenarios described above illustrate the required capabilities of the CWS for achieving workflow configurability.

### 3.2.2 Environment Adaptability Scenarios

Now that scenarios related to workflow reconfiguration and how these changes should be supported by the CWS have been discussed, the focus will shift to scenarios showcasing variability in the CWS's environment across different hospitals and how the system should handle and adapt to them.

- The most certain change identified through the analysis in chapter 3 is the uncertainty of EMR existence in hospitals responsible for storing electronic patient records. So, suppose that the proposed workflows implemented in the CWS assume that EMR exists in the hospitals to retrieve automatically information from it. Now imagine a hospital wanting to acquire CWS, where this system does not exist. Instead, clinicians in such a hospital need to input information manually into the CWS when needed. This is just an example of the necessity of the system's adaptability in different environments because numerous systems could be integrated with the CWS, such as patient monitors, CT scanners, or other hospital information systems apart from EMR. For the purposes of this study, the specific systems do not matter, as the proposed CWS design is intended to be adaptable to any system that may need to be integrated. Also, even if the integrated systems remain the same, they are evolving over time, potentially changing their interfaces or functionalities. For the two above scenarios, the system should be capable of adapting with no-code or minimal-code maintenance. No-code refers to a solution where configuration settings can make the system adapt during setup to the hospital environment, and minimal-code maintenance refers to a situation where the system release needs to be further developed for all hospitals rather than creating a separate release to adapt the system to a specific one. For example, if a new type of system interface has been identified that is possible to interact with CWS, CWS must be extended to enable this system integration for all hospitals. Minimal-code maintenance also means that the new release will not affect the entire system, but only the part of the system responsible for integrating external systems. This makes evident that the system design should be modular enough to handle this requirement. Thus, this scenario captures the necessity of CWS adaptability and emphasizes that it should be designed to consider external system variability without being tightly coupled to it to minimize environmental impact on the CWS.

- Another scenario the system should be able to handle is a situation where more than one external system is responsible for capturing the information needed by the workflow in CWS. For example, the workflow needs to know the patient's surgery history. For this, two systems might be responsible: EHR and EMR. The first is for inter-hospital shared information, and the second is for in-hospital-related information. Another case would be that the workflow requires the results from the CT scan, and these could be retrieved either from the CT machine itself or from the EMR, where they are stored permanently. In these kinds of scenarios, the system should enable the implementation of protocols for handling more than one response for the requested information.

- Additionally, hospitals have multiple systems integrated into their environments. Some already mentioned are EHR systems, CDSS, and, of course, all medical machines, such as patient monitors. So, in an environment with so many systems, there are interoperability protocols based on which different systems can communicate. A hospital is an example of such an environment, so the system should be capable of integrating interoperability protocols.

- A last important aspect is how the system should handle external system failures. For example, what happens if the EHR system is temporarily offline while the CWS has requested data from it. For such a scenario, the system should support the existence of functionality to handle errors that come from external systems.

### 3.2.3 Deployment Scenarios

The last category of scenarios that the CWS should be able to handle concerns the deployment alternatives. As mentioned in subsection 3.1.3, the system should enable scenarios where different deployment options are required.

- Suppose that the system supports only edge-based workflow execution, meaning computation takes place near the patient, such as in a patient monitor. For example, when a patient is in an ambulance with no connectivity, specific workflows might be required to start being executed immediately. Thus, the system design should be able to be deployed in a decentralized manner, in specific devices close to the patient, like patient monitors.
- On the other hand, suppose that hospitals need to leverage the CWS to handle more centralized patient care. For example, the CWS could be deployed in a centralized infrastructure to execute multiple workflows for multiple patients at the same time. In such a scenario, the system must be able to be deployed in a server or cloud, and concurrent workflows must be coordinated and managed to enable such a deployment.
- Finally, if the two above scenarios are combined, a third deployment scenario could be derived for network unavailability cases. Suppose that the workflows that have started being executed in the ambulance, where no (stable) connectivity exists, are transferred to the centralized deployment in the hospital once it is reached so that the workflow execution can continue in the centralized deployment. So, such a scenario illustrates the necessity of both edge computing and centralized deployment.

The above-mentioned scenarios are some context-related examples of the three main requirements related to the CWS, which are enabling workflow configurability, handling environment variability, and supporting the mentioned deployment options.These examples set is a representative set scenarios that the CWS should be able to handle throughout its lifetime in order to be considered as an evolvable system.

## 3.3 System Qualities Selection

After this analysis presented in the previous sections, it is time to summarize and specify the qualities based on which the system design will be developed. The three key drivers qualities that the system design should be based on are the following:

- Configurability: In CWS, configurability refers to the ability to define and modify clinical workflows through external configuration files, without altering or recompiling source code. This includes configuring workflow variation points such as task order, transitions, roles, conditions, and events. A system is considered configurable when the relevant workflow changes can be applied through configuration alone, taking effect without requiring a new system build or redeployment. Configurability is not satisfied if changes require modifying, recompiling, or redeploying system code to update the system behavior.
- Adaptability & Flexibility: Adaptability refers to the ability of the CWS to operate in different hospital environments by reconfiguring its use of external systems without code changes. Flexibility refers to the ease with which these adaptations can be performed at deployment or runtime. The system is considered adaptable and flexible when differences in external systems and infrastructure can be resolved using configuration alone, and no core logic must be changed.
- Integrability: Integrability refers to the ability of the system to connect with new external systems or services without requiring modifications to the existing internal logic or structure. This quality is achieved when new interfaces or connectors can be added independently, without altering existing system behavior.

However, additional nice-to-have qualities have been chosen to complement the key ones:

- Extensibility: Extensibility refers to the ability to introduce new functionalities without requiring changes to existing workflows or system components. The system is considered extensible when new components can be added independently and without causing implementation or testing regressions. Extensibility is not satisfied if existing components must be modified or restructured to accommodate new functionality.
- Scalability: The CWS must be capable of scaling its processing capacity (e.g., number of components instances) in response to varying workload demands (number of workflow instances, patients, etc.).
- Modularity: Modularity means that system functionality is divided into independently manageable components with well-defined responsibilities. The system is modular if components can be developed, tested, deployed, or replaced independently. If a change to one component forces changes in others due to tight coupling, modularity is compromised.
- *Robustness:* Robustness refers to the system's ability to behave as expected under variability in the external environment. It is considered satisfied when the system operates correctly even if different external systems are connected to the CWS. The system is not robust if changes in external systems or temporary unavailability in external systems cause crashes, stalled workflows, or incorrect behavior.
- Maintainability: Maintainability refers to the ease with which the system can be modified to fix bugs, improve performance, or adapt to minor changes. It is considered satisfied when a change can be made in isolation, tested locally, and deployed without requiring changes to unrelated components or revalidation of the entire system. Maintainability is poor when fixes cascade into other modules or require broad testing.
- Upgradability: Upgradability is the ability to upgrade specific system components without redeploying or recompiling the entire system. The system is considered upgradable when systems components can be upgraded independently requiring minimal engineering effort. Upgradability is not satisfied if a component upgrade requires system-wide redeployment or retesting.
- Resilience: Resilience refers to the system's ability to detect, isolate, and recover from partial failures (e.g., unavailable external systems) while continuing workflow execution. The system is resilient if it retries, reroutes, or gracefully degrades when failures occur, and workflow state is recoverable.

The set of qualities shown above includes those primarily related to evolvability, as identified in the literature. Additionally, an importance-based taxonomy has been developed to highlight the qualities most relevant to the specific problem context of this study. These will be considered during both the system design and evaluation phases. To summarize this chapter, the main requirements, their related change scenarios, and the corresponding system qualities are summarized in Table 3.3.

| Rank | Requirement | Relevant Change Scenarios | Related System Qualities |
|---|---|---|---|
| 1 | Workflow Configuraility | All in subsection 3.2.1 | Configurability, Modularity, Maintainability, Extensibility |
| 2 | System Adaptability, Flexibility & Integrability | All in subsection 3.2.2 | Adaptability, Flexibility, Integrability, Extensibility, Modularity, Maintainability, Robustness, Resilience |
| 3 | Deployment Flexibility & Adaptability | All in subsection 3.2.3 | Adaptability, Flexibility, Extensibility, Modularity, Maintainability, Upgradability, Scalability |

Table 3.3: Mapping of Requirements to Change Scenarios and System Qualities

# Chapter 4

# System Architecture

In the previous chapters, the system's three key requirements were thoroughly analyzed, supported by concrete change scenarios to make them specific and testable. Relevant system qualities were also identified and prioritized, clarifying the overall design goals. This chapter presents a system architecture that addresses these requirements and scenarios, and explains how specific design decisions contribute to satisfying them. Potential alternative approaches are also discussed where appropriate.

## 4.1 Solution Approach & Components

We begin by focusing on the solution covering the requirements regarding workflow configurability. In the chapter 1, all the solutions provided for facing the stated or related problem statements, [32, 38, 65, 66, 67, 68], have a common ground between them and this is that all of them utilize an execution engine notion which is responsible for interpreting workflow definitions and orchestrates their execution. In the current work, the idea is to include a single execution engine capable of interpreting any workflow definition, provided it adheres to a specific syntax. When a workflow is triggered, the execution engine instantiates and manages a corresponding workflow instance, which represents the execution of a workflow definition in a specific data-driven context (e.g., patient-specific data). This separation between workflow definition and instance enables the system to accommodate changes in workflow logic without requiring modifications to the execution engine itself. This decoupling is a foundational principle of the proposed solution.

Building on this, a key concept in developing a configurable CWS is to avoid hardcoding workflow definitions within the system. Instead, workflows are expressed through configuration files interpreted by the execution engine. These files specify the structure and behavior (control logic) of a workflow, including its tasks, conditions, transitions, and events. Configuration files are a natural fit because workflows are inherently structured and rule-based. They define what needs to happen, when, and under what conditions. In this project, such files are used to describe the flow of workflow elements in a way that can be easily adapted and reused. As a detailed workflow definition explanation has already been provided in subsection subsection 3.1.1, the configuration file can include all the mentioned components needed to define the workflow and be fed to the execution engine, which interprets it and executes the defined flow.

With the provided explanation of the approach so far, it can be understood that workflows can be defined in configuration files without the need for system code-level redevelopment or rebuilding the execution engine (or other parts) of the system to interpret new versions of the workflow definitions expressed in these files. It is also evident that, due to the nature of configuration files, they do not require compilation and can be directly fed to the execution engine, which is designed to interpret and execute them at runtime. While a configuration file defines the workflow's control logic, such as the sequence of tasks, conditions, events and transitions, it does not by itself implement the functional logic of each task. For the system to execute a task (e.g., measuring

temperature), an implementation of that functionality must exist and be associated with the task. This linkage is established directly within the configuration file, where each task definition includes a reference (e.g., a function name or service endpoint) that points to the actual implementation. At runtime, the execution engine uses this reference to invoke the corresponding functionality and carry out the task.

However, this approach has a limitation. If the tasks or their associated functionalities are not modular, any required changes may still need to be made at the code level, which reduces the benefit of using configuration files. Modular task implementations, those designed as independent, reusable units, are essential to preserve the benefits of configurability. The problem arises when functionalities are implemented in a monolithic or tightly coupled manner, making it difficult to use parts of them independently or reconfigure them based on changing workflow needs. To avoid this, the linked functionalities should be designed to support configurability, extensibility, adaptability, and maintainability. To illustrate the explained limitation, suppose the CWS includes a code-level functionality linked to a workflow task for measuring all vital signs (temperature, heart rate, respiration rate, and blood pressure) simultaneously. If a specific hospital needs to measure only one of these (e.g., temperature) independently, and the implementation does not allow this separation, code-level changes would be required. This undermines the benefits of a configurable CWS. Therefore, designing functionalities in a modular fashion, each responsible for a single, well-defined operation, enables flexible task linking, promotes reuse, and simplifies workflow adaptation to meet specific clinical needs.

### 4.1.1 Unit Functions

Due to the aforementioned considerations, the proposed approach uses *Unit Functions (UFs)*. This is one of the most critical aspects that enables workflows to be easily configurable and minimize the need for workflow testing when a reconfiguration occurs. UFs are code-level implementations of functionality, responsible for encapsulating and executing a single, fine-grained clinical task that is not meaningfully decomposable. For example, systolic blood pressure measurement is a functionality that must be performed in a specific, rarely changeable way, and there is no benefit in breaking this action into smaller tasks. Therefore, the functionality that implements how systolic blood pressure is measured can be developed once and reused as a unit block in workflow definitions as many times as needed throughout the CWS life cycle. Other examples of such UFs include vital signs measurement (e.g., temperature, heart rate), medication administration checks, eligibility verification (e.g., immune status, recent surgeries), and communication-related functions that will be explained in the next sections (e.g., sending alerts or triggering workflows).

A comprehensive set of UFs must be implemented in advance for this approach to be functional. Once the UFs have been developed, they can be extensively tested, compiled and serve as unit building blocks that can be integrated into workflow definition tasks. This way, when the workflow is reconfigured, the functionality required for each task has already been tested, ensuring correct task execution. Thus, only the control logic defined in the workflow definition must be tested, and not the underlying functionality. The main goal of this idea is to provide a solution that decouples the control logic of workflow definitions from code-level functionality implementation, and this is precisely what the proposed approach, using UFs, achieves.

Going one step further, it should be mentioned that UFs themselves do not directly perform operations such as temperature taking. Instead, they define and encapsulate the functionality needed for a given workflow task. The actual operation execution of temperature measurement is carried out by external systems, such as a patient monitor. A separate component of the CWS (adaptor) is responsible for actually interacting with external systems to carry out the actual functionality. UFs simply define and expose an interface with the required task functional logic to the execution engine. As a result, UFs remain independent of how the actual operations are implemented or how external systems are accessed. This allows UFs to focus solely on defining what functionality is needed, without being concerned with its execution details.

### 4.1.2 Adaptors

Having explained the contribution of UFs to the workflow configurability of the system, we now shift the focus to adaptability across different hospital environments. To achieve this, the design of UFs must be decoupled from the specific external systems that carry out their requests. For example, a UF responsible for measuring temperature should not depend on communicating with a particular patient monitor, since different hospitals may use different systems for this function. To enable this decoupling, a set of *adaptors* is placed at the boundaries of the system. Each adaptor is responsible for interacting with external systems of a given type, invoking their services through the appropriate interoperability protocols and processing their responses. In doing so, adaptors also translate the external system's data formats and semantics into an internal representation, referred to as the CWS internal notation, which ensures that other internal components can correctly interpret and use the received data. A type of external system is defined by core features such as its functional role, interaction behavior, interoperability protocol, and any additional characteristics that might be relevant.

Through this approach, adaptors provide adaptability of the CWS to different external system types, while ensuring that the core system remains independent and interoperable across hospital environments. Specifically, adaptors handle the execution of UF requests that require external services, such as taking temperature measurements or retrieving a patient's surgical history. Additionally, adaptors can provide interfaces to external systems to interact with the CWS, for example, to trigger workflow execution.

Like UFs, adaptors are designed as services. For each identified external system type, a corresponding adaptor must be implemented to support this architecture. Therefore, an analysis should be conducted to identify the complete set of relevant external system types. Once the necessary adaptors are implemented, the system can invoke the services of the external systems without the need to redevelop components of the system in case that system variability exist across hospitals. Instead, the appropriate adaptors can simply be enabled or disabled to meet the existing system types of each hospital. Importantly, each adaptor also undertakes the responsibility of determining which specific external system, of the specific type, to invoke for a given UF request. For example, an adaptor responsible for patient monitors must maintain the mapping of which monitor is assigned to which patient, ensuring that requests are routed to the correct device.

Finally, adaptors enable the CWS to be designed and implemented independently of the interoperability protocols used by external systems. Without adaptors, the system would need to be built for a specific interoperability protocol, limiting its ability to adapt to other protocols. By introducing adaptors, this separation of concerns is achieved, ensuring the core system remains protocol-independent.

Having explained some of the core ideas and components of the system design, multiple other components have been integrated based on the requirements gathered during the analysis to construct the entire system architecture. The proposed CWS architecture is shown below in the Figure 4.1. Each rounded corner rectangle represents a system component, and the arrows indicate the flow of data and control between components, such as workflow definitions, service requests, and responses. These represent logical interactions between the system's components. Let us analyze the rest of the components individually, and then diagrams detailing their interactions and exchanged data types will be provided.

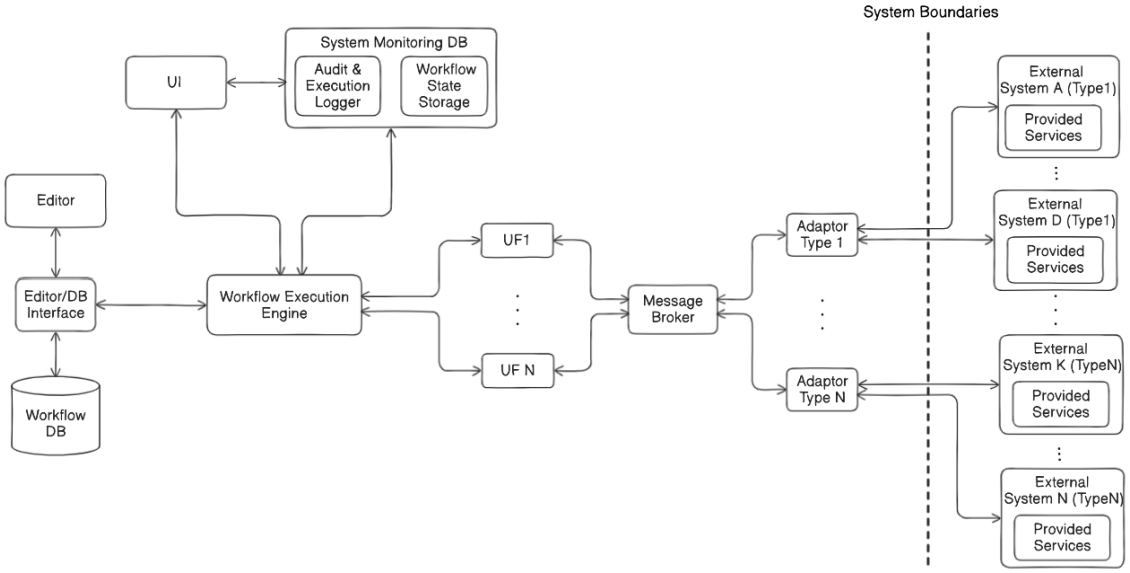



Figure 4.1: Proposed Architecture - Logical View

### 4.1.3 Workflow Editor & DB

A database, *Workflow DB*, stores configuration files, each containing the definition of a single clinical workflow. There are multiple modeling languages for the workflows to be expressed, one of which is Business Process Model and Notation (BPMN), which will be analytically explained in the chapter 5. Alongside *Workflow DB*, an *Editor*, through an *Interface*, can load a desired workflow definition and reconfigure it. The editor can create and store new workflows in the database as well. It provides the CWS workflow designers with the necessary tools to perform workflow (re)configuration actions, which are implemented by the editor, to adapt the loaded workflow to the specific hospital needs. In addition, the editor ensures that the generated reconfigured workflow conforms with both syntax rules so the execution engine can interpret and execute it without errors, as well as semantics required for ensuring clinical validity, such as adherence to domain standards defined by organizations like WHO. A list of the implemented UFs can be offered to assign functionality to tasks within the workflow. Through the editor, a specific UF can be linked to a task and executed once the workflow is executed and the specific task is triggered. Finally, after the reconfiguration, the editor enables the user to deploy the workflow definition file to the execution engine to be ready for execution. Initially, workflow definitions can either be developed using the CWS editor, following the proposed workflow guidelines, or directly stored in the database if already available.

### 4.1.4 Execution Engine

One of the core components of the proposed system is the *execution engine*. Its main functionality is to receive requests for workflow execution, interpret a workflow definition from its configuration file, instantiate it, orchestrate workflow execution, and trigger the execution of UFs linked to specific tasks. A workflow definition serves as a blueprint, while the execution engine is capable of handling multiple instances of that definition simultaneously, for example, one instance per patient. Users can modify workflow definitions by updating the configuration file via the editor instead of writing workflows directly in code, which would require recompilation of the CWS. The execution engine processes workflow definition files consistently. While the editor enforces the syntax and semantic validity of the definitions, the engine can reliably interpret and execute

them, even after reconfiguration. As a result, the execution engine remains unchanged even as workflow definitions evolve.

### 4.1.5 UI & System Monitoring DB

In addition to the execution engine, components for *storing the state of each workflow being executed* and for *audit execution logging* also exist. The workflow state represents the set of active tasks in a workflow instance, or in Petri Nets terms, the distribution of tokens during execution, corresponding to a specific patient. It also includes the set of completed tasks, as well as all variables and their current values for that patient. The execution logger can store more extensive execution logs, both for successful task executions and for erroneous ones. Displaying this information to end-users or administrative staff is essential for supporting task completion and issue resolution. In case of workflow execution failure or system restart, the *System Monitoring DB* component can be used to retrieve the most recent workflow state and support recovery of the workflow instance. In total, the *System Monitoring DB* component fulfills this role, acting as a database that stores the current state of workflow executions. Other system components (such as the UI) retrieve this information for updates. A *user interface (UI)* is provided for user interaction, for displaying all this information, as well as handling user input for tasks that need to be executed manually and where input is required.

At this point, mentioning that the system might have multiple UIs is helpful. One case could be the UI illustrated in Figure 4.1. However, a UI could also exist in an external system. For example, consider a case where CWS is deployed in a centralized manner, and one workflow's task is to measure the temperature of a specific patient. Unless the task is automatically executed by a system like a patient monitor, manual measurement and input by the clinician might be needed. This input might need to be provided through the patient monitor UI. The proposed system supports this scenario by passing the relevant workflow state to the appropriate UF, which forwards it to the adaptor. The adaptor then communicates with the external system acting as the UI, enabling it to render the task and collect the required input.

### 4.1.6 External Systems

An *external system* is any system that the CWS needs to interact with to complete a workflow being executed. As explained in the previous section, external systems interact with the CWS through the designated adaptors. For example, if a workflow needs to automatically measure a patient's temperature, the patient monitor, which is capable of executing such a request, is an external system to the CWS. These systems can include medical devices (such as patient monitors), user interfaces for clinicians to enter data manually, or information systems such as EMRs used for storing and sharing patient records. External systems can get invoked by CWS requests (e.g., measure a patient's temperature) and send requests to the CWS. For example, suppose a patient monitor has an integrated alert that, under specific conditions, needs to trigger a workflow in the CWS, e.g., a sepsis workflow that evaluates whether the patient is in a septic condition. In this case, the CWS receives the request from the external system via the designated adaptor and triggers the sepsis workflow execution, in the execution engine, using a designated UF.

### 4.1.7 UFs - Adaptors communication (Message Broker)

A key decision point for the design is how the UFs communicate with the adaptors. As mentioned, the design of UFs should be decoupled from the external environment's variability, which is precisely why adaptors were introduced. Even though the responsibility for interacting with external systems has been delegated to adaptors, UFs should also remain decoupled from the adaptors themselves. Coupling them would be equivalent to coupling the UFs directly with the external systems. The most standard way for UFs to communicate with adaptors would be to know precisely which adaptor to call, hardcoding the communication into their implementation. However,

this introduces a point-to-point connection between UFs and adaptors, effectively coupling them and thus contradicting the original design goal of decoupling. One valid option for achieving this decoupling is the blackboard style, where a standard knowledge base exists, and both UFs and adaptors can read from and/or write to it without needing to know about each other. For example, a UF could write its requests to a database (the blackboard), and the adaptors would poll the relevant tables or fields, retrieve the requests or information, execute them accordingly, and then write back the responses. However, both mentioned approaches introduce specific challenges.

The first and most significant issue with point-to-point communication is that the UFs and the adaptors become tightly coupled. This means that communication is based on implementing the UFs to invoke a specific adaptor directly. Suppose there is no EHR in a hospital, and EMR should be used to retrieve patient information. In this case, the UF responsible for retrieving the patient's information, rather than communicating with an adaptor connected to EHR, would need to communicate with an adaptor connected to EMR. Such a scenario can easily lead to the need to customize the implemented UF in case of a point-to-point connection of UFs with adaptors. The second issue relates to the blackboard approach. While its simplicity is a strength, it provides no synchronization between the components that generate a request and those that should receive it. If the UFs and adaptors are connected via a properly designed database, then whenever a UF wants to send a request to an adaptor, it must write it to the database. Since the adaptor does not receive any notification of this event, it must frequently poll the database to stay updated. Similarly, when the adaptor writes back the response, the UF waiting for it must also poll the database to detect when an update has occurred. Thus, this approach introduces a semi-structured communication protocol.

To overcome these drawbacks and enable dynamic, loosely coupled communication between UFs and adaptors, the publish-subscribe architectural style is considered and applied. This approach offers two main advantages. Firstly, it decouples the UFs from the adaptors. The publisher can publish the desired message or request using topics, and the *message broker* forwards the message to all adaptors subscribed to that topic(e.g., EHR type and EMR type adaptors). Let us suppose again that the UF responsible for retrieving patient information is being executed. Instead of the UF needing to know which adaptor is responsible for interacting with the designated external system, it only needs to know the topic to which to publish the message. Similarly, the receiver, such as the adaptor responsible for communicating with the patient monitor, does not need to know the message's publisher. It simply receives the message on the subscribed topic and executes the required request. This critical advantage makes the system highly adaptable and configurable to different hospital environments with no code-level changes. This advantage is also applied to the other direction of data flow, from the adaptors to UFs. When the adaptor has received the response from the external system and translates it to internal CWS notation, it just publishes the response message to a designated topic to which the UF, initially made the request, is subscribed. Thus, adaptors also do not need to know from which UF the message/request comes, they only need to know the topic on which to publish the response.

To explain in more detail how *publish-subscribe* works, let us explain its four key components [69]. *Messages* are communication data sent from sender to receiver. Messages can have many data types, from strings to complex objects or others. Every message has a *topic* associated with it. The *message broker* acts like an intermediary channel between senders and receivers. It maintains a list of receivers interested in messages about that topic. *Subscriber* is the message recipient and has to register (or subscribe) to topics of interest. Finally, *publisher* is the component that sends messages. It creates messages about a topic and sends them to a message broker, which forwards them to all subscribers who have subscribed to receive messages for this topic. This interaction between the publisher and subscribers is a one-to-many relationship. Also, the publisher does not need to know who is using the information it is publishing, and the subscribers do not need to know where the message comes from. However, in practice, this information may still be useful, for instance, for logging, debugging, or coordination purposes, and can be included in the message payload if needed. In our case, UFs and adaptors act like publishers and subscribers. UFs are publishers when they need to send a request to adaptors and subscribers when waiting for adaptors' response. Adaptors are subscribers when waiting for messages from UFs to carry out requests and

publishers when they publish responses or requests from external systems to UFs.

UFs, as described, are services responsible for implementing a single-responsibility functionality. So, one UF is subscribed to a unique topic. For example, the *Measure Temperature UF* will subscribe only to *Temperature.Response* topic to receive any message responding to temperature requests. Accordingly, they publish only in one topic, e.g., *Temperature.Request* to request temperature taking from the designated external system through the adaptors. So, only *Measure Temperature UF* will receive any message related to temperature measurement response. In the same way, each adaptor subscribes to any topic related to the external system's functionality with which they are interacting. For example, the *Patient Monitor Adaptor* will subscribe to *Temperature.Request*, *Heart.Rate.Request*, *Blood.Pressure.Request* and other topics related to the patient monitor functionality. Finally, adaptors publish the respective measurements received from the external system on the response topic related to the topic of the incoming request. For example, when a message is received on the topic of *Temperature.Request*, the adaptor will send the response message on *Temperature.Response* and so the *Measure Temperature UF* will receive the wanted temperature value. Simply put, the relation between UFs-adaptors is one-to-many because there is a case that more than one external system can handle a UF request. In contrast, the relation between adaptors-UFs is one-to-one because the messages published by adaptors end up on a single UF. To ensure the correct adaptor responds to a received request, each adaptor maintains a local mapping of patients to the external systems or devices they are associated with. Finally, all exchanged messages include the necessary metadata (e.g., patient ID, workflow instance ID etc.) to identify the relevant patient and correlate the response with the originating workflow instance.

Additionally, this approach supports the scenario mentioned in the requirements analysis, where more than one system may hold information about a patient (e.g., EHR and EMR). In such a case, the UF requesting the patient's surgical history can publish a message received and processed by multiple systems. Of course, this advantage introduces the need for UFs to include additional functionality or protocols to handle multiple responses appropriately. In general, each component maintains scoped knowledge: adaptors are aware of the external systems they interface with, the message broker manages topic-based routing between UFs and adaptors, and workflow definitions determine which UFs are invoked and when. Lastly, using this event-driven approach, synchronization is handled by the message broker by default. However, scenarios involving lost messages or message broker failures must be considered part of a fault analysis.

Finally, it must be mentioned that the design of UFs and Adaptors aligns well with Service-Oriented Architecture (SOA) principles. SOA promotes modularity, reusability, and separation of concerns by decomposing system functionality into loosely coupled, independently deployable services [70]. Traditional SOA implementations often rely on point-to-point communication using synchronous protocols. However, the architecture presented in the current work adopts an event-driven communication model, where services interact asynchronously through a message broker [71]. While the communication style between services differs, the core SOA principles, modularization, abstraction, and independent service evolution, are preserved. Using asynchronous messaging (message broker) enhances decoupling and scalability, making the system more resilient to changing scenarios. To conclude, in Table 4.1 the explained system components are summarized and with their relevant high level responsibilities.

| Component | Responsibility |
|---|---|
| Unit Functions (UFs) | Single-responsibility services; Encapsulates functional logic; enable configurable and reusable workflows. |
| Adaptors | Interface with different types of external systems; manage interoperability (protocols, data formats); maintain patient-to-system mappings and resolve invocation priorities. |
| Execution Engine | Instantiates and orchestrates workflows based on definitions; triggers appropriate UFs for task execution; manages workflows instances states. |
| Workflow DB | Stores workflow definitions in a structured configuration format (e.g., BPMN). |
| Editor | Enables creation and reconfiguration of workflow definitions; ensures syntactic correctness and clinical semantic validity. |
| UI | Displays workflow state and logs; enables interaction with running workflows; collects manual input when required. |
| System Monitoring DB | Stores workflow instance states and execution logs; supports UI display and failure recovery. |
| External Systems | Execute medical operations (e.g., measuring temperature); communicate with the CWS via adaptors; may also trigger workflow execution. |
| Message Broker | Facilitates asynchronous publish-subscribe messaging between UFs and adaptors; ensures complete decoupling between UFs and adaptors. |

Table 4.1: Overview of CWS Components and Responsibilities

## 4.2 System Behavior & Solution–Requirements Alignment

This section describes the system's components interactions and demonstrates that the proposed architecture supports the three main requirements described in subsection 3.1.4. This section focuses on the system's design explanation, while remaining independent of implementation details. In chapter 5, it will be explained, in detail, how each change scenario shown in section 3.2 is handled by the system proof-of-concept implementation.

### 4.2.1 Workflow Reconfiguration

Let us start by explaining how the workflow reconfiguration works in the proposed system, handling the variation points mentioned in Table 3.1. The components' interaction during the workflow reconfiguration is shown in Figure 4.2. Using the *editor*, the user responsible for reconfiguring the workflows starts by selecting the specific workflow definition to be reconfigured. Based on the selected workflow, the editor through the *Editor/DB Interface* queries the database for the workflow definition file with the specific ID or name. The database executes the query and responds with the workflow definition file matched with the specific given ID. The user can now start reconfiguring the workflow using the editor, which has integrated functionality capable of applying reconfiguration actions for the mentioned variation points. For example, the editor has functionality that allows the user to add, remove, or reorder tasks or similarly act upon other workflow elements such as conditions, transitions, or events.

The editor validates every reconfiguration action to ensure that the syntactic structure of the workflow remains correct and executable. In addition to syntax validation, it also enforces semantic constraints to prevent configurations that may be structurally correct but violate clinical logic or workflow semantics. Once all reconfiguration steps are completed and the workflow passes validation, the user can save the updated workflow definition using the editor, which interacts with the database to store the file. Finally, for the execution engine to be able to execute the updated workflow definition, the definition must be deployed in the execution engine. Thus, the user indicates the deployment workflow name, and the editor deploys the workflow definition file by using the editor's interface to interact with the *execution engine*. Once the workflow definition file is deployed, the execution engine can receive workflow trigger requests to execute the specific workflow.

To reflect on the requirements, a reasonable question to be answered is how this solution handles the reconfiguration of workflows without code changes, system testing, or rebuilds. The main point of this solution is that workflow definitions are configuration files. Thus, if they are reconfigured based on the syntactic and semantic rules the editor enforces and ensures workflow definition compatibility with the execution engine, then the workflow execution engine can interpret and

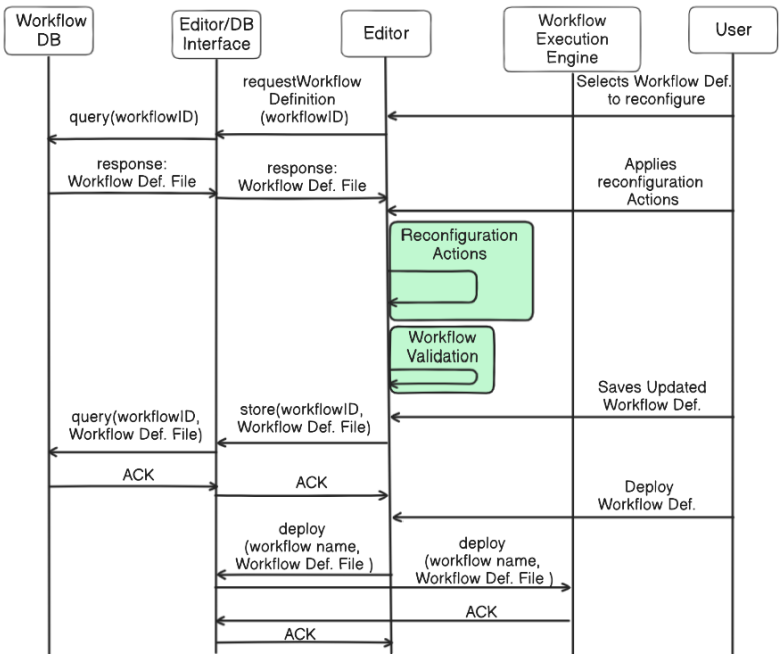



Figure 4.2: Workflow Reconfiguration Sequence Diagram

execute the updated workflow definition file without needing any customization on the execution engine side. Syntactic rules include the correct use of workflow elements (e.g., start/end events, valid task transitions), while semantic rules ensure clinical correctness, such as enforcing task order and logical dependencies (e.g., diagnosis before treatment). Of course, as explained in section 4.1, a central point that enables this easy reconfiguration is the notion of UFs working as modular building blocks. While the editor can be used to customize the workflow definition, users can use it to link the tasks with the appropriate implemented UFs to define the functionality that needs to be triggered and executed upon a task execution.

It is important to clarify that the reconfiguration of a workflow definition affects only future workflow instances, as only the updated definitions are deployed to the execution engine. Workflow instances that are already running at the time of a definition update will continue to execute based on the version of the definition they were instantiated with. This separation ensures that system behavior remains stable during updates and avoids the risk of inconsistency during ongoing executions.

### System Design - Components Interaction

Before continuing with the demonstration of the remaining requirements, let us showcase how the rest of the system's components (*UI*, *System Monitoring DB*, *Execution Engine* interact. These components interact to trigger a workflow from the UI, execute manual tasks through the UI, and in both cases, to update the workflow state in the UI. The first case is depicted in Figure 4.3. So, the user selects, in the UI, a workflow to be executed with a specific name. Additionally, it is assumed that the user provides the patient ID for which the workflow should run. The UI, in turn, sends a message to the *execution engine* containing the workflow name and the specified variables, thereby triggering the execution of the workflow. Then, an internal process occurs for the workflow instance to be initialized. Once execution has started, the *execution engine* generates a process ID to identify the workflow instance uniquely, initializes the workflow state, and inserts the given variables into the workflow's instance variables. Then, the *execution engine* stores the newly created workflow execution instance state in the *system monitoring DB*, along with all the necessary related data. Finally, the UI is updated by the *system monitoring DB* component and

displays this information to the user.

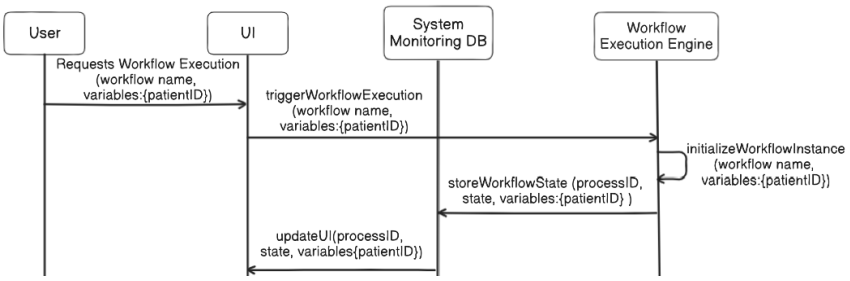



Figure 4.3: Workflow Execution Trigger

Similarly, as depicted in Figure 4.4, when a manual task needs to be executed via the *UI*, the user gives the required input to the task rendered in the *UI* and marks it as complete. So the task is manually executed by a user. Each task is identified with a unique task ID in the workflow instance it belongs. The *UI* notifies the *execution engine* with a message including the process ID, the task ID that has been executed, the user input, and the task status, and on its turn updates the workflow instance state and variables. Finally, the *execution engine* updates the workflow state in the *system monitoring DB* component, which updates the *UI* accordingly with the new state to be rendered.

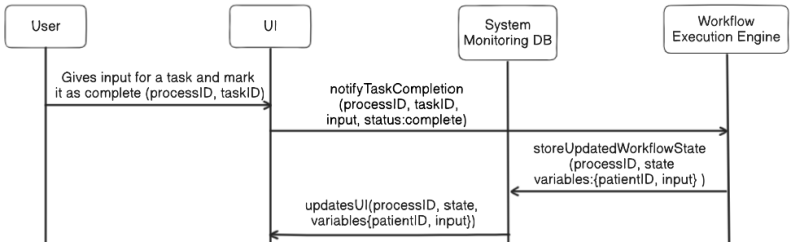



Figure 4.4: Manual Task Execution by User and Resulting System Interactions

Finally, to explain how the *Execution Engine*, *UFs*, *Message Broker*, and *Adaptors* interact, let us first clarify the relationship between *UFs*, *Adaptors*, and *External Systems*. Conceptually, each UF communicates with all adaptors connected to external systems that provide functionality related to the UF's purpose. Each adaptor maintains information about which of the external systems, it is connected to, are responsible for handling a specific UF request (for example, based on patient assignment). When a UF sends a request, all relevant adaptors receive the message. Each adaptor then evaluates whether any linked external system is responsible for handling that request, and if so, it invokes the appropriate one. Such a scenario is depicted in Figure 4.5.

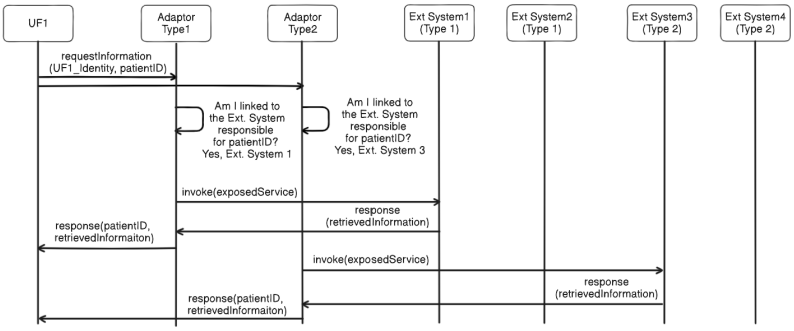



Figure 4.5: High-level interaction between UFs, Adaptors, and External Systems.

To make this more concrete, consider the example of a UF that needs to retrieve the temperature of a patient with a specific patient ID. If there are two different types of patient monitor adaptors in the system, both will receive the request. Each adaptor will then check whether it is connected to the patient monitor assigned to that patient. Now consider the scenario where two patient monitors (one from each type) are dedicated to the same patient. In this case, both adaptors will invoke their respective patient monitors, and both may send responses to the UF. The UF can be implemented to handle this case by, for example, accepting the first valid response received. In contrast, if only one patient monitor is assigned to that patient, then only the corresponding adaptor will send a request to its linked monitor, while the other adaptor may either ignore the incoming message or send a message to UF, indicating that no assigned patient monitor is available for this request.

Now the responsibilites of each component has explained clearly. Thus, let us analize how actually the componnts interact using the proposed system design. During the system initiation phase, all the *UFs* and *adaptors* notify the *message broker* to which topics they are interested. Thus, the message broker constructs a list with this information to appropriately route the incoming messages. This is depicted in Figure 4.6.

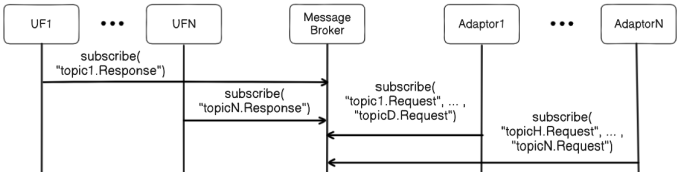



Figure 4.6: Message Broker - Initiation phase

The interaction between components during the execution of a workflow task by a *UF* is shown in Figure 4.7. Let's assume that for executing *UF1* only *Adaptor (Type 1)* is relevant and only *External System 1 (Type 1)* can respond to requests made by *UF1*. In such a case, the *execution engine* triggers the *UF1* by sending a request, including the process ID, the task ID for which the UF needs to be executed, and the workflow instance variables. *UF1* publishes a message, containing the process ID, task ID, and variables, to a topic on the *message broker*. The *message broker* receives it and, having the list of all the subscribers to the specific topic, forwards it to the subscribed ones, in this case, *Adaptor Type 1*, responsible for interacting with the appropriate external system. Upon receiving the message, the *adaptor* calls the appropriate exposed service of the *External System 1*, passing the necessary data.

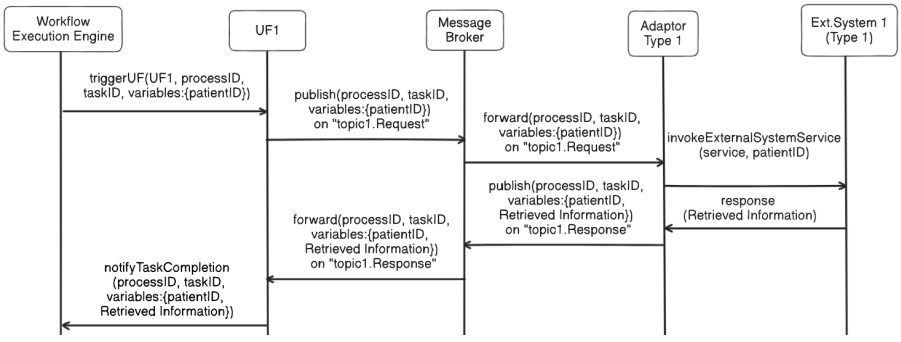



Figure 4.7: Task Execution using External Systems

When the response is returned by the external system, including the retrieved information, the *adaptor* publishes this data on the response topic on which the UF has subscribed to receive the response of its message initially sent. The *message broker* forwards this response back to the subscribed *UF*, which in this case is *UF1*. Upon receiving the response, the *UF* assigns the returned information to the workflow variables in the context of the given process ID and marks

the task as completed. Then *UF* notifies *execution engine* that the task with the specific task ID has been completed successfully, thus allowing the workflow execution to continue. As shown in the previous diagrams, after the task is completed, the *execution engine* updates the workflow state in *system monitoring DB*, and the *UI* gets updated with the new workflow instance state and variables.

### 4.2.2 Environmental Variability

Having explained all the interactions of the system components, we can continue by verifying the system's capability of handling external environmental variability. UFs, the message broker design selection, and the adaptors play an important role in handling environment variability. A UF responsible for retrieving patient surgery history, for example, could publish a message on the "Patient.Surgery.History.Request" topic, which is routed through the message broker to all subscribed adaptors. Adaptors, with the proposed modular design, can be set as active (enabled) or inactive (disabled) during system deployment based on the availability of the external system. If both the EMR and the EHR adaptors are subscribed to this topic, they will receive and handle the request message appropriately. So, following this example, if the EHR is unavailable in a specific hospital and thus disabled in the system settings, and the EMR is available, without any code-level changes in the UFs or adaptors, the EMR adaptor (which will be enabled in that case) will receive the message and carry out the received request. It is important to mention that this approach heavily relies on the fact that the set of adaptors has been carefully crafted after thorough analysis to ensure that one adaptor exists for every required functionality and external system option. However, what happens if multiple systems responsible for the same functionality are enabled? For example, what if both EMR and EHR are available, and thus, their adaptors are enabled in the system settings? In this case, the UFs or execution engine should undertake the responsibility to handle the multiple-response scenario, e.g., accept the first response received.

Additionally, adaptors play a significant role in handling interoperability between heterogeneous systems. In system-of-systems (SoS) environments, such as clinical settings, multiple subsystems from different vendors must work together, even though they often use different interfaces, technologies, and data formats [72, 73]. As these systems evolve, maintaining direct integrations becomes complex and error-prone. In our case, interoperability[74, 72] is essential for enabling the CWS to communicate with existing hospital systems. Adaptors can adopt HL7 FHIR [75], or any other appropriate interoperability standard, to align with the protocols used in each hospital environment. Since each adaptor is responsible for one external system, different interoperability protocols can be used per adaptor, which is sometimes unavoidable, as not all vendors follow the same standards.

Another consideration related to the adaptors and external systems is what happens if an external system does not respond to the adaptor's request and how the CWS should handle this. In such a case, the adaptor can notify the UF that sent the request to it using the appropriate response-topic. Generally, the CWS relies on the fact that the external systems work correctly and are available. However, there should be mechanisms to be resilient to these failures. One approach to consider is that the workflow definition includes redundancy through manual tasks. For example, let us consider that the *Measure Temperature* task is usually executed utilizing a UF, which, through the adaptors, invokes an external system like a patient monitor to retrieve the patient's temperature. If the external system exposes an API, the adaptor can use it to invoke the required functionality and process the response accordingly (e.g., store it in the expected location). This is the default assumption. However, if the patient monitor is not responding, the adaptor can have a timeout and let the UF know that the patient monitor is not reachable. The UF, in turn, notifies the execution engine about the situation, and at this time, a redundant manual task can be triggered to be executed by a nurse who manually enters the temperature value into the system UI. Though, relying on manual fallback tasks introduces a new class of failure, like human-related delays (e.g., the nurse may be slow). To mitigate this, the workflow can integrate time constraints in the fallback manual tasks for raising alerts or escalations, ensuring that workflows do not stall indefinitely due to human inaction.

Another solution to avoid complicating the logic within the workflow definition is to implement an alternative flow directly within the UF. For instance, if an automated task fails to execute via the patient monitor, the UF can handle the failure by sending a message to a different adaptor. This adaptor is meant to pass the task that needs to be executed to an external system/UI, allowing a nurse to manually input the temperature value and complete the original task without implementing the redundant task inside the workflow, thus simplifying reconfiguration and reducing testing effort. Both approaches have their pros and cons. Adding redundancy to the workflow definition can be a good option because it keeps the UF implementation simple and responsible only for one functionality. However, this complicates the workflow definition and makes it harder to maintain. On the other hand, implementing the alternative flow inside the UF can be a test-free solution when the workflow has been reconfigured, as the entire UF functionality has already been tested. However, it makes the UF responsible for multiple functions (executing both automated and manual tasks). Thus, to alter this logic, the UF must be customized at the code level, contrasting with the initial requirements. Given the system's design principles and requirement to keep UFs single-purpose and testable, the first approach is preferred.

A final consideration related to the adaptors concept is that if they are used to trigger workflows from external systems, this assumes that the external systems know what request to make and where (endpoint). This may be an unrealistic scenario for external systems that are not provided by Philips because the systems' vendors would need to update their systems in order to invoke this CWS's functionality. So, a simplified scenario is considered, and this is that workflows are triggered either from external systems provided by Philips or from the CWS UI. However, it should be noted that this issue also persists with alternative options, such as the blackboard approach, where the external systems write their requests into a database. Even in that case, the external systems still need to know where to store their requests or responses. Thus, this is a general constraint, namely, the requirement for external systems to have prior knowledge of the correct request format and endpoint, and not specific to the adaptor concept.

In summary, environmental adaptability is achieved through the UFs design, the use of the publish/subscribe architectural style and the adaptor-based design. By enabling only the adaptors corresponding to the systems available in a given hospital, the CWS can adapt to different environments without requiring code-level changes.

### 4.2.3 Deployment Responsibilities

Now, let us move to the final requirement concerning the system's deployment responsibilities. The question is whether the proposed system can handle different deployment scenarios to adapt to varying needs. A key strength of the proposed architecture is the modularization of its components, which enables them to be deployed completely independently for decentralization and scale accordingly when needed. However, the two main deployment scenarios are the centralized and the per-patient approaches. The first handles workflow execution for multiple patients within a shared infrastructure, while the second involves deploying a dedicated instance of the system for each patient. Figure 4.8 shows a system deployment that supports the centralized scenario, which we will now explain in more detail. The core capability required for centralized deployment is that all system components can be deployed on a centralized infrastructure, such as a hospital server or a cloud platform. The modular design of the architecture enables horizontal scaling of components (such as adding more UF instances) to handle increased workloads, which is why the proposed architecture strongly supports this deployment scenario. Specifically, the workflow execution engine, UFs, system monitoring DB, adaptors, and message broker can all be deployed centrally. For this to work effectively, the allocated resources of the server or cloud environment must be sufficient to ensure the smooth operation of the Clinical Workflow System (CWS) components. Finally, another key enabler for centralized deployment is the system's ability to execute multiple workflows concurrently for different patients. Previous diagrams illustrate this capability, demonstrating that each workflow execution instance is associated with a unique identifier (process ID). As a result, the system can distinguish between multiple active workflow instances

and their corresponding messages.

Consider a scenario where a centralized patient monitoring system, such as Philips PIC iX, is used. In such a setup, the CWS UI component can be deployed within the monitoring system's infrastructure, enabling it to render the states of ongoing workflow executions. Components such as the editor, workflow database, and their corresponding interfaces can be deployed in the cloud to support centralized management further, allowing for remote reconfiguration and flexibility. Since the adaptors and other core components of the CWS are also hosted centrally, it becomes possible to remotely turn specific adaptors on or off and also perform system maintenance efficiently. Finally, leveraging the hospital's network infrastructure allows the CWS to communicate seamlessly with all necessary external systems.

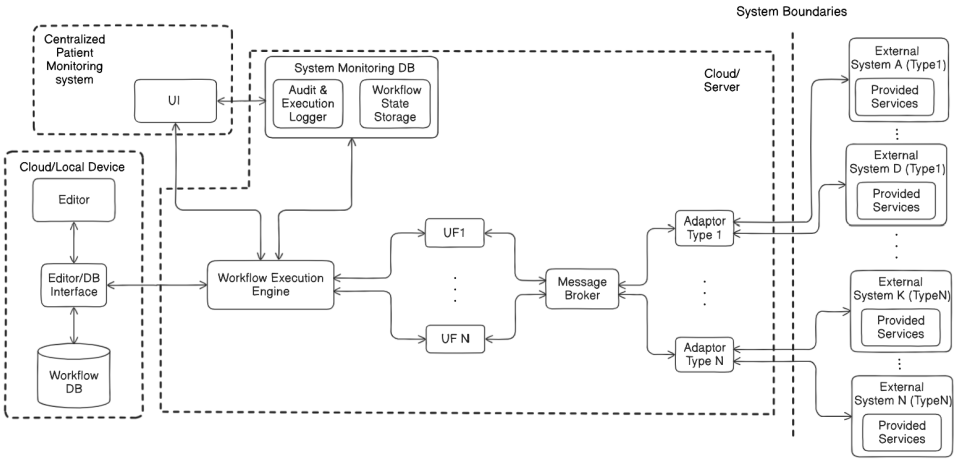



Figure 4.8: Centralized deployment scenario

Now, let us consider a per-patient deployment scenario. In this case, workflows need to be executed on patient-specific devices, such as patient monitors, with a one-to-one relationship between the device and the patient. This can be required in situations where a centralized CWS deployment is not accessible. One example is when the patient is in an ambulance and workflows must begin immediately, before the patient reaches a hospital where a centralized CWS is available. Another example is when the system needs to operate at the edge, close to the patient, due to the absence of central infrastructure. A deployment for such a case is depicted in Figure 4.9. In this scenario, all of the system's core components are deployed on the patient monitor (or similar edge device), allowing the system to function independently for each patient. This setup allows the CWS to integrate naturally into the hospital system by embedding core workflow functionality directly within hospital-owned devices such as patient monitors. Communication with existing hospital infrastructure occurs via the adaptors, which enable seamless interfacing with EMRs, EHRs, and other clinical systems already in use. Thanks to the modular architecture, even in this deployment, workflows can still be reconfigured remotely. This is possible because the reconfiguration components, the editor and workflow database, can be hosted either in the cloud or on a local device. When hosted in the cloud, reconfigured workflows can be deployed to all required edge devices remotely. When hosted on a local device, the updated workflows can be distributed by physically connecting the local device to the target edge devices if a remote network connection is not available.

Finally, consider the case where the central server becomes unavailable during centralized deployment. A mixed deployment strategy could be used to handle this. In this approach, while the main setup is a centralized deployment, a per-patient deployment can also be applied on edge devices (Patient Monitors). The per-patient deployments can be enabled when the central server becomes unavailable. However, this requires additional functionality so that the edge devices can detect the central server's unavailability and always stay updated with the relevant workflow states for the specific patient they are monitoring to continue execution seamlessly. As a first approach,

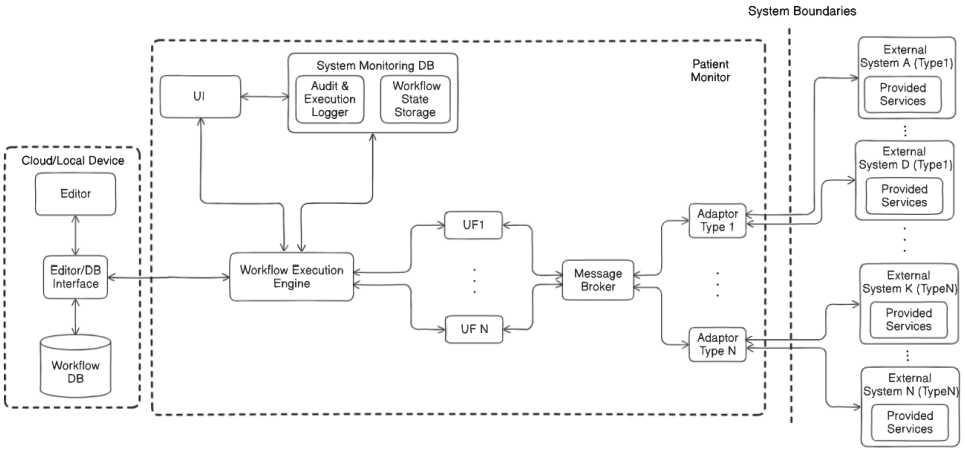



Figure 4.9: Per-patient deployment scenario

the CWS deployment on edge devices can be treated as an external system in the central CWS deployment that continuously updates the patients' workflow states. This way, the per-patient deployments are always aware of the latest workflow states they are interested in and can take over execution from the most recent state if the server goes down. This example demonstrates the current design's flexibility rather than detailing a complete implementation. Also, such a deployment can support cases where workflows have already started running in a patient monitor in an ambulance, for a patient. When the ambulance arrives at the hospital and stable network connectivity exists, the centralized CWS can be updated by the edge deployment in the ambulance's patient monitor and take over the workflow execution. The previous discussion pinpoints the proposed system design capabilities. However, to enable such a sophisticated scenario, dedicated synchronization protocols between different CWS instances must be implemented.

### 4.2.4 Workflow Execution Scenario

After explaining all these aspects of the proposed system, let us look at a high-level explanation of how the system works during workflow execution using the sequence diagram provided in Figure 4.10. So, let us suppose that the sepsis workflow shown in Figure 3.3 needs to be executed because an alert is raised by a patient monitor (*External System PM*) dedicated to specific patient with *patientID*. This alert is intended to trigger the execution of a sepsis clinical workflow. A necessary precondition is that the workflow must already be deployed in the execution engine, as shown in Figure 4.2, to be recognized and accessible for execution. When the PM raises the alert, it sends an appropriate request to its corresponding *adaptor* (*PM Adaptor*) for sepsis workflow trigger, which translates the external request into an internal command compatible with the CWS. The *PM Adaptor* publishes the trigger message to the *message broker*, indicating the corresponding topic and for which patient the workflow needs to be executed. A dedicated UF, *UF Workflow Triggering*, responsible for handling workflow triggers, is subscribed to the relevant topic and receives the message. The *UF Workflow Triggering*, knowing the interface of the *execution engine*, transmits an appropriate message to the workflow *execution engine*, which identifies the workflow name that needs to be executed and proceeds to the execution initiation accordingly.

There are multiple concurrency scenarios when executing the workflow's first four steps, the ones in a parallel arrangement. For example, they can all be executed simultaneously or in sequence, meaning that one is executed after the other throughout time. For clarity in this example, they are shown executed sequentially. So, task *Measure Heart Rate* starts to be executed. Assuming this task is linked to the *UF Measure Heart Rate*, the *execution engine* triggers the UF to execute the wanted functionality. In turn, the UF publishes a message on the topic

*Heart.Rate.Request*, which, through the *message broker*, is received by the subscribed enabled adaptor, *PM Adaptor*, responsible for interacting with the PM. For simplicity, let us assume that only the PM is responsible for carrying out this request in this hospital, so only this adaptor receives the message. The adaptor, by knowing the PM linked to the specific patient, its services and applying the needed interoperability protocol invokes an exposed service by the PM related to heart rate measurement. Once the *PM* sends its response, the *adaptor* translates it and publishes the heart rate measurement on the topic *Heart.Rate.Response*. In this way, *UF Measure Heart Rate*, subscribed to this topic, receives the message and accordingly transmits the retrieved heart rate measurement to the *execution engine* and marks the task as complete. The *execution engine* then updates the workflow state and variables and stores them in the *system monitoring DB* component, which also updates the *UI*. In the same way, the rest of the automated tasks of the workflow are executed, with the execution engine interacting with other UFs that utilize either the PM adaptor or other system's adaptors.

Suppose for that *Evaluate Mental status* task, no external system is used and has been modeled to be executed manually by a clinician via the *UI*. In this case, the *execution engine* directly updates the workflow state in the *system monitoring DB* components, which update the *UI* where the new state and the manual task to be executed is rendered. In this case, clinicians must execute a manual task to evaluate the patient's mental status. When done, they input their judgment into the *UI* and mark the task as complete. The *UI* transmits the input and status of the task to the *execution engine*, and the workflow state is updated accordingly, marking the task as complete. Accordingly, with the tasks executed using UFs, the *execution engine* in the manual tasks, when completed, updates the workflow state in the *system monitoring DB* component, and the *UI* renders the new workflow state. Now, assuming that all the four steps before the condition in Figure 3.3 are executed, the condition component, having all the required inputs, is evaluated, and the correct flow and next tasks are enabled based on the outcome.

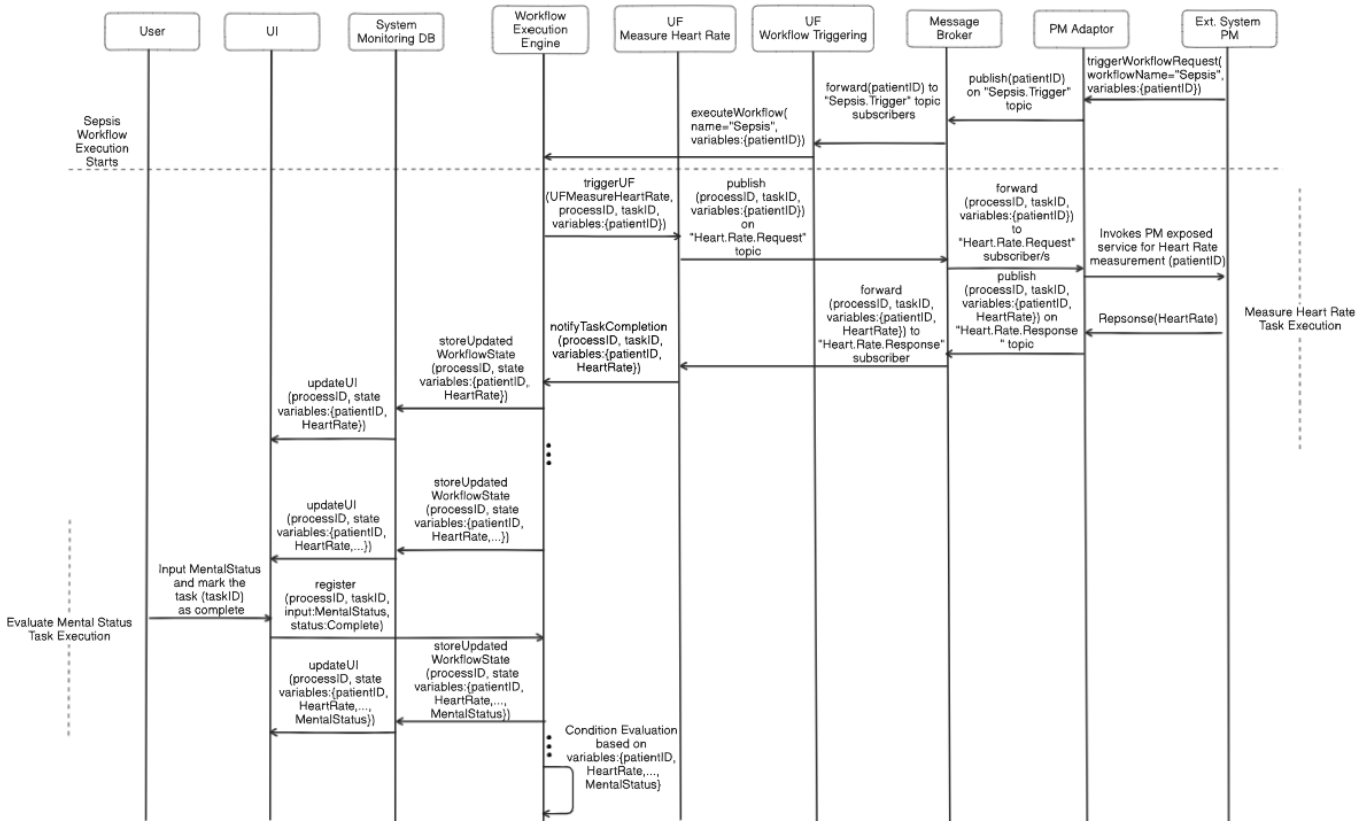



Figure 4.10: Part of Sepsis Workflow Execution Sequence Diagram

This is a simplified example of how the system works and executes a workflow. A discussion point that might be raised from this example could be how the system functions if a workflow, e.g., the sepsis workflow, needs to be executed for five different patients. This scenario is reasonable and can be handled by the system using process identifiers (processID), as shown in the diagram's messages. Thus, workflow states and the generated messages can be differentiated based on this, as well as additional identifiers like taskID, which uniquely identifies to which task the message corresponds, and patientID, which uniquely identifies the patient in workflows. Using a combination of such identifiers, messages can be differentiated for workflow instances, tasks, and patients.

To summarize the capabilities of the proposed system, it has been explained how it addresses all three main requirements presented in chapter 3. Firstly, workflow reconfiguration without code-level changes is handled by leveraging the concept of the execution engine in combination with workflow definitions being expressed and stored in configuration files. These files can be reconfigured using an editor capable of applying specific rules to produce valid workflow definitions that the execution engine can interpret. Additionally, by using UFs, which act as modular blocks of functionality, specific functionality can be linked to tasks within the workflow, enabling easy reconfiguration without needing code re-testing. Secondly, the variability of the CWS environment can be handled through the UFs, the message broker, adaptors, and appropriate interoperability protocols. The message broker decouples the UFs from the adaptors and allows the system to adapt to the enabled adaptors automatically, without any code-level changes. However, new adaptors may be developed when new unforeseen systems types arise. The system's design enables this scenario with minimal cost, as the maintenance and deployment of each adaptor can be done independently without any effect on the rest of the components of the system. Finally, deployment responsibilities for centralized, per-patient, and mixed deployment scenarios have been discussed, showcasing that the system can meet different deployment requirements. Different scenarios have been discussed for each requirement, proving the requirement coverage. In conclusion, the proposed system fully supports the stated requirements. In chapter 5, a proof of concept system implementation for the proposed system design is presented, and the change scenarios that the system should be capable of handling are demonstrated in section 5.6.

## 4.3 Alternatives

As mentioned in the section 4.1, the proposed workflow management systems in the literature strongly align with the proposed system approach that follows the same core approach as this study, using an execution engine, and then workflows are defined in configuration files. However, multiple new components and aspects have been handled in this study, like UFs, the message broker, and adaptors, which handle critical other aspects like modularity, interoperability, and environment variability.

An alternative approach that could be considered for addressing workflow execution and reconfiguration is the data-oriented design. Although no system that needs to reconfigure workflow structures appears to follow this paradigm in the current literature, the data-oriented approach organizes system behavior around the flow and transformation of data objects rather than through explicit orchestration logic [76]. In this paradigm, data acts as the primary driver of execution. Specifically, each transformation step is triggered as soon as input data becomes available, making the system inherently reactive. Such an approach is intuitive in data-centric domains such as ETL pipelines, stream processing, and scientific computing, where traceability of data transformations and simplicity of flow are the main goals [77]. Theoretically, this could benefit workflow execution by enabling declarative modeling of data dependencies. However, the absence of constructs for modeling coordination logic, conditional branching, and human-in-the-loop decisions makes this approach unsuitable for process-driven environments such as clinical decision support and workflow execution systems like the CWS proposed in this study. These limitations reduce its suitability for scenarios where workflow state, contextual decisions, and external interactions must exist.

From another viewpoint, an alternative to the centralized execution engine approach is a decentralized execution model. In this model, instead of a central engine orchestrating workflow execution, the system could provide a set of services, such as UFs, to execute individual workflow tasks. Each patient is represented by an object containing all relevant data, including the workflow definition that needs to be executed and the current state. In this setup, if each UF can interpret the workflow embedded in the patient object and determine the next task to execute, then execution could proceed decentralized. The patient object would be passed from one UF to the next, with each UF updating the workflow state as execution progresses. While this approach reduces reliance on a single orchestrating component, thus eliminating a potential bottleneck, it introduces additional complexity. Each UF needs to incorporate logic for interpreting the workflow, updating the state, and routing the patient object to the appropriate next UF. Due to the extensive literature review conducted throughout this study, the core idea of using an orchestrator, such as an execution engine, has been adopted and decided to be the optimal solution for workflow configurability and system design simplicity. In our case, the term orchestrator refers to the centralized workflow execution engine, which drives workflow execution by triggering UFs, updating the execution state, and determining the next steps. However, in scenarios where increased system complexity is acceptable, the decentralized alternative could be considered, as it eliminates the single point of failure and may enhance fault tolerance.

A related discussion was presented in the previous section on adaptors. One limitation of using adaptors is that if an external system needs to initiate interaction with the CWS (for example, to trigger a sepsis workflow), the external system must be adapted to the CWS to make such requests. This limitation is common in other integration patterns as well, such as the blackboard model. Furthermore, the use of the publish-subscribe pattern ensures decoupling between publishers and subscribers, which is essential for keeping UFs and adaptors independent. Another possible design approach would be to implement all system components to handle multiple interoperability protocols directly. However, this would make the CWS rigid and difficult to evolve when new protocols need to be supported. By adopting the adaptor pattern, the system achieves a clear separation of concerns, which is that interoperability protocols are handled within the adaptors, allowing the core CWS components to remain protocol-independent. In summary, this design choice allows the system to adapt to environmental variability without requiring code-level changes.

Regarding the UFs, their use leads to modular functional blocks that can be reused to simplify workflow definition. This modularity allows testing to be conducted primarily at the UF level after development without requiring additional testing of the implemented logic when workflows are reconfigured, or UFs are reused in different parts of the process. Also, the service-oriented approach helps in case UFs need to be maintained, as this can be done without affecting the rest of the system, leading to an evolvable system thinking. From the workflow editor's perspective, the editor applies specific rules to ensure workflow definitions do not contain syntactic or semantic errors. Finally, through examples, it has been shown that the system's modularity and the treatment of external systems as separate entities enable even complex deployment scenarios, such as hybrid centralized and decentralized configurations, instrumental in cases where the central server fails or the network is unavailable. In conclusion, the current design is considered optimal following this system review, given the requirements.

## 4.4 Trade-offs

This section discusses key trade-offs encountered when designing an evolvable system such as the CWS.

One primary trade-off for designing CWS is centralized versus decentralized workflow orchestration. A centralized approach is based on a single orchestration component to interpret the workflow definition and manage the control flow of the entire process. This offers clear advantages. It provides a global view of the workflow execution instances, simplifies monitoring and auditing, and supports a straightforward way to reconfigure and execute a workflow definition.

However, one limitation of the centralized approach is that the orchestrator becomes a single point of control and, thus, a potential performance bottleneck or failure point. The architecture becomes entirely dependent on a central coordination component, reducing fault tolerance. In contrast, a decentralized orchestration approach distributes the decision-making across individual services, where each component contains enough logic to determine the next step that needs to be executed. This improves resilience. However, it complicates system design, implementation, and debugging. Thus, the centralized approach has been selected for multiple reasons. Firstly, design simplicity and modularity enhance system evolvability by keeping each system's component strictly with a single responsibility. Next, it provides a simple and concrete way to execute a workflow, reconfigure it and redeploy it, and it can provide a complete overview of the workflow execution instances. Finally, the drawback of a single point of failure or performance bottleneck can be overcome by deploying more than one instance of the execution engine, provided hardware resources allow. Thus, the benefits of the centralized approach outweigh the benefits of the decentralized approach.

Another architectural trade-off in the CWS is the deliberate separation of functionality from the workflow configuration definitions by implementing UFs as independent and reusable services. This separation offers several advantages. Each UF encapsulates a specific clinical task functionality, such as measuring heart rate, and can be tested, deployed, and reused independently of the workflow definition and other system components. As a result, when workflow definitions are updated, or workflow tasks are rearranged, the linked UFs in the workflow definition do not require retesting, and this simplifies workflow maintenance and reduces regression risk. It also keeps workflow definitions cleaner and more declarative by delegating functional logic (e.g., clinical measurements or queries) to UFs, while the control logic (e.g., sequencing and branching) remains within the workflow definition. In contrast, an alternative approach could be that inside each automated task in the workflow definition, the external system could be set to be invoked directly, including the functionality of handling responses. This approach provides a simple but not extensible or maintainable solution, while development efforts should take place for each task, a point that can introduce errors and extensive workflow testing. In contrast, using the UFs approach requires a higher upfront implementation effort. UFs must be carefully designed with a single responsibility, message-handling capabilities, and decoupling from adaptors. While this adds complexity during development, it pays off in the long term while it enables workflow reconfiguration without impacting core functionality, a key requirement for CWS evolvability in dynamic clinical environments.

Additionally, the approach used to integrate CWS with external systems presents a trade-off. While adaptors make CWS capable of interacting with external system's exposed services, and decouple the rest of the CWS components of concerns related to interoperability they also introduce maintenance challenges. Adaptors act as intermediaries that make transparent to UFs and the rest of the system components how an external system should be contacted or, from the other side, transform incoming requests/responses to internal system notation. This promotes adaptor reusability across different UFs, creates a clear separation of concerns between system components, and ultimately makes the system adaptable to different clinical environments across hospitals. However, this abstraction comes at the cost of increased development and maintenance effort. Each adaptor linked to a type of external systems must be built, tested, and maintained separately, and any changes in external system interfaces may require corresponding updates in the adaptors logic. As the number of external systems grows, the cumulative maintenance overhead becomes significant, requiring ongoing coordination with external stakeholders and version management. Despite these shortcomings, the adaptability provided by the adaptor approach makes it suitable for the CWS.

Another architectural decision involves choosing where to place the logic for mapping patients to their corresponding external systems and in general mapping of which external system should be invoked for each request. One option is to embed this mapping within the message broker, enabling it to route messages only to the appropriate adaptors based on patient identifiers. This approach could optimize message delivery, ensuring that only relevant adaptors receive and process each message. However, it would require the message broker to maintain and continuously update

dynamic patient-to-system mappings, significantly increasing its complexity. Alternatively, the chosen approach is to delegate the mapping responsibility to the adaptors. In this design, all adaptors subscribed to a given topic receive the message but apply their internal logic to determine whether the message pertains to a patient they are configured to handle. While this may result in more messages being received by adaptors, the design preserves the message broker's simplicity. It also promotes modularity by localizing system-specific knowledge within the relevant adaptor. Thus, the added message filtering cost at the adaptor level is considered acceptable in exchange for system maintainability and modularity.

Another key trade-off is between achieving modularity and independence versus managing increased component implementation, deployment, and infrastructure overhead. This becomes relevant when deciding how UFs and adaptors should be designed and deployed. By making the choice to adopt a highly modular architecture leads to the result that components can be developed, tested, deployed, and updated independently without causing system-wide downtime and requiring retesting of unrelated components. This promotes flexibility, scalability, and maintainability, which are all crucial for a system designed to evolve continuously. In the CWS architecture proposal, UFs such as heart rate or temperature measurement are implemented, tested, and deployed as separate services to the execution engine. This allows for isolated updates and scaling. However, modularity also introduces significant maintenance and operational overhead. As the number of components grows, the number of deployed services grows as well. An alternative approach, grouping multiple UFs into a single service, would reduce infrastructure costs and simplify administration. However, this would compromise modularity and increase the risk of unintended side effects when maintaining individual functions. Updates would require redeploying the entire grouped service, which reduces agility. Ultimately, although modularity increases overhead, it better supports the goal of workflow reconfiguration and, in general, system evolvability, which is essential for adapting to evolving clinical workflows.

Finally, while the modular architecture of the CWS significantly improves maintainability, testability, and simplicity, it also introduces challenges during operation time in case of malfunctions. Each UF and adaptor is designed to be an independent service. This allows for efficient unit and integration testing, enabling bugs to be identified and fixed at the component level without affecting the rest of the system. However, the complete execution of a clinical workflow now spans multiple layers, including the execution engine and interface, UFs, message broker, adaptors, and external systems. As a result, when issues arise, identifying the root cause requires tracing the interaction across all these distributed components. Debugging is no longer a single-layer activity but a multi-layer investigation involving logs, message payloads, and other sources that can reveal the root cause. Therefore, while the system gains in modularity and testability, it also incurs traceability and debugging complexity costs.

All the mentioned trade offs are summarized concisely in Table 4.2. In conclusion, while the architectural decisions behind the CWS introduce some additional development and operational complexity, the benefits in flexibility, maintainability, and adaptability strongly outweigh these trade-offs. The proposed system is well-suited to handle evolving clinical workflows and heterogeneous hospital environments.

| **Design Decision (Gain)** | **Corresponding Trade-off** |
|---|---|
| Centralized orchestration for workflow execution, control and monitoring | Single point of failure and potential performance bottleneck |
| Functional separation of workflow definitions via reusable UFs | Higher initial design and implementation effort |
| Use of adaptors for external system integration | Potential maintenance overhead as external interfaces evolve |
| Delegation of patient-to-system mapping logic to adaptors (clear separation of concerns) | Message filtering load on adaptors and potential for redundant message processing |
| Service modularity enabling independent development and scaling | Initial implementation (and deployment resources) overhead |
| Modular and testable components with single responsibility | Multi-layer tracing/debugging complexity across distributed services |

Table 4.2: Summary of key architectural trade-offs in the CWS design

# Chapter 5

# Proof of Concept

In this chapter, a proof-of-concept implementation of the proposed system is presented. The goal of this implementation is to demonstrate the system's capabilities and show that it can practically satisfy the outlined requirements. While multiple implementation alternatives exist, the implementation of the components will be selected to maximize development efficiency, align with the project timeline, and effectively demonstrate the system's core capabilities. To maintain focus on the goal of the proof of concept being to validate the system's design capabilities, the detailed proof of concept implementation aspects are provided in the appendix.

To begin with, one of the first steps in constructing such a proof of concept is selecting an appropriate modeling language capable of expressing a (clinical) workflow, as described in the previous chapters. This includes representing tasks, transitions, conditions, events, actors, and pre/postconditions while also supporting the integration of UFs. The modeling language must be supported by an editor that enforces workflow definition rules, an execution engine that is capable of interpreting and executing the defined workflows, and a user interface that visualizes the running workflows state and manual tasks to be executed by the clinicians. Therefore, this choice is crucial to the overall system implementation considering the project's boundaries.

## 5.1 Workflow Modeling Language

To support the selection of the appropriate workflow modeling language, a high-level comparison of alternatives, including BPMN, CMMN, AWS Step Functions, and others, is provided in Appendix A. The analysis shows that BPMN and CMMN are the most suitable for clinical workflows due to their mature tooling, simple notation, and support for human-in-the-loop tasks. Among them, BPMN was selected for this proof of concept because it offers more intuitive modeling for structured processes and better tool support.

BPMN and CMMN are standardized and widely used modeling languages and provide mature, user-friendly tooling, including graphical editors, execution engines, and workflow monitoring interfaces. Also, both of them support the required workflow components. Between the two, BPMN offers a more intuitive modeling experience for structured processes, broader adoption, extensive documentation, and strong community and enterprise support. Furthermore, BPMN tools like Camunda provide seamless integration of both automated and human tasks, form-based user interactions, and workflow state tracking features that are directly aligned with the needs of clinical workflow modeling. Although the system architecture is designed to support both BPMN and CMMN, BPMN has been selected for the current proof of concept because it enables modeling of strictly defined workflows processes, a feature that represents the workflows that are reviewed in the current study and also promote clinical processes standardization. BPMN also enables faster development, more transparent communication between stakeholders, and more immediate validation of the proposed system's functionality.

An important point to highlight is that while BPMN is selected for the proof of concept

implementation of the current study to demonstrate the feasibility of the proposed system, this choice does not imply that the other alternatives are not supported by the proposed system design. CMMN remains a fully supported and viable alternative, and deeper clinical or organizational requirements could be considered in order to use it in the final system implementation. For the other workflow modeling approaches, such as AWS Step Functions and Netflix Conductor, initial analysis shows that they offer the essential components, and these strongly align with the proposed system design. However, a dedicated evaluation would be necessary to assess their suitability for clinical use cases. Another point that must be considered when adopting a specific modeling language is whether one aligns better with the hospital policies and standards. For example, if there is a requirement for the system to align closely with the AWS ecosystem, then AWS Step Functions might be the preferred option. In any case, while this detailed information is not available for the scope of this project and thus a final decision cannot be taken at this point, it must be pointed out that the system design is general enough to accommodate any of these alternatives and this underscores the value of the system's proposal.

The current study has selected BPMN as the modeling language for clinical workflow definitions. BPMN is supported by the Camunda Modeler, a graphical editor that helps users design BPMN-compliant workflows and validates them for structural correctness. Camunda also provides the Zeebe client, which serves as a communication interface to the Zeebe workflow engine, the execution engine responsible for executing workflows. In addition, Camunda offers integrated tools for workflows state monitoring (Camunda Operate, Elasticsearch) and a user interface for task management (Camunda Tasklist). The former is essential for tracking the workflow state, and the latter for executing manual workflow tasks. In summary, BPMN is well supported by the needed components for modeling and execution.

## 5.2 BPMN Expressive Power

BPMN provides all the necessary components required to implement a clinical workflow. The core elements include tasks (manual and automated), transitions, conditions, and events. Automated tasks in BPMN can be linked to UFs services triggered when the task is executed. As shown in Table A.1, BPMN is a standardized workflow modeling language widely adopted for workflow development and execution. Also based on experts opinion, sepsis clinical workflow is a representative example of clinical workflows. Thus, attempting to implement the sepsis workflow, shown in the previous chapters, using BPMN provides concrete evidence that BPMN can model clinical workflows whose elements and structure have been extensively analyzed.

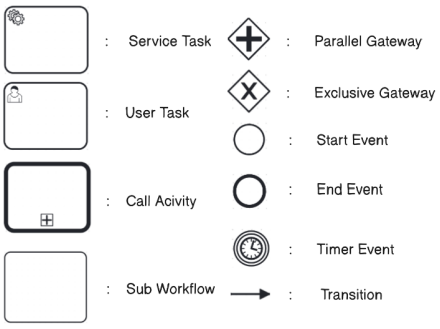



Figure 5.1: BPMN Notation

The notation used in the sepsis workflow expressed in BPMN is illustrated in Figure 5.1. In this context, a service task refers to a task executed by a UF. A user task is rendered in the UI as a form, which the clinician must complete manually and then mark as done. A call activity represents a reference to an external workflow, which is triggered once the call activity is activated.

A sub-workflow serves a similar purpose but is modeled directly within the main workflow. The parallel gateway functions as a token splitter and joiner, allowing tasks to run in parallel and then synchronizing them. The exclusive gateway enables conditional flow control, activating only one outgoing path based on a defined condition. Start and end events define where the workflow begins and terminates. Finally, timer events can be attached to tasks, and counting can begin when the task is activated. If the specified time elapses before completion, the timer triggers and reroutes the token along an alternate path, preventing the original task from executing. This mechanism is useful for implementing timeouts or delays.

To ensure that BPMN fulfills all required clinical workflow features, the following section provides an in-depth mapping between the sepsis guideline and its BPMN implementation. The BPMN implementation shown in Figure 5.3 is derived from the proposed sepsis guideline illustrated in Figure 1.2, using the detailed BPMN notation explained earlier. Let us analyze this example. At the beginning, a start event is responsible for initiating the workflow upon being triggered. Once triggered, the workflow begins execution. As an example of a precondition, it has been specified that the workflow cannot proceed unless a patient ID has been set. If the patient ID is missing from the workflow trigger request, an appropriate action is taken to prompt the clinician to provide it. This is just one option. Alternatively, the workflow could include a service task linked to an appropriate UF to retrieve or generate the patient ID from an external system based on other given variables. Similarly, postconditions can also be implemented. As can be seen, the precondition logic has been encapsulated within a sub-process element to make it easier to distinguish from the rest of the workflow. This precondition governs the execution of the entire workflow, meaning the workflow cannot start unless the patient ID is present. Suppose a precondition or postcondition is meant to apply to a specific task rather than the entire workflow. In that case, it must be placed immediately before or after that task.

The truth of the precondition is established as follows. The start event initiates the sub-process, which then reaches an exclusive gateway. If the patient ID has already been provided when the workflow is triggered, the condition is met, and the path directly leads to the end event which marks the precondition sub-process as complete. Otherwise, the user task *Insert Patient ID* is triggered, requiring a clinician to enter the patient ID. In either case, when the precondition sub-process completes, the patient ID is guaranteed to be present in the workflow context. This example demonstrates that BPMN supports events, pre/postconditions, manual tasks, and conditional transitions.

Adjacent to this, a parallel gateway indicates that the following tasks should be executed independently of each other. The upper three tasks are service tasks, whereas the fourth is a user task. The service tasks are linked to UFs and are executed by external systems, and the user tasks need to be executed by the hospital staff. For example, as depicted in Figure 5.2, the *Measure Heart Rate* service task is associated with the *measureHeartRate* job type, which corresponds to the name of an implemented UF service. When this BPMN task is executed, it creates a job with the corresponding job type. The UFs with the corresponding job type, then undertakes the job, executes it appropriately, and returns the result back to the BPMN task once completed. For user tasks, JSON forms can be created and rendered in the UI to allow users to input data.

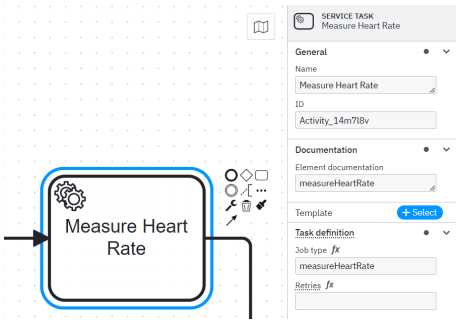



Figure 5.2: UF linkage to BPMN service task

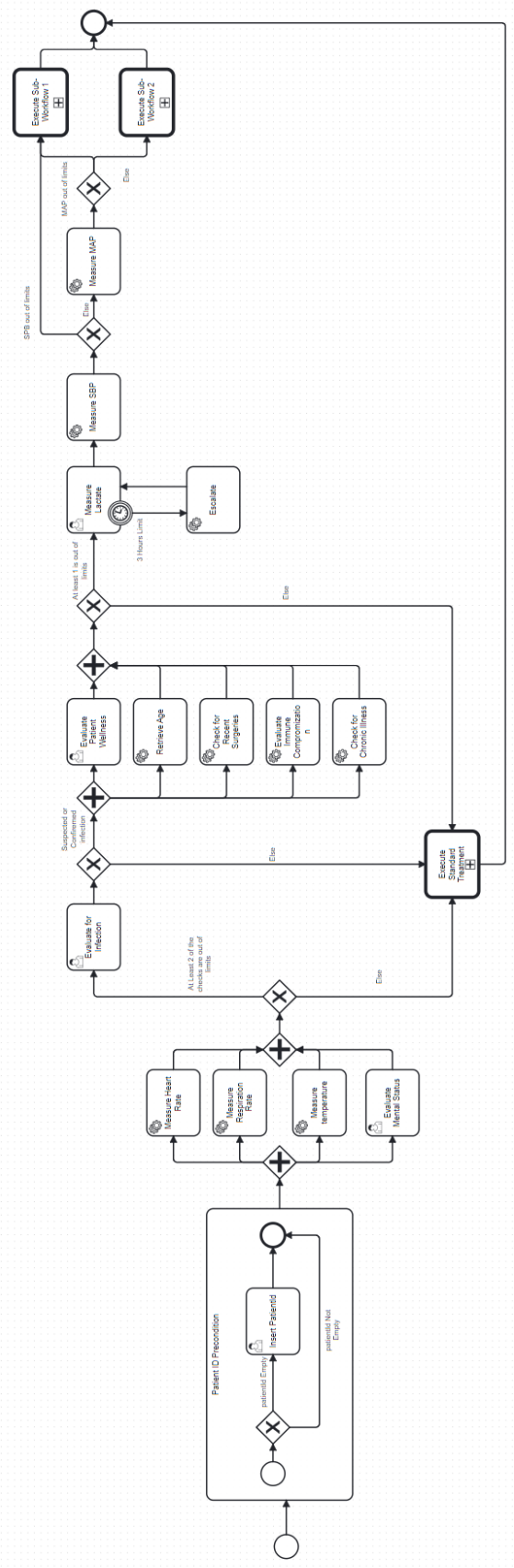



Figure 5.3: Sepsis Workflow implementation using BPMN

Once all tasks within the parallel gateway section are completed, the execution proceeds to a condition evaluation that checks whether more than two checks are deemed abnormal, enabling the appropriate outgoing transition. The conditions are implemented in a configurable way, as shown in Figure 5.4, using the workflow defined variables and there is one for each outgoing transition of the exclusive gateway. In the same manner, the rest of the workflow is constructed using conditions, transitions, events, manual tasks, service tasks. This approach effectively captures the intended functionality within the BPMN workflow definition, which is ultimately stored as an XML configuration file.

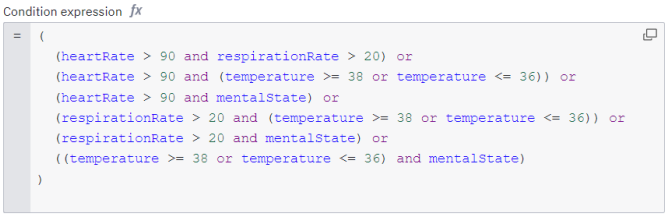



Figure 5.4: Condition Expression Example

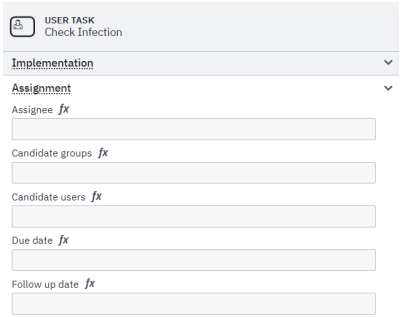



Figure 5.5: Users/Roles/Actors Assignment in a User Task

Finally, three key elements in the workflow implementation should be mentioned. First, there are three call activities *Execute Standard Treatment*, *Execute Sub-Workflow 1*, and *Execute Sub-Workflow 2*. When triggered, each initiates the execution of a linked BPMN workflow. Once the linked workflow is complete, the call activity is marked as complete. This mechanism enables the reuse of related workflows without duplicating logic or overcomplicating the main workflow. Second, a timer boundary event has been added to the *Measure Lactate* task. This timer limits how long the task can remain active. In this case, lactate must be measured within three hours. If the task is not completed within this time frame, the timer triggers an escalation to ensure timely clinical intervention, and the *Measure Lactate* task is activated once again for immediate execution. Finally, as shown in Figure 5.5, actors can be assigned to user tasks either by explicitly specifying a user or by defining a group of users eligible to perform the task.

Given these capabilities, and the fact that BPMN supports all essential workflow constructs such as conditions, transitions, events, service calls for executing sub-workflow, and both manual and automated tasks, it is evident that BPMN provides sufficient expressive power for modeling and executing clinical workflows. By leveraging these core elements, clinical processes can be accurately represented, reused, and maintained within a standardized and executable BPMN format.

## 5.3 Design-to-Implementation Mapping

The proof of concept system implementation architecture is depicted in Figure 5.6. In this diagram the directed arrows give information about the components interaction. The following section maps the components of the proposed system architecture to their corresponding elements in the system implementation diagram.

As shown on the left side, the *Camunda Modeler* in the system implementation corresponds to the *Editor* in the proposed system design. The *Camunda Modeler* provides all the necessary tools for the users to reconfigure BPMN workflows as needed and validate their syntax. Regarding workflow semantics, no validation is provided by the specific editor, so this functionality should be integrated into the editor component in a future implementation. In this implementation, there is no separate editor interface shown connecting to the *Modeler* to the *Zeebe Cluster* or the *database*

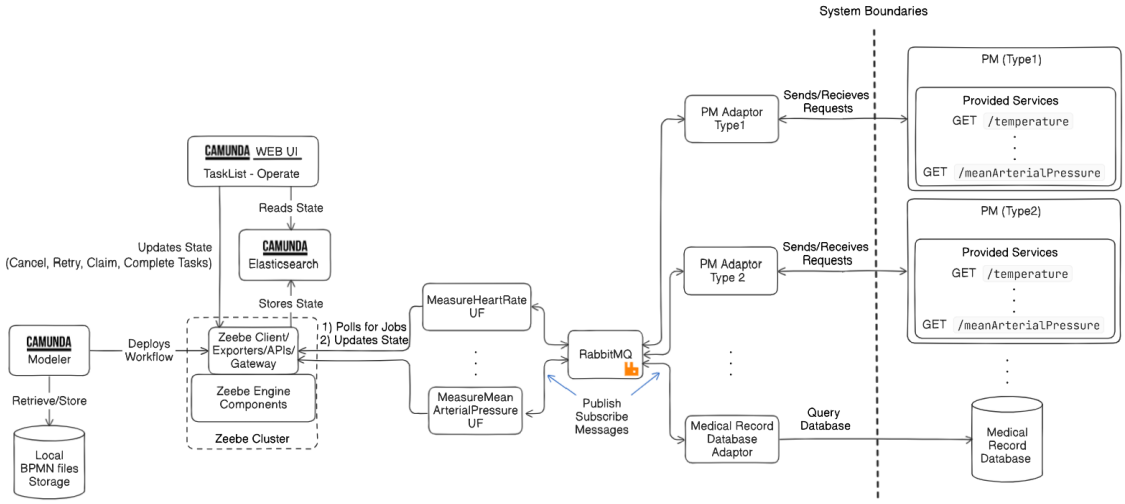



Figure 5.6: CWS Proof-of-Concept Implementation

as this is integrated into the *Modeler*. Additionally, no database was used to store the workflow definitions; instead, for the sake of the proof of concept, the BPMN files are saved locally on the machine where the *Camunda Modeler* is running.

The *Workflow Execution Engine* in the proposed architecture maps to the *Zeebe Cluster* in the actual implementation. In more detail, this cluster contains the *Zeebe Engine* and a set of interfaces used to communicate with different components. The cluster is responsible for executing workflows that are deployed from the *Camunda Modeler* via the *Zeebe Gateway*, which exposes gRPC or REST endpoints. The *Zeebe Client* is used by external workers, a.k.a. UFs, to poll the *Zeebe Engine* for jobs (created by the BPMN tasks during execution for the linked UF), execute the required business logic, and then report the results back. Meanwhile, the *Exporter* is responsible for storing workflow execution data in *Camunda's Elasticsearch*, which is used by the *UI*, in this case *Camunda Tasklist* and *Operate*, to render manual tasks and display workflow instance states. Additionally, *Camunda Tasklist* interacts with the *Zeebe Engine* through internal service *APIs*. These *APIs* allow the *UI* to perform actions such as claiming, inserting input, or completing manual tasks, which also update the workflow state. These components collectively form the *Zeebe Cluster*, which fulfills the role of the execution engine in the proposed architecture.

In the actual implementation, the *System Monitoring DB* component from the proposed design corresponds to *Camunda Elasticsearch*, which stores workflow state and execution data. On the other hand, the *UI* component in the proposed architecture is reflected through three Camunda interfaces. The first is *Tasklist*, which renders manual tasks and provides a user-friendly interface for user input. The second is *Operate*, which allows users to view active workflow instances and monitor their current state. The third, *Optimize*, can optionally be included to generate reports on how efficiently workflows are executed. This third component has not been integrated into the current implementation, as it is not essential for demonstrating the proof of concept. However, it can be added later if needed without requiring system architecture changes.

Regarding the adaptors and the external systems, in the implemented system it has been assumed that in total there are three external systems types, two different patient monitors types and one database simulating EMR resulting in three corresponding adaptor types. Considering the remaining components, the mapping is clear as the UFs and message broker (*RabbitMQ*) are represented in the proposed design just as they appear in the system implementation diagram. An interesting point to observe is that the proposed architecture aligns closely with the implementation, with all components and their interactions matching the original design.

## 5.4 Components Implementation

This section presents the main implementation concepts of the system components so that in the next section, the behavior and capabilities of the implemented system can be explained using sequence diagrams similar to those constructed to describe the proposed system design behavior.

All system components, including the Zeebe Engine, Tasklist, Operate, Unit Functions (UFs), adaptors, and message broker (RabbitMQ) have been implemented and deployed as independent containerized services. All Camunda-related components are used as provided by Camunda. The detailed implementation of all the components can be found in Appendix B. For this proof-of-concept, the necessary UFs for the sepsis workflow execution have been implemented. Accordingly, it has been assumed that the environment where the CWS will be deployed includes two types of PMs; therefore, simulators for these PMs and a database representing an EMR have been developed as well as their corresponding adaptors.

UFs and adaptors communicate asynchronously via RabbitMQ using topic-based routing, while external systems simulate real-world sources such as PMs and EMR. This section uses certain messaging-related terms (e.g., RabbitMQ exchange, binding and routing keys and queues). In short, a RabbitMQ exchange acts like the message broker and routes messages to consumers' queues based on the messages' routing key, which acts like a topic. A binding key defines which messages (by routing key) a consumer's queue should receive. These are defined in detail in Appendix B. Each component plays a specific role in workflow execution, contributing to the system's ability to support reconfiguration, integration with various clinical environments, and flexible deployment options.

## 5.5 System Proof-of-Concept Behavior

Now that all the components of the proof of concept have been explained let's dive into an execution scenario of the sepsis workflow, shown in Figure 5.3, to demonstrate how the entire system works, reconfigure the workflow and execute it. Firstly, during the initialization phase of the system, all the containers of the components of the system start to be running. At this point, the correct adaptors containers are enabled to adapt to the available external systems. Once the UFs and adaptors services start running, they notify RabbitMQ exchange of the topics they are interested in. This is depicted in Figure 5.7.

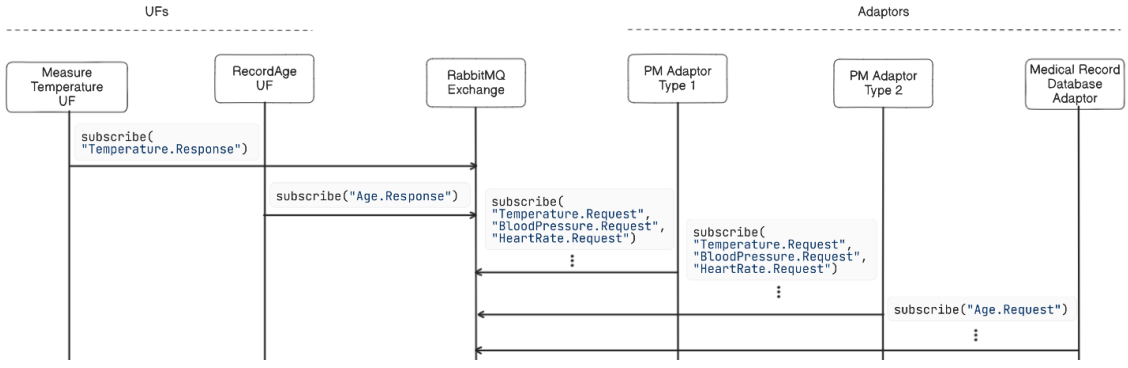



Figure 5.7: RabbitMQ Initialization

After all the components containers have started running and the RabbitMQ initialized, the system is ready to function properly. Let us begin by showing how the reconfiguration occurs in the implemented system. In Figure 5.8, it is shown that the implementation of the system enables exactly the functionality proposed in the architecture proposal. Upon user request, the modeler retrieves the workflow definition from the local storage (disk), applies the allowed actions (will be shown in detail in section 5.6), validation takes place, the user, saves it, and then deploys it

to the Execution Engine with a specific ID, which will be used when the workflow needs to be executed so that the Zeebe Engine can locate it accordingly. Finally, in the above diagram, the components involved and their mapping with the architectural design components of the current study are provided. For example, the editor, combined with its interface, is mapped in the proof of concept implementation to the Camunda Modeler.

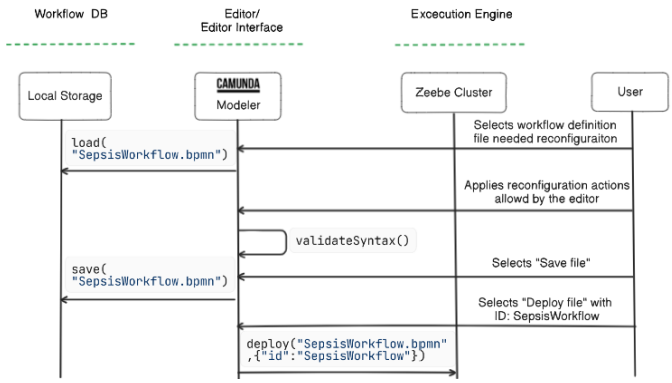



Figure 5.8: Workflow Definition Reconfiguration

Now, let us suppose that the sepsis workflow needs to be executed. For this scenario, we assume that the patient monitor triggers the workflow execution to detect whether the patient is in a septic condition. In Figure 5.9, it is depicted how this is done. Firstly, the PM sends the request to the PM Adaptor, which translates this request into CWS internal functionality. Therefore, it publishes a message containing the patient ID on the *Sepsis.execution* routing key. The exchange receives the message, which checks which consumers have binding keys that match the message's routing key to their queues. Thus, it finds that the *WorkflowsTrigger UF* has a queue named *Sepsis.execution.queue*, which is bound with the specific key, and the message is forwarded to this queue to be processed by the UF. The UF, in turn, triggers the workflow with ID *SepsisWorkflow* that is known by the Zeebe Cluster.

As has been explained, UFs are services that can be used by the BPMN tasks to execute the task. In this case, this UF is not used by the BPMN tasks to execute jobs but is used from the other direction (adaptors) to trigger the workflow execution. In this case, this UF acts as a helper component to trigger workflow execution in the Zeebe Cluster when such requests come from external systems.

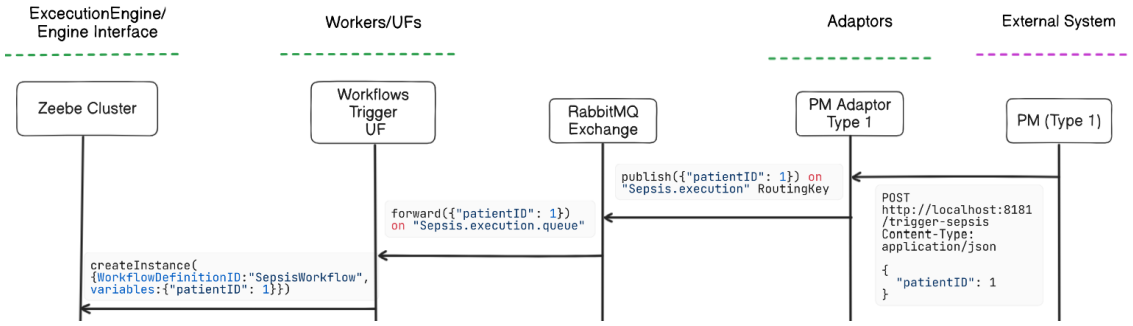



Figure 5.9: Sepsis Workflow Execution Triggering

Now, the workflow execution is orchestrated by the Zeebe Cluster. After the workflow execution has been triggered, the Zeebe Cluster executes all the tasks defined in the workflow definition. But how does the implemented system execute each task? To explain this, let us first define some notions. In the BPMN workflow definition, tasks are defined. These tasks represent work to be

carried out. There are multiple types of tasks, but for this proof of concept, only User Tasks and Service Tasks are used. The first requires the user to provide manual input for completion, while the second uses an implemented service to complete the task automatically. A User Task results in a human-interactive task visible in the Camunda Tasklist, identified by a taskId. In contrast, when a Service Task (such as measuring temperature) is activated, the Zeebe Cluster creates a job identified by a job type, which is then handled by a worker. This job represents the automated execution of the task. To execute these jobs, the UF services come into play and they are the components in the system responsible for processing and completing the jobs.

Having said that, in Figure 5.10 it is shown how the service task *Measure Heart Rate* is executed in the CWS proof of concept implementation. Once the execution has reached the *Measure Heart Rate* task in the BPMN definition, the Zeebe Cluster creates a job for the UF that has been linked to the respective BPMN task. The link has been created via the job type defined in the BPMN task. For its execution, the *Measure Heart Rate UF* polls periodically the Zeebe Cluster to identify if there is any job matching their defined job type. In this way, the *Measure Heart Rate UF* identifies that there is a job with jobKey 22517, which globally specifies uniquely this task in all workflow execution instances and contains, in addition, the variable patientID to specify for which patient this job needs to be executed. As jobKey is globally unique there is no need to use processID to specify to which workflow instance this job belongs. In turn, *Check HeartRate UF* publishes a message on the respective routing key. In the message, the jobKey and the patientID are encapsulated in order be identified by the recipient for which patient the job needs to be executed and also to which job it is referring to, which will be needed for the response message. The message is forwarded in the same way as explained before to the correct adaptor's queue, which is the PM Adaptor Type 1. The PM Adaptor now sends a request to the exposed service of PM (Type 1), which is expected to be linked to the patient with ID 1, to retrieve the heart rate. The PM Adaptor translates the response and creates the message to be published on the *HeartRate.response* routing key, which ends up on the *Measure Heart Rate UF*. Now, the UF knowing the jobKey, the job is marked as complete, and the variable of *heartRate* is added to the workflow state. In the same way, interactions with other UFs and external systems are done as depicted in Figure 5.11.

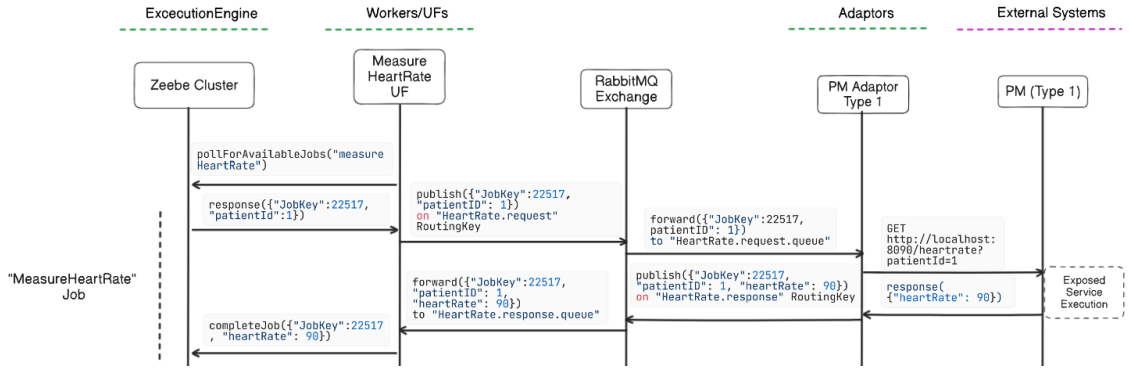



Figure 5.10: Measure Heart Rate Task Execution

It would be beneficial, at this point, to make clear that UFs are subscribed to unique routing keys that are only related to its functionality. For example, *Measure Heart Rate UF* will only be subscribed to *HeartRate.response* routing key, and no other UF will be subscribed to this topic. This ensures that response from adaptors will end up only in the correct UFs to complete the ongoing job execution. On the other hand, adaptors are subscribed to all related topics with the corresponding external system's functionality.

Finally, the execution of the *Evaluate Mental Status* user task is depicted in Figure 5.12. When

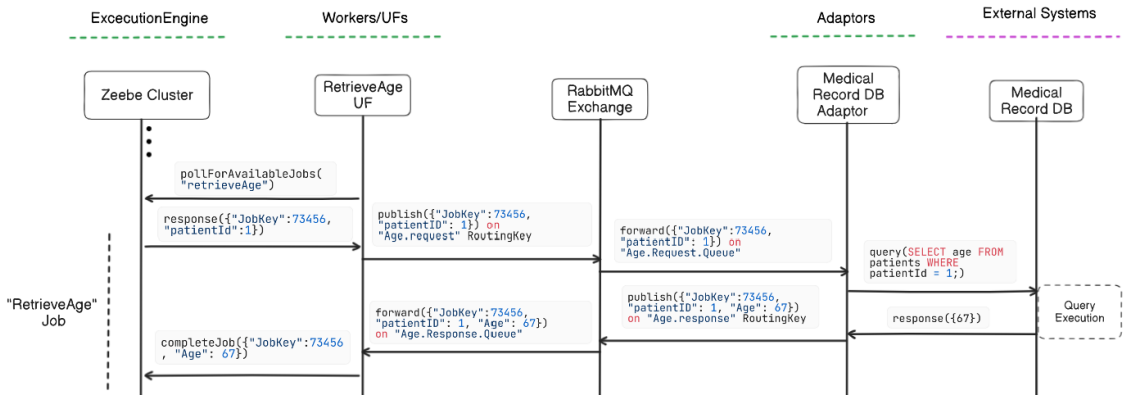



Figure 5.11: Retrieve Age Task Execution

the workflow execution reaches a user task, the Zeebe Cluster updates the workflow state, as it does each time it changes, stores it in Elasticsearch, and, finally, the two UI components (Operate and Tasklist) poll Elasticsearch. Tasklist polls it to get any newly available user task to render and enable the user to execute it. Operate polls it to render the new workflow state and variables. Now, the user can claim the rendered task and provide input manually. When the input is given (MentalStateAltered: false) and the user marks the task as complete, this information is sent to the Zeebe Cluster, which adds the new variable into the workflow state and again updates the workflow state in Elasticsearch, and thus the updated state is rendered in Operate. When any task is completed and the Zeebe Cluster updates Elasticsearch with the new workflow state, it essentially marks the BPMN task as completed, adds any new variables to the workflow state, and stores additional logs. Based on the provided explanation, the rest of the workflow can be executed.

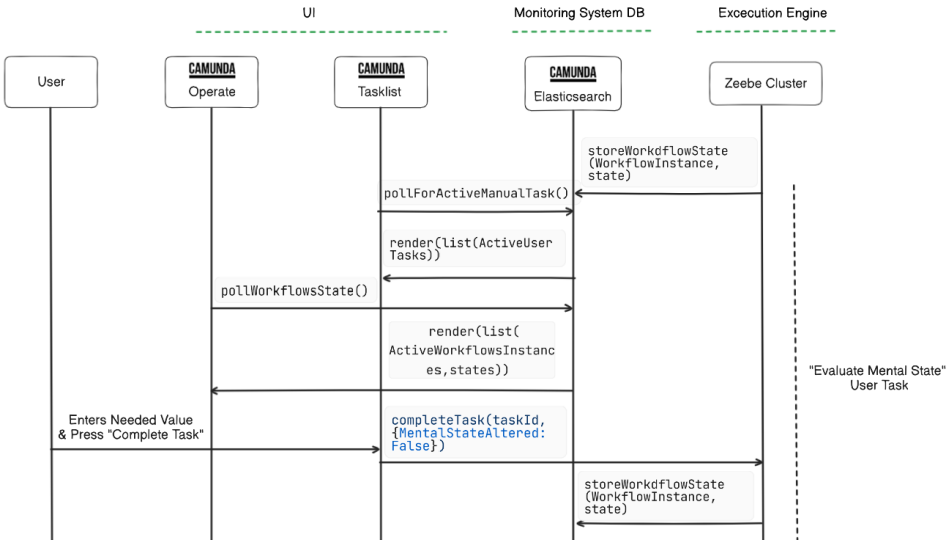



Figure 5.12: Manual Task Execution

To summarize, the provided diagrams demonstrate that the proof-of-concept system implementation is capable of satisfying the project requirements. Specifically, it is capable of reconfiguring workflows without requiring code-level changes. Secondly, it can adapt to varying clinical environments by dynamically enabling the appropriate adaptors containers, without affecting other components of the system. Finally, it supports different deployment options, as all components

are independently deployable and can be hosted on the desired hardware. While the provided examples confirm the intended system behavior, the following section goes one step further and it systematically validates each of the requirements defined in the change scenarios. Each requirement will be demonstrated in a reproducible manner, providing concrete evidence that the system can support dynamic reconfiguration, adaptability, and modular deployment and all without code modifications.

## 5.6 Change Scenarios Demonstration - Results

Having provided explanation of the implemented system components and behavior, let's focus on demonstrating all the change scenarios that have been defined as system requirements in section 3.2. These scenarios will serve as reference examples, and adapted versions of them will be used to demonstrate similar actions within the Sepsis workflow shown in Figure 5.3.

Assuming that Figure 5.3 represents the initial implementation of the workflow to be reconfigured, the change scenario demonstration will begin with the workflow reconfigurability scenarios. Specifically, we will start with task reconfiguration actions, including addition, removal, and reordering.

### 5.6.1 Task Addition

The first scenario is about task addition. So, let us assume that a task *Measure SBP*, which measures the patient's systolic blood pressure, needs to be added after *Measure Heart Rate* to meet hospital-specific requirements. The following images show how this can be done using the Camunda Modeler. Firstly, the sepsis workflow definition file has been loaded from local storage and opened in the modeler. Figure 5.13 illustrates the part of the Sepsis workflow that will be reconfigured. In Figure 5.14, the different task types that can be added to the workflow are shown. For this demonstration, a Service Task is needed, which is a task executed automatically by a UF. In the same way, a User Task could be added respectively. By selecting the Service Task in the *Create element* menu, the user can place it in the desired location in the workflow, as shown in Figure 5.15 and Figure 5.16. Then, in Figure 5.17, the core properties of the service task need to be configured, such as the *Name* and the *Job Type*, which corresponds to the name of the UF that will be triggered when the service task is executed in the BPMN workflow.

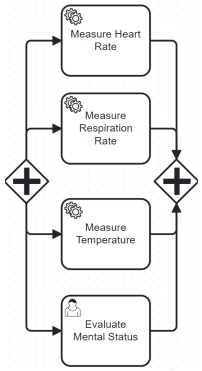



Figure 5.13: Sepsis Workflow part to be reconfigured

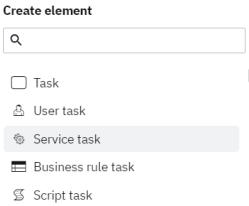



Figure 5.14: Create Element Menu

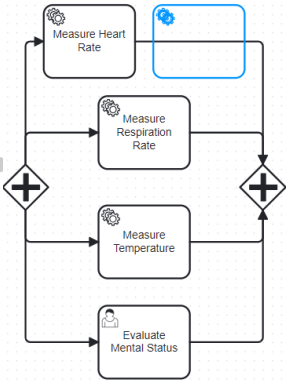



Figure 5.15: Service Task Addition

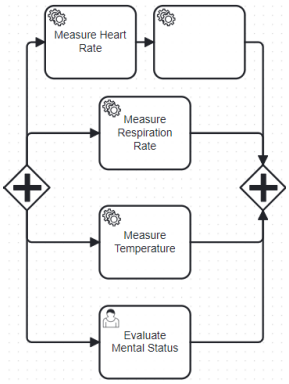



Figure 5.16: Service Task Placement

After configuring the new service task properties, the editor provides error checking, as shown in Figure 5.19, to ensure that no BPMN rules violations exist in the workflow definition. However, the clinical logic that the workflow will execute cannot be tested this way; this only verifies that

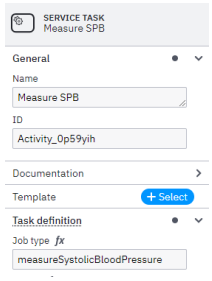



Figure 5.17: Service Task Properties

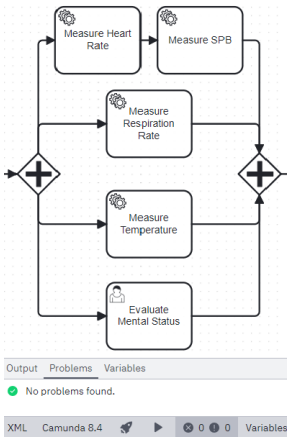



Figure 5.18: After reconfiguration Error Checking takes place

```
<bpmn:serviceTask id="Activity_12wsd55" name="Measure SBP">
  <bpmn:extensionElements>
    <zeebe:taskDefinition type="checkSystolicBloodPressure" />
  </bpmn:extensionElements>
  <bpmn:incoming>Flow_03c80ai</bpmn:incoming>
  <bpmn:outgoing>Flow_102hu39</bpmn:outgoing>
</bpmn:serviceTask>
```

Figure 5.19: XML Configuration File Element

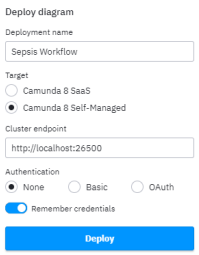



Figure 5.20: Sepsis Workflow Deployment

BPMN rules have been followed correctly. The editor, in future implementation, can also be augmented with semantic rules provided by health authorities (e.g., WHO), which can include validation of clinical logic and pre- and postconditions as well. Once the workflow passes this validation, it can be saved/updated in the XML configuration file. In Figure 5.19, the added element is represented as XML code specifying the properties that have been set such as the task type, name, the job type, and the incoming and outgoing transitions. Lastly, after the workflow has been saved, the modeler is used to deploy the reconfigured sepsis workflow into the execution engine, as shown in Figure 5.20, where the *Cluster endpoint* refers to the *Zeebe Cluster* endpoint.

Error checking of the reconfigured workflow happens after every change in the configuration file in any of the following workflow change scenarios. Similarly, the XML representation of the BPMN is updated each time the workflow is saved. After each reconfiguration, the updated workflow must be deployed again to the execution engine, so that it is ready for execution by the *Zeebe Cluster*.

### 5.6.2 Task Removal

In the same way, task removal will be demonstrated. Based on the generated scenario, let us assume that *Measure SBP* is deemed unnecessary according to the hospital protocols for sepsis condition identification and treatment, so the workflow must be reconfigured. As shown in Figure 5.21, *Measure SBP* is selected, and the possible actions for the specific element are shown in the pop-up menu on the right. By selecting the bin icon, the element can be removed, resulting in Figure 5.22, where the task has been deleted, and the outgoing transition from *Measure Heart Rate* has been reconfigured to target the parallel gateway, which was previously the target of the removed task. Once saved, the corresponding reconfiguration is then reflected in the XML representation of the BPMN, and error checking and deployment can be performed the same way as already described.

### 5.6.3 Task Reordering

In the same way, let us demonstrate how task reordering is supported by the system's editor, Camunda Modeler. Assume that the *Measure SBP* and *Measure Heart Rate* tasks need to be reordered. In Figure 5.23, the initial task arrangement is depicted. By deleting their related transitions, as shown in Figure 5.24, in the same way, a task was previously deleted, the tasks can now be dragged and dropped into the correct order, as shown in Figure 5.25. Next, a transition can be added by selecting the parallel gateway and choosing the option to add a transition from the pop-up menu, connecting it with *Measure SBP*. Similarly, the two tasks can be connected via

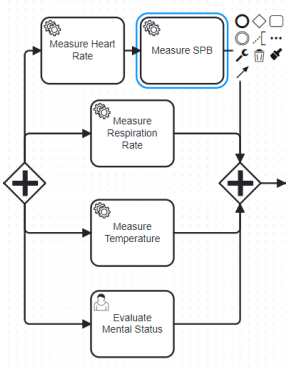



Figure 5.21: Task element to be removed

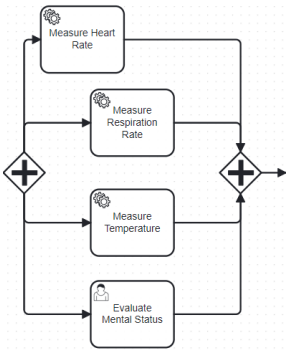



Figure 5.22: Task element removed

a transition, and *Measure Heart Rate* can be connected to the second parallel gateway, as depicted in Figure 5.26.

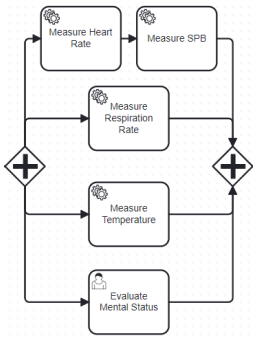



Figure 5.23: Initial task arrangement

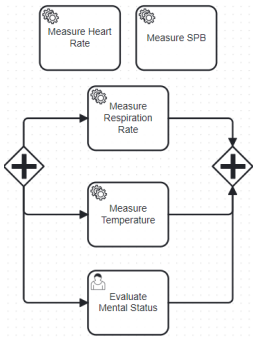



Figure 5.24: Transitions deletion

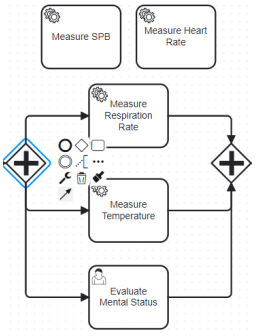



Figure 5.25: New task arrangement

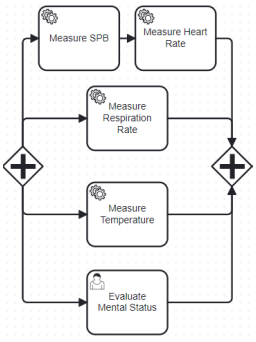



Figure 5.26: Transitions addition

### 5.6.4 Time Constraints - Events

Considering time constraints, events can be added to the workflow to enforce timing requirements. For example, if in the sepsis workflow it needs to be ensured that the *Evaluate Mental Status* task (as shown in Figure 5.27) is executed within one hour, a set of timer event elements (shown in Figure 5.28) is available and can be added to that specific task, as illustrated in Figure 5.30. The timer's properties are configured appropriately, as shown in Figure 5.29, which is set to fire after one hour using the ISO 8601 duration format (PT1H). By changing this value, the time constraints can be reconfigured. This timer starts counting from the moment the task is triggered for execution. If one hour passes without completing the task, the timer event is triggered, transferring the token from *Evaluate Mental Status* to the *Escalate Delay* task, which escalates the delay. Once escalated, the token is transferred back to *Evaluate Mental Status* to be completed.

A timer boundary event is used in this case because it behaves differently from other timer types. Unlike a timer start event, which replaces the start event and triggers workflow execution at a specific time, or an intermediate catch event, which delays the flow without being attached to a task, a boundary event is attached to a task and monitors its execution time. A Timer Intermediate Catch Event can also be used to introduce a delay between tasks before continuing the process flow. Timer events can also be removed, in the same way as shown earlier in the task removal scenario.

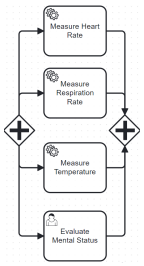



Figure 5.27: Initial workflow part

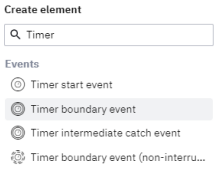



Figure 5.28: Timer event selection

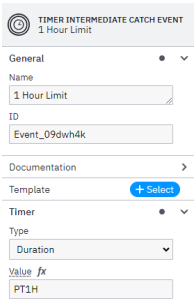



Figure 5.29: Timer configuration

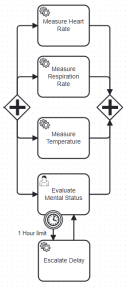



Figure 5.30: Workflow with timer and escalation

### 5.6.5 Conditions

As stated in the change scenarios, conditions must support addition, removal, and editing. Suppose the part of the sepsis workflow to be reconfigured is shown in Figure 5.31. A condition needs to be added to meet hospital requirements and execute the *Evaluate Mental Status* task conditionally in order to improve workflow efficiency. By deleting the transition that connects *Measure Temperature* and *Evaluate Mental Status* and selecting *Measure Temperature*, an exclusive gateway (condition) can be added, as shown in Figure 5.32. By adding the appropriate transitions from the gateway to the desired components, the intended execution flow is created, as shown in Figure 5.34. The specific conditions still need to be set. This is done by defining the conditions under which each outgoing transition from the exclusive gateway is triggered. In Figure 5.33, the properties of the transition from the exclusive gateway to *Evaluate Mental Status* are shown. In the Condition Expression field, it is set that this transition should be executed only if the temperature measured in the previous task is greater than or equal to 38°C. The condition for the other transition is set accordingly. The variable *temperature* is defined by the *Measure Temperature UF* implementation and must be known to add it to the BPMN workflow representation. Variables can also be added from manual tasks and used similarly. Therefore, it can be seen that conditions can be added without requiring code-level changes.

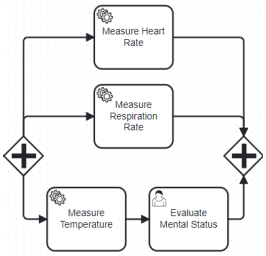



Figure 5.31: Initial workflow part to be reconfigured

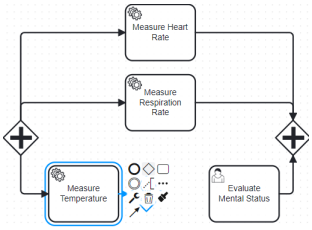



Figure 5.32: Exclusive gateway added to control flow

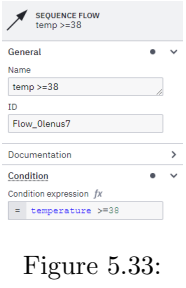



Figure 5.33: Transition condition configuration (temperature 38°C)

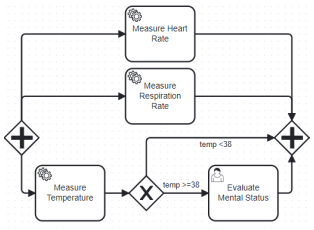



Figure 5.34: Final conditional workflow with transitions

One drawback of the condition setting is that, since the BPMN is unaware of the variables set by the UFs at design time, a correctness check of whether the variable is correctly written is impossible. However, the condition should follow Friendly Enough Expression Language (FEEL) and can be checked in its format. Now, if a condition needs to be removed, it can be removed in the same way as a task or transition removal. With its removal, all the related transitions

will automatically be removed, so the new task relations need to be set once again. Finally, it is obvious from Figure 5.33 that the editor can change the Condition Expression, and thus, the condition is configurable as long as the condition follows the FEEL rules and the variables are written correctly.

### 5.6.6 Transitions

As shown in the change scenarios, the editor should support transition addition, removal, origin and target editing. For example, multiple transitions were added during the demonstration of condition addition. These connected the exclusive gateway with *Evaluate Mental Status* and the parallel gateway. By selecting the exclusive gateway, the pop-up menu shows all the allowed elements that can be added. As shown in Figure 5.32, one of these is the transition (directed arrow). Once selected, the origin of the transition is attached to the exclusive gateway, and the target is attached to the mouse pointer, allowing the user to select any component to set as the transition's target. Accordingly, if a transition's origin or target needs to be reconfigured, the transition must first be selected. Then, the origin or target can be dragged and dropped to the new desired component.

### 5.6.7 Pre-postconditions

In the change scenarios, it has been stated that preconditions and postconditions defined in the workflow should be modifiable. To illustrate this with an example, a precondition for the sepsis workflow has been implemented, as shown in Figure 5.35. Just after the start event of the workflow, a subprocess component is placed, inside which a small process is defined. This process can be completed only if the patient ID has already been provided in the request for workflow execution and is thus registered in the process instance variables, or otherwise must be provided by the clinicians executing the user task. In this way, the rest of the workflow ensures that for every task, the patient ID is available, and when an automated task is executed, the patient ID is included in the message payload so that the UFs and adaptors can identify for which patient that task needs to be executed.

The precondition could also be implemented without being encapsulated in a subprocess component. However, having it implemented as a subprocess clarifies the boundaries of the precondition and makes it easier to reconfigure, as its responsibilities are separated from the rest of the workflow and testing for its functionality can take place only within this component. Using the previously described reconfiguration actions, a precondition can be modified, added or removed without requiring any code-level changes. Similarly, if a precondition is placed before a specific task within the workflow, it acts as the precondition for that task. If placed after a specific task or after the workflow, with the correct logic, it can act as a postcondition, ensuring that the task or workflow has been executed correctly and produces the expected results.

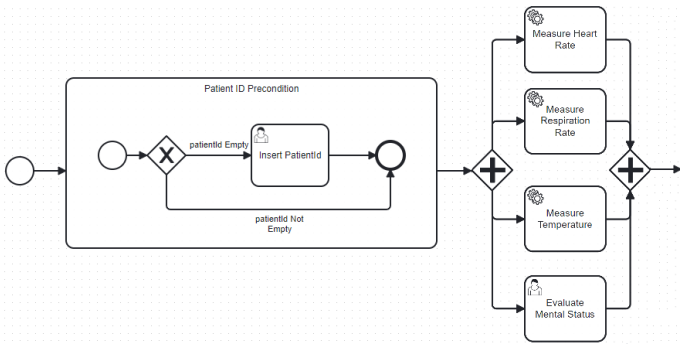



Figure 5.35: Example precondition in workflow.

### 5.6.8 System Adaptation to external Environment Variability

Regarding the system adaptation to environmental variability, in the change scenarios, it has been explained, that same external systems may not exist in different hospitals. The core of the idea in such a scenario is that, based on the systems available at each hospital, the correct adaptors should be enabled, and thus the system adapts automatically without any code-level changes, assuming that a complete list of adaptors, for the different types of external systems that CWS needs to interact with, has been implemented .

For such a scenario, an analogous demonstration has been created to show that the system can adapt in such situations. Suppose that there exist two types of patient monitors globally, as has been shown in the system implementation architectural diagram (Figure 5.6). Therefore, two different adaptors need to be implemented in the CWS to communicate with these two different patient monitors. Now suppose that the CWS needs to be deployed in a hospital that uses the patient monitors of type one and in another the other type of patient monitors is used. In such a situation, the system can adapt by running the corresponding adaptor service during the deployment phase of the system, in this case deployed in a container, as shown in Figure 5.36.

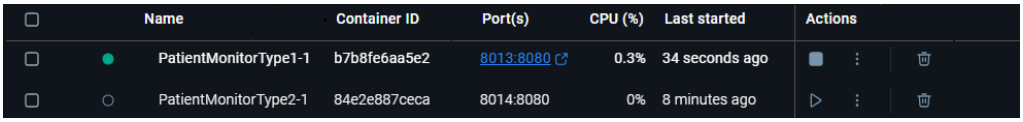



Figure 5.36: Enabled adaptor container for hospital-specific patient monitor type.

A reasonable question is why this does not lead to code-level changes to other components. This is illustrated in Figure 5.37. This image illustrates that, for example, the *Measure Temperature UF*, when it publishes a message with the routing key/topic *Temperature.Request*, the respective queues of both adaptors are interested in such topics. Thus, any enabled adaptor based on the hospital where the CWS is deployed can receive the message, process it, and respond. So, for example, in a hospital where only PMs of type one exist, only the corresponding adaptor will be enabled, and the other will be set disabled (container not running). Also, using the decoupling advantage that the *Exchange* decouples the UFs and Adaptors, none of the UFs need to be customized to adapt to the set of enabled adaptors. However, an assumption has been made that at any time, at least one adaptor that can satisfy UF requests is enabled. To conclude, when an adaptor container is enabled, it will automatically, without any code-level change, receive the appropriate requests from UFs and will be able to respond. That way, the *Check Temperature UF* does not know which adaptor serves its requests, but in any case it will receive the required measurement without concerning about interoperability and external systems details.

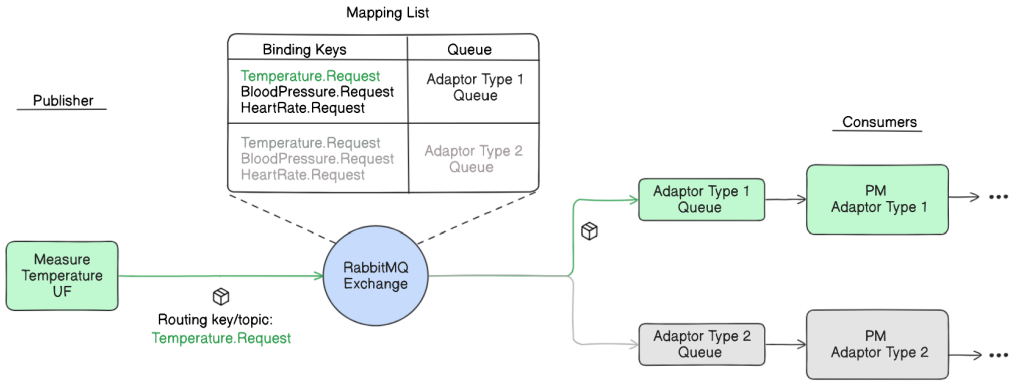



Figure 5.37: Decoupled UFs–Adaptors communication via RabbitMQ exchange

Finally, there might be cases where multiple external systems are capable of carrying out the

same request and are both enabled. For example, suppose that for a reason the two different types of patient monitors included in the proof of concept implementation exist in the hospital and be linked to the same patient, so both can respond to temperature measurement requests. In this case, the UF must implement logic to handle multiple responses. One approach can be to accept the first successful response and ignore or flag any subsequent ones. In the proof of concept implementation, if two different PM adaptors are enabled, both will receive the same request and return a response to the *Measure Temperature UF*. The UF then retains the first response and discards any further ones, while the task is marked as completed, as shown in Figure 5.38. In this picture, the logs from the *Measure Temperature UF* are depicted. So once it receives the first response from one of the two adaptors, it marks the job as complete and assigns the new temperature variable into the variables of the workflow instance by interacting with the execution engine. However, it is shown that another successful response for the specific job is received later. Then the second response is ignored as the task has already been completed after receiving the first temperature response.

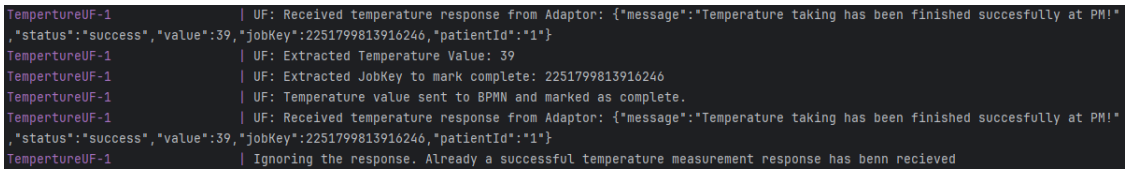



Figure 5.38: First temperature response processed; subsequent responses ignored.

### 5.6.9 Interoperability

Interoperability protocols for interacting with external systems should be implemented inside the adaptors. In the provided system implementation, each adaptor handles communication with a specific type of external system, translating requests and responses using the appropriate protocol. For example, *PM Adaptor Type 1* implements the necessary protocol to interact with the respective device type. By implementing adaptors for these system types, the overall system becomes capable of interoperable interaction with a wide range of external systems in a modular, maintainable, and extensible way. For example, in case that CWS adopts HL7 FHIR protocol for the internal components communication then any external system can be integrated with CWS, as long as exist an adaptor that translates between FHIR and the external system's native API.

### 5.6.10 External System Failure - Messaging Failure

In addition to the other aspects of the system, the fact that the CWS interacts with external systems and depends on them for task completion means that mechanisms must be in place to safeguard workflow execution in the event of errors. Specifically, the system must be capable of handling situations where external systems fail, become unreachable, or do not respond to CWS requests. Error-handling mechanisms can be implemented within the adaptors to manage such situations. For example, logic can be added in the PM adaptor to detect request failures and respond appropriately to the UFs that requested the information. An illustrative example is when the *Measure Heart Rate UF* requests the heart rate for a patient with ID 1234, but the patient monitor assigned to that patient is unresponsive, or its network connection has failed. In such a case, the adaptor should recognize the failure and respond with a specific code or message to the UF, indicating that the request did not succeed due to external system failure.

This approach can be further supported by defining postconditions in the BPMN workflow model, which act as validation checkpoints to confirm whether tasks have been completed successfully. If not, the workflow can trigger compensating or fallback actions, as shown in Figure 5.39. Another approach is to let the UFs handle such failures directly by activating redundant or backup

actions. While this is supported by the system design, it raises the concern that UFs should remain focused on their core responsibility of performing a single, well-defined task—and should not become overloaded with complex error-handling logic.

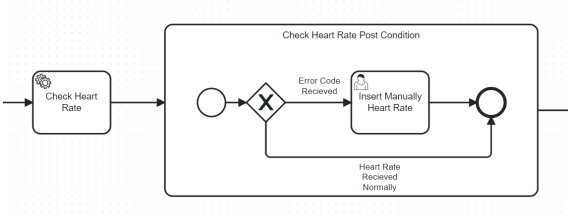



Figure 5.39: Postcondition fallback for heart rate retrieval failure.

### 5.6.11 Deployment Scenarios

In change scenarios related to the deployment of the system, it has been highlighted that different component splits should be supported to enable flexible deployment configurations. In Figure 5.40, the most granular component separation is illustrated, which addresses these requirements.

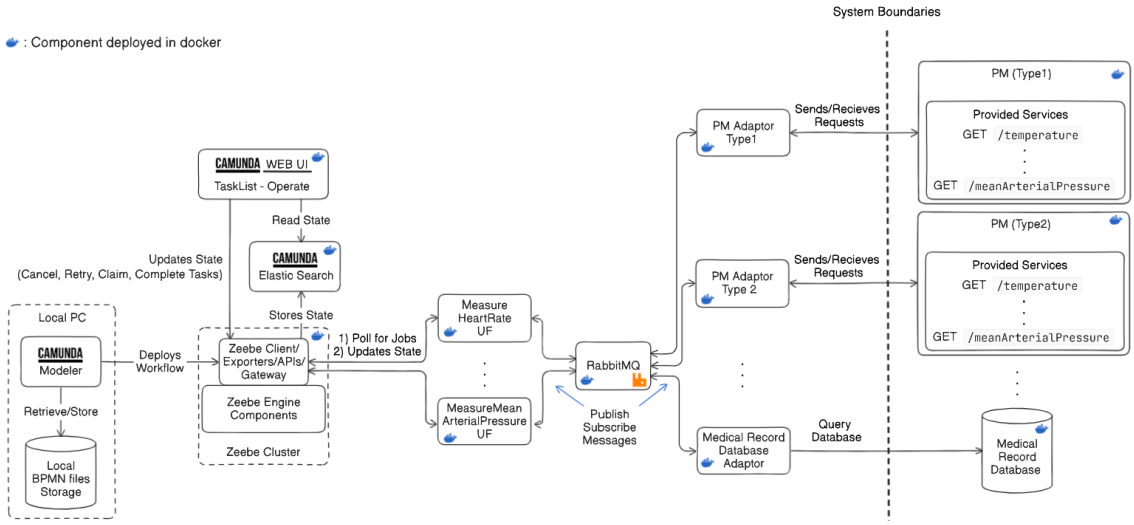



Figure 5.40: System Deployment

For the proof of concept, Docker containers have been chosen as the deployment method for most system components. These containers run locally on a single machine. A screenshot showing the Docker container dashboard, confirming that all core CWS components are deployed and running in containers, is included in the Appendix (see section B.6). Only the Camunda Modeler and the Workflow Storage components are not containerized and instead run directly on the local PC. Importantly, due to the portability and isolation capabilities offered by containers, it is evident that they can be deployed on any device with sufficient resources, thereby adapting to different deployment needs. For instance, in a centralized deployment, all components could be hosted in the cloud or on a central server. Conversely, if the CWS needs to be deployed in a per-patient approach in independent patient monitors, the components can be deployed directly on those devices.

Regarding the centralized deployment scenario, the CWS should be capable of executing multiple workflow instances for different patients. The diagrams provided in the previous sections

show that each job created from the execution engine and executed by the UFs is uniquely defined by the *JobKey*. This identifier enables the system to identify jobs from different workflow instances uniquely. Thus, the system can operate even when multiple workflow instances are executed simultaneously.Lastly to handle mixed deployment scenarios, the system may be deployed in both centralized and edge environments, communicating through a dedicated adaptor. In such cases, clear protocols must be established to define which system instance acts as the master in order to avoid inconsistencies in workflow execution and data synchronization.

In conclusion, this chapter effectively demonstrated that the proposed system architecture can provide the required functionality stated in the requirements analysis. A proof-of-concept system implementation has been provided, and all the advantages of the system have been highlighted, resulting in a system that, based on the requirements, is capable of coping with new requirements with minimal cost and without code-level changes. The proof-of-concept code implementation, is available at GitHub.

# Chapter 6

# Evaluation & Results

The purpose of this chapter is to assess whether the CWS design as proposed in chapter 4 and implemented in chapter 5 satisfies the three core requirements identified, with these being workflow configurability, adaptability to external system variability, and flexibility of deployment. These requirements were operationalized through a set of change scenarios described in section 3.2. Each scenario was created to test the system's ability to handle the relevant changes. Therefore, successfully handling these change scenarios provides evidence of requirement satisfaction.

## 6.1 Evaluation Criteria

The evaluation of the CWS will be done based on three core criteria. First, whether the system can handle the predefined change scenarios without requiring code-level modifications. Second, whether the target system qualities that guided the system design, such as configurability, adaptability, integrability, and others, are realized in the architecture and validated through relevant change scenarios or architectural analysis. For each quality, the evaluation examines the architectural concepts that support it, and whether their intended goals are confirmed by the system's behavior when executing the scenarios or through design inspection. Third, whether the system maintains architectural integrity and correct behavior when applying changes. Together, these criteria provide a structured basis to assess whether the system meets its intended design goals and demonstrates the required level of evolvability.

As many of the target qualities directly contribute to satisfying the three key requirements, there is a natural overlap between the key requirement evaluations and the quality-based evaluations. However, providing direct evaluation of these two perspectives separately offer a comprehensive assessment of how the system achieves its intended properties and design objectives.

## 6.2 Results per System Requirement

This subsection presents the results of evaluating the system against the defined system requirements.

### 6.2.1 Workflow Configurability

The first requirement, workflow configurability, was tested through scenarios covering all the workflow variation points shown in Table 3.1. All scenarios were implemented through the workflow editor without code-level changes, as well as deployed in the execution engine and executed successfully. An additional concise demonstration is depicted in the following images. Initially, the base version of the workflow definition is presented in Appendix C, where Figure C.1 shows a part of the initial workflow definition, and Figure C.2 shows the corresponding execution state of the workflow instance in Camunda Operate. Then, in Figure 6.1, the reconfigured part of the

workflow is shown, applying the reconfiguration actions already demonstrated, and after deploying the updated workflow in the Zeebe Cluster, it is executed, and its state is shown in Figure 6.2, which shows that despite the changes, the updated workflow definition can be instantiated and executed by the Zeebe Cluster without any code-level change. The blue arrows indicate the paths the execution followed and in the instance history panel (bottom left corner) is shown which tasks have already executed.

In both images showing the workflow state, all prior tasks (e.g., *Measure Heart Rate*, *Measure Respiration Rate*, etc.) have checkmarks next to them in the instance history panel, indicating successful execution. The current task is *Evaluate Mental Status*, with an indicator that it is waiting for user input (shown as a green circle), which is why the workflow has not yet proceeded to the next tasks. For executing this task a user must complete it manually using Camunda Tasklist (see Figure C.3).All related test runs produced correct behavior, with the intended changes reflected in workflow execution, while the editor ensures workflow syntax validity before deployment. Based on the results and the evaluated scenarios, the workflow configurability requirement, as defined in section 3.3, is deemed satisfied.

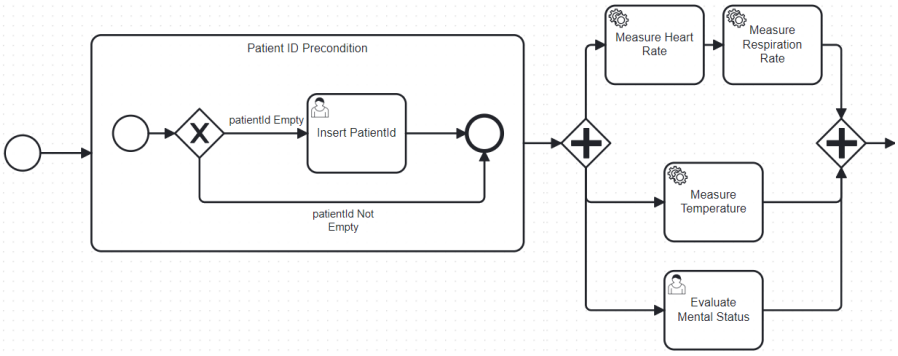



Figure 6.1: Alternative workflow configuration reflecting a different structure

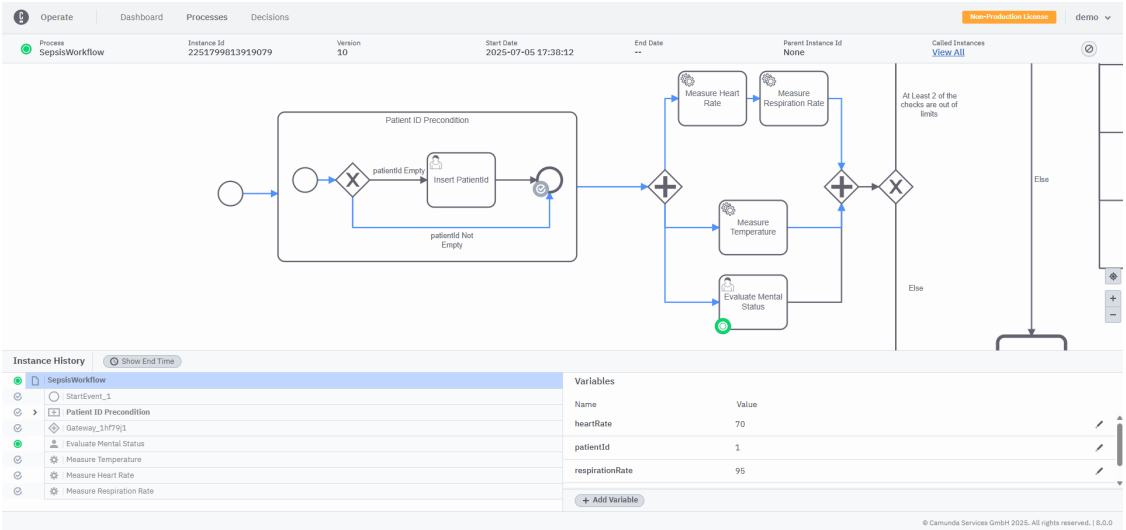



Figure 6.2: Camunda Operate - Successful execution of the alternative workflow configuration

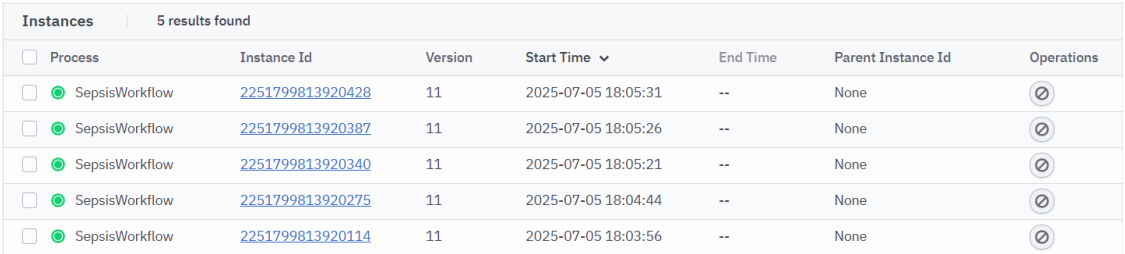
### 6.2.2 CWS Adaptability, Flexibility & Integrability with External Systems

The second requirement, CWS adaptability, flexibility and integrability with external systems, was evaluated through scenarios involving interaction with different types of external systems and dynamic selection of the appropriate adaptor based on the CWS environment. In this proof of concept, the correct adaptors are enabled or disabled manually. However, the system design allows for incorporating additional logic to automate this process. For example, a hospital specific configuration file could specify which adaptors to enable based on the hospital environment, or the deployment system could include a service discovery mechanism to automatically detect available systems and launch the corresponding adaptors. The current implementation leaves room for such extensions.

All scenarios were implemented using the system's adaptors without requiring code-level modifications, and all integration tests were completed successfully. The system was able to handle variability of external systems by enabling the correct adaptor for the external system present in the operational environment (see Figure 5.36). This confirms that the CWS adapts to external system variability without code changes. This is supported by the decoupling of UFs and adaptors through the message broker (see Figure 5.37). Additionally, making the adaptors responsible for applying interoperability protocols ensures that the rest of the system remains independent from external systems and is easily adaptable to external system availability. In the case of multiple responses from different external systems, UFs can apply protocols and handle this situation (see Figure 5.38). Also, in case of external system failure, the CWS is capable of handling failed requests by implementing fallback actions in the workflow definition (see Figure 5.39). Based on these results and the evaluated scenarios, the requirements related to CWS adaptability, flexibility, and integrability as defined in section 3.3, are deemed satisfied, with the system demonstrating the ability to operate flexibly in different hospital environments.

### 6.2.3 Deployment Adaptability & Flexibility

The third requirement, deployment adaptability and flexibility, was demonstrated through the containerized deployment of the system components. For these scenarios, all components were deployed in Docker containers, as shown in section B.6, demonstrating their ability to be deployed on different infrastructures and independently of the hardware. In the centralized deployment scenario, the system can handle concurrent workflow executions using unique identifiers for workflow instances and, by extension, for jobs/tasks. For example, in Figure 6.3, Camunda Operate shows that multiple instances of the sepsis workflow definition are executing concurrently in the system, with each workflow running for a different patient, identified by an assigned variable inside each workflow instance. The per-patient deployment is supported, as all components can be deployed on the same machine and execute workflows for a specific patient only. Similarly to centralized deployment running multiple instances of the same workflow definition, both deployment types can also handle different workflow definition instances execution concurrently.



Instances | 5 results found

| Process | Instance Id | Version | Start Time | End Time | Parent Instance Id | Operations |
|---|---|---|---|---|---|---|
| SepsisWorkflow | 2251799813920428 | 11 | 2025-07-05 18:05:31 | -- | None | |
| SepsisWorkflow | 2251799813920387 | 11 | 2025-07-05 18:05:26 | -- | None | |
| SepsisWorkflow | 2251799813920340 | 11 | 2025-07-05 18:05:21 | -- | None | |
| SepsisWorkflow | 2251799813920275 | 11 | 2025-07-05 18:04:44 | -- | None | |
| SepsisWorkflow | 2251799813920114 | 11 | 2025-07-05 18:03:56 | -- | None | |

Figure 6.3: Concurrent workflow execution supported

The mixed deployment scenario (workflow migration between centralized and per-patient deployment) was not practically demonstrated due to time constraints. However, the system's mod-

ular design enables the integration of another CWS deployment as an external system, in the same way it interacts with other external systems. For this scenario to be implemented, synchronization protocols must be applied in both CWS deployments, specifically within the adaptors responsible for cross-CWS communication. In summary, the system design satisfies both centralized and per-patient deployment scenarios, and with future extensibility mixed deployment. Also, in extreme cases, the CWS design supports fully decentralized deployment, with each component running on different hardware.

## 6.3 Results per System Quality

Now that the three main system requirements were explicitly evaluated, let's move on evaluating the realization of the target qualities in the system design, based on the architecture concepts used and the change scenarios. The definitions of each of these qualities, in the CWS context, has been provided in section 3.3.

- *Configurability:* It is achieved by separating control logic (workflow definition) from functional logic (execution). An editor can reconfigure the workflow definition using configuration files by following specific rules and linking UF services to add functionality. Then, the configuration file can be independently executed by the execution engine, which interprets and executes it. This approach enables complete decoupling of system implementation from workflow definitions, allowing workflow reconfiguration without code-level changes and system rebuilt. This quality was evaluated through the workflow configurability scenarios (task addition, removal, reordering, role assignment, conditions, transitions, pre- and postconditions, and events). One such reconfiguration is shown in Figure 6.1, where an alternative workflow structure was defined through configuration. Figure 6.2 confirms that the modified workflow was executed successfully, with tasks progressing correctly and the system responding as expected. Additionally, all related scenarios, shown in section 5.6, were successfully implemented and executed through configuration alone without code changes or system rebuilt. Based on the results and the evaluated scenarios, the workflow configurability requirement, as defined in section 3.3, is deemed satisfied.

- *Adaptability & Flexibility:* Adaptability is enabled by using adaptors, one for each type of external system. This makes the system capable of adapting to the external system availability at deployment time without needing any code-level changes by enabling the appropriate adaptors based on the availability of external systems. To support this, the assumption that for each external system type an adaptor is already implemented and available to be deployed must hold. Flexibility is enabled by the modularity incorporated into the system to add the needed UFs and adaptors without affecting any other components of the system. Decoupling of UFs and adaptors, enabled by the message broker, further supports both qualities. These qualities were tested through the external system variability scenarios in subsection 5.6.8, including the absence of PM type two, the use of two external systems for the same data field, thus integrating new external systems through configuration. In all cases, the system successfully adapted to the external environment through configuration alone. Based on the evaluated scenarios, the system adaptability and flexibility requirements, as defined in section 3.3, are deemed satisfied.

- *Integrability:* Integrability is achieved by decoupling the CWS components' responsibilities and implementation, from external system-specific logic. All such logic is encapsulated in an intermediate layer at the system boundaries, the adaptors. By having one adaptor per external system type, the system can integrate with new external systems without requiring changes to existing components. When a new external system type arises, a corresponding adaptor can be implemented, compiled, and deployed without modifying any other part of the system, as adaptors are decoupled from both the engine and the UFs. Communication is standardized via predefined message contracts of publish-subscribe protocol, enabling the

system to integrate new functionality through configuration alone (e.g., enabling or disabling the adaptor). This was demonstrated through the external system variability scenarios (subsection 5.6.8), including adding new external systems, switching between available systems, and handling the absence of expected systems. In summary, integration was achieved solely by adding a new adaptor and configuring it (enabled/disabled), while no changes to other components were necessary. Thus the system integrability requirement, as defined in section 3.3, is deemed satisfied.

- *Extensibility:* This is achieved by adopting a highly modular architecture. UFs can be extended based on evolving requirements, and adaptors can be added to support new types of external systems. Due to the decoupling of UFs and adaptors, new functionalities or integrations can be introduced without impacting the other system components. Extensibility is demonstrated via the Sepsis workflow execution scenario, where each UF can be implemented independently. If a new UF needs to be added, this can happen without modifying the execution engine, message broker, or adaptors. Similarly, adaptors can be added, updated, or maintained similarly. These additions happen without causing any impact to other components, while the communication between UFs and adaptors is standardized via the message broker. The proof of concept leverages this to introduce new UFs and adaptors independently, validating that system behavior can be extended correctly without modification of existing components. Thus, the system demonstrates strong extensibility, allowing it to evolve with minimal effort and minimal architectural touch-points.

- *Scalability:* This is achieved by containerized deployment of the system architecture components in different containers, which can scale independently. Based on the workload (running workflow instances), the system's components can scale up or down, supporting this quality. Even though explicit tests on workload did not take place, the modular architecture allows component instances to scale as needed. By utilizing Kubernetes, the system could supports automatic workload-based scaling through mechanisms such as the Horizontal Pod Autoscaler (HPA), allowing multiple instances of components such as the execution engine, UFs, message broker, or adaptors to run in parallel and share the workload efficiently. In conclusion, scalability is supported by the system, although it was not stress-tested in the proof of concept.

- *Modularity:* This is achieved by clearly separating system concerns across different components. Thus, each component can achieve a specific goal leading to clear separation of concerns. This is demonstrated in all diagrams and change scenarios provided (e.g. Figure 5.6). UFs can be added or modified without affecting other UFs, adaptors or the execution engine. Similarly, adaptors can be added or modified without affecting other components, and workflows can be modified without changes to the execution engine. Together, these demonstrate that all components can evolve independently. In conclusion, modularity is demonstrated and satisfied for the considered change scenarios, and it is a key enabler of other qualities of the system such as configurability, adaptability or integrability. In contrast, without modular adaptors, and under a more coarse-grained design, introducing or modifying a single adaptor type would affect the other related adaptors, thereby breaking the system's ability to evolve components independently.

- *Robustness:* This is achieved by designing the system to remain functionally correct despite variability in external systems. Each external system is accessed through a dedicated adaptor, ensuring that variations in protocols or availability do not affect overall system behavior. For example, based on the example of Figure 5.36, the sepsis workflow has been executed twice, once using the PM type 1 and in a separate execution using the PM type 2. This was done by enabling in each execution the appropriate adaptor. Figure 6.4 and Figure 6.5 show two independent executions of the *Measure Respiration Rate* UF, each performed via a different type of patient monitor using its corresponding adaptor. Thus, having tested the functionality of adaptors for the external system type that is responsible, then the

internal system behavior can remain independent of the external systems. Based on these design principles and implementation patterns, robustness is considered demonstrated in the system.

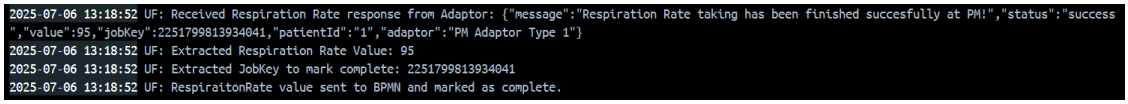



Figure 6.4: *Measure Respiration Rate UF* executed through *PM Adaptor Type 1*


Figure 6.5: *Measure Respiration Rate UF* executed through *PM Adaptor Type 2*

- *Maintainability:* This is achieved by making the system modular and adopting a clear separation of concerns between components. As a result, components can be maintained independently without affecting the functionality of others. Engineers can pinpoint the component in need of maintenance and proceed without requiring extensive system-wide testing. For example, when an adaptor needs to be updated, such as to support a new interoperability protocol, this can be done without modifying other system components. The system supports local changes with limited regression scope and avoids system-wide revalidation, as adaptors are independent from the message broker and other components. The same applies to UFs, which are not coupled to adaptors. Maintenance in workflow definitions is only needed if the UF's logical endpoint (e.g., the Zeebe task type or semantic identifier) or core function changes. This is an uncommon scenario, as UFs are fine-grained, clinically grounded units with little need for modification. These modularity and thus maintainability of the system is illustrated in the architecture diagrams (see Figure 5.6) in the previous chapter. Based on this analysis, the system satisfies maintainability for the considered scenarios.

- *Upgradability:* This is enabled, similarly, by the modular architecture of the system and clear separation of concerns. Modularity allows engineers to identify which component needs to be upgraded and do it without affecting the rest of the system. This is achieved by implementing and deploying the new component into the system. For example, suppose a new UF or adaptor is identified or the execution engine needs to be upgraded to support more workflow languages and needs to be integrated into the system. In that case, it can be independently implemented and integrated into the system as long as it provides the required interfaces, like the rest of the components of this type. Thus, upgradability is integrated into the system and can occur without requiring the rest of the system's components to be redeployed.

- *Resilience:* This is achieved by the system's ability to manage external system failures through various mechanisms. These include the use of postconditions in tasks that depend on external systems (as shown in Figure 5.39), ensuring that failures, in external or internal components, are detected and handled appropriately. Additionally this can be done with fallback actions incorporated in the workflow as shown in Figure 6.6, where timer event can identify unresponded requests to external systems and execute manual fallback actions. Finally, the workflow engine persists workflow state, allowing recovery by allowing retry executing the failed task and continuation. This is demonstrated in **??** and Figure 6.7. Based on the evaluated scenarios, the system resilience requirements, as defined in section 3.3, is deemed satisfied.

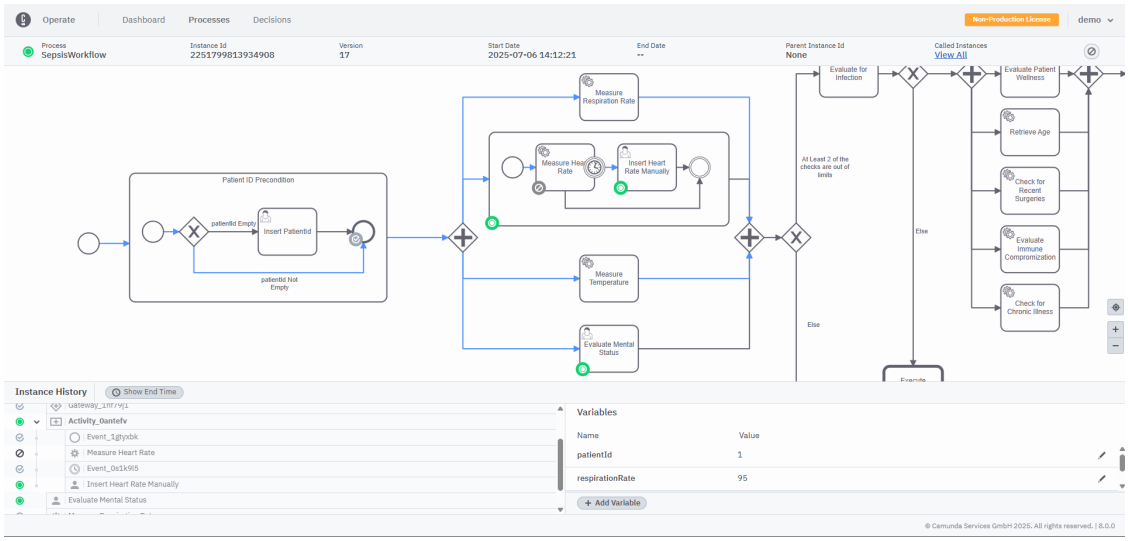



Figure 6.6: Resilience to external systems failure with fallback task

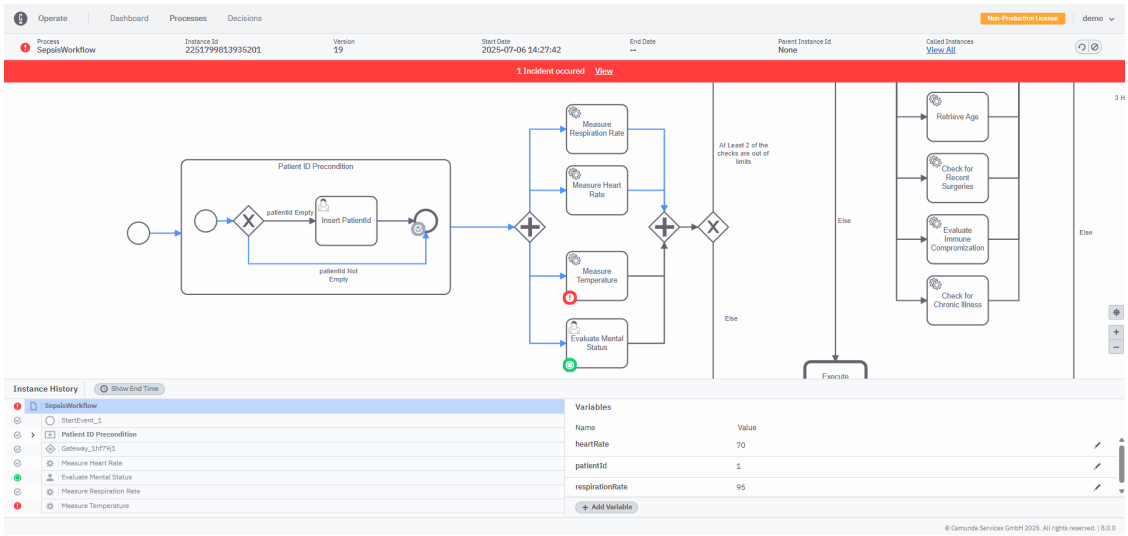



Figure 6.7: Failure Simulation

| Flow Node | Job Id | Creation Time | Error Message | Root Cause Instance | Operations |
|---|---|---|---|---|---|
| Measure Temperature | 2251799813935225 | 2025-07-06 14:27:43 | Simulated failure in Temperature UF | -- | |

Figure 6.8: Failure Simulation - Retry operation

Although it isn't part of the selected system qualities, it may be interesting to discuss composability quality. Composability is a design principle that refers to the ability to build more complex behavior by combining smaller, self-contained components. A system is said to be composable if its individual parts can be reused and assembled in various ways without requiring internal components modifications. In the context of clinical workflows, composability means that simple building blocks, such as UFs that perform individual clinical tasks, can be arranged or grouped together to form more advanced logic. For example, a blood pressure check and a heart rate measurement could be composed into a broader "vital signs assessment" sub-workflow. This can be achieved by defining the correct sequence of tasks in the workflow definition using the modularly defined UFs. Thus, composability is supported by the system at the workflow level, meaning that no code-level changes are required. However, for system engineering stakeholders, if this is to be done at the UF implementation level itself, it is evident that implementation testing and recompilation of the new UF generated by combining the two individual ones should take place.

Finally, the scenarios that can be handled without code-level changes have been thoroughly explained. However, there are two main cases where code modifications are unavoidable: the addition or maintenance of UFs and adaptors. In the previous sections, these components were assumed to already exist as prerequisites, that is, a complete set of UFs and adaptors was available. Nonetheless, unforeseen cases may arise where a new adaptor type or UF is needed. In such situations, component implementation, testing, and deployment become necessary. Similarly, the maintenance of any system component inherently requires code-level changes, along with testing and rebuilding. However, even in such scenarios that code level changes are still required the system maintains the architectural integrity, as all component are fully decoupled and independently replaceable. As previously discussed, the impact of such changes is minimal, since all components are independent and unaffected by modifications in other parts of the system.

Having discussed extensively all these system qualities, in the Table 6.1 bellow is summarized all the evaluation done above. For each system quality, has been summarized which change scenarios are used for evaluation, the problem it addresses, how its solved and the success level.

| System Quality | Change Scenario(s) | Problem Addressed | Captured By | Level of Success |
|---|---|---|---|---|
| **Workflow Configurability** | All workflow reconfiguration scenarios shown in section 5.6 (e.g., task addition/removal/reordering, role change, conditions, transitions, pre/postconditions, events) | Enable variation in clinical workflow logic across hospitals without code-level changes | BPMN-based workflow reconfiguration using Camunda Modeler and Zeebe engine | All scenarios demonstrated successfully |
| **Adaptability & Flexibility** | External system variability, integration of new systems, multiple external responses, deployment alternatives (per-patient vs. centralized) | Accommodate heterogeneity of IT infrastructure and support hospital-specific workflows and deployment needs | Modular adaptor system, message broker decoupling, enable dissable adaptors without code-level changes | All scenarios demonstrated successfully except of the the mixed deployment scenario which needs further implementation to be practically demonstrated |
| **Integrability** | Plug-in adaptors for protocol-specific communication, switching between external system types | Isolate external protocol logic and enable plug-and-play integration | Adaptors using message broker abstraction and standard message contracts | All scenarios demonstrated successfully |
| **Resilience & Robustness** | Handling external system variability, failures or timeouts, retry mechanisms, fallback actions | Ensure continued operation and graceful degradation under failure | BPMN fallback flows, timer events, post-conditions, persisted workflow state | All scenarios demonstrated successfully |
| **Scalability** | Increase in number of patients or concurrently running workflows | Future-proof system for large-scale deployment needs | Modular components, Kubernetes-ready deployment, container-based architecture | Not practically but theoretically tested, but supported |
| **Extensibility** | Adding new UFs or external system adaptors | Extend functionality for evolving clinical or IT requirements | Modular architecture message-based interface contracts | All theoretical scenarios demonstrated successfully |
| **Maintainability & Upgradability** | Component-level changes, bug fixes, protocol updates, upgrades to UF or adaptor logic | Enable safe, localized upgrades and long-term maintenance | Clear separation of concerns, modular independent components | All theoretical scenarios demonstrated successfully |
| **Modularity** | Isolated component evolution (e.g., UFs, adaptors, engine) | Allow independent development and replacement of components | Containerized components with publish/subscribe decoupling | Satisfied |

Table 6.1: Summary of Evaluation per System Quality and Scenario

In conclusion, the evaluation results show that all three core system requirements are satisfied, and all targeted qualities are demonstrated in the system architecture and validated through the corresponding change scenarios and diagrams. The system therefore meets its intended design goals and exhibits the desired level of evolvability.

## 6.4 Proposed Design Boundaries and Future Extensions

The earlier section presented a thorough assessment of the system's architecture and its capability to meet the specified requirements. This section will address features that have yet to be incorporated into the existing design/proof of concept implementation but may become relevant as future operational requirements emerge. Although the current architecture theoretically accommodates many of these extensions, by identifying these not-yet-implemented aspects, we outline the current boundaries of the system's functionality and pinpoint areas where future needs could pose challenges on the architecture. However, this does not imply inherent limitations in the architecture itself. Rather, these boundaries reflect conscious design choices made during this development phase and serve to clarify which extensions would require additional mechanisms or refinement.

The proposed system design uses a publish/subscribe communication model, which brings many advantages in modularity, decoupling, and scalability. Although this design choice does not inherently support hard real-time guarantees (i.e., deterministic timing under all loads), bounded real-time behavior can still be achieved. This can be done by defining upper bounds on input workload and ensuring that the message broker and the rest of the processing components (UFs, adaptors) operate within specified performance limits. Thus, by integrating proper load management, the system could deliver predictable execution times for operations execution. In case of future requirements require real-time assurances, further mechanisms such as rate limiting, priority queues could be explored.

The current architecture relies on a single instance of a message broker to manage communication between UFs and adaptors, which creates a possible single point of failure. Consequently, if the message broker becomes inaccessible, communication between UFs and adaptors will be interrupted. However, this is not a fundamental limitation of the proposed system, because the system design can be extended to support high availability by integrating a high-availability message broker cluster. For instance, a high-availability message broker cluster can be deployed using Kubernetes, with replication and failover across broker pods. Kubernetes enable automatic recovery, while a load balancer can be used to distribute traffic across instances to maintain availability. Thus, the cluster could manage individual broker instance failures without causing system disruptions.

Another consideration in the current system design is the lack of native support for global transactional consistency across multiple UFs, or external system calls. In sectors like banking, this would mean that if several operations must either all succeed or all fail, the system would need to manage distributed transactions or automatic rollbacks. However, the proposed system does not integrate a distributed transaction manager in its design. In contrast, transactional consistency can be incorporated into the workflow definition. As has been shown, the design supports the atomic execution of individual UFs and allows workflow designers to model compensating actions that handle failures in multi-step operations. This approach allows workflow definitions to incorporate domain-specific transactional behavior as needed by the clinical use case. Since such scenarios are relatively uncommon in clinical settings, the use of workflow modeling does not overly complicate the system architecture. Thus, although global rollback mechanisms are not integrated into the system design, transactional consistency can still be achieved through workflow modeling.

A related consideration to the previous one is that the current system design provides basic monitoring via audit and execution logs but does not include full support for distributed tracing across UFs, adaptors, and external systems. This could introduce difficulties in identifying the root cause of failures or performance bottlenecks in complex workflows involving multiple asynchronous components. However, this can be compensated by integrating distributed tracing tools

like Grafana Tempo [78]. Each system's component/service can be instrumented using OpenTelemetry [79] to emit trace data. OpenTelemetry is a standard framework for generating and collecting trace information from distributed systems. Instrumentation refers to adding code or using agents that generate trace spans for key operations and propagate context across services. Grafana Tempo then collects this trace data and stores it, while Grafana [80] provides the interface to visualize and analyze the traces. This way traceability of the distributed CWS can be achieved and provide effective monitoring of the entire system.

Another consideration relates to the adaptors and, to some extent, the UFs. The system assumes that for each expected type of external system, a corresponding adaptor has been developed in advance and is ready for deployment. If an unforeseen external system must be integrated, a new adaptor must be implemented, which is a typical requirement for any system interacting with heterogeneous devices. This is not a limitation of the architecture, but rather a practical integration step that the design enables with minimal cost. A similar assumption applies to UFs, though in this case it is less problematic since clinical operations are relatively predictable, and a complete set of UFs can be developed at design time.

Finally, regarding system deployment, the main assumption is that there is at least one separate deployment per hospital. However, if multiple hospitals were to use the same shared cloud deployment, even though workflow instances and messages could be differentiated using a hospital identifier, there would be no strict separation at the data storage level (e.g., monitoring and state storage). In such a case, there is a potential risk of data mixing, for example, within the system monitoring component. To mitigate this, strict access control mechanisms and tenant isolation would be required. This is not an inherent limitation of the architecture, but rather a scenario it enables. The architecture supports both deployment models, and in the shared case, additional safeguards must be put in place to preserve data privacy.

The above considerations establish the boundaries of the currently proposed system design and implementation and point out some areas that could benefit from further implementation. Taking into consideration that system design is an iterative process, and new requirements arise through it, these requirements may be addressed in future iterations of the system, as the key requirements of the current project were focused on solving the fundamental problems of CWS workflow configurability, adaptability to external systems variability, and deployment alternatives.

Finally, in the Table 6.2 bellow, the mentioned system boundaries can be summarized, ranked by severity, and identified whether the boundary originates in the system design or implementation of the proof of concept, the impact, and the mitigation effort.

| Rank | Boundary | Impact | Source | Mitigation Effort | Notes |
|---|---|---|---|---|---|
| 1 | No global transactional consistency | Medium | Design | Medium | Affects workflow correctness when involving multiple UFs or external systems. It requires compensating logic in workflows. |
| 2 | No distributed tracing support | Medium | Implementation | Medium | Affects observability and debugging in workflow execution. This can be addressed using OpenTelemetry and Grafana Tempo. |
| 3 | Single point of failure: message broker | Medium | Implementation | Low | Causes disruption if the broker fails, however can be resolved with high-availability setup in Kubernetes. |
| 4 | No hard real-time guarantees | Low–Medium | Implementation | Medium | Relevant only if strict timing constraints are introduced. Bounded behavior can be achieved with rate limiting and resource management. |
| 5 | Assumption of prebuilt adaptors/UFs | Low | Design | Low | Common in integration-heavy systems. New adaptors are added as needed per external system type. |
| 6 | Per-hospital vs. shared deployment | Low | Implementation | Low | Architecture supports both. Shared deployments require standard tenant isolation mechanisms for data protection. |

Table 6.2: Ranking of Current System Design & Implementation Boundaries

As can be seen, most of the boundaries relate to the current phase of the proof of concept that could be faced by future developments. The only boundary originating from the system design are the lack of global transactional consistency, which, in the context of CWS, can be mitigated through workflow modeling, and the need for code-level changes when introducing new UFs or adaptors. However, these are handled effectively by ensuring that such changes have minimal impact and do not affect other system components. Thus, the system design is demonstrated to be well aligned with the evolving requirements without posing any major limitations.

# Chapter 7

# Research Question Answers & Discussion

So having presented all the steps taken for requirements analysis, system design, proof of concept implementation and scenario based evaluation as well as the current system design boundaries, let us concisely summarize all the findings revealed by this project by answering the research questions stated at the beginning of the current study.

## 7.1 Research Questions' Answers

### 7.1.1 Research Question 1

**RQ1: What are the key characteristics and evolving requirements of clinical workflows & CWS that should be supported by the proposed system design?**

1. What are the current requirements from the system stakeholders that clinical workflows demand from the system to enable workflow configurability and system adaptability to external system variability?

2. What are the predictable future requirements of CWS that the system design/architecture should accommodate?

This first research question aimed to identify the key characteristics and evolving requirements of CWS that the proposed system design should support. A thorough problem analysis, literature study, and PESTEL analysis were conducted in chapter 2 and chapter 3 to address this question. Domain specialists were consulted in order to validate the disclosed present and future CWS requirements. The key current and predictable requirements identified for such a system focus on three primary categories, which are (1) Workflow Configurability, (2) System Adaptability, Flexibility, and Integrability to the external environment, and (3) Deployment Flexibility and Adaptability discussed in subsection 3.1.4 and summarized in Table 3.3. Finally, system qualities related to evolvability and the identified requirements were selected to guide the system design. These qualities were contextualized for CWS and are discussed in section 3.3, based on their general definitions in the literature (Table 2.2).

### 7.1.2 Research Question 2

**RQ2: How can the CWS architecture be designed to support workflow configurability and system adaptability without requiring code-level modification and satisfy the identified current and future system requirements?**

1. What are the required components for such an architectural design?

2. How should workflows be defined to allow configurability?

3. What are the trade-offs that when building such a configurable and adaptable system should be considered?

The second research question aimed to reveal the CWS design that can address the identified evolving requirements. Regarding the first sub-question, the concepts applied to satisfy the identified requirements were analyzed in detail in chapter 4, and all the required components needed for designing such a system are summarized in Table 4.1, along with their primary responsibilities. Some of these are the workflow editor and execution engine, UFs, adaptors for external systems, a message broker for UFs-adaptors decoupling, and components for user interaction and workflow state storage. In addition to the main system design diagram shown in Figure 4.1, multiple other sequence diagrams are provided in section 4.2 to explain the system's behavior and its alignment with the requirements. The main concept applied to enable workflow configurability was the decoupling of the workflow control and functional logic. Regarding the second sub-question, the workflow control is defined in the workflow definition files, which the execution engine can interpret to trigger the execution of functional logic using the UFs. This separation allows reconfiguration of workflows without system rebuild, by editing configuration files instead of altering source code. Finally, regarding the third sub-question, the trade-offs identified during the design process are summarized and depicted in Table 4.2.

### 7.1.3 Research Question 3

**RQ3: What are appropriate criteria/metrics/questions for assessing evolvability of CWS?**

1. How can these criteria be applied to evaluate the proposed architecture's ability to adapt to future workflow/system changes?

2. What change scenarios could be used to test the system's configurability and adaptability to the different hospital environments?

3. What future workflow/system requirements are unlikely to be handled by the proposed system, and what limitations do these requirements pose?

Regarding the third research question, the system design was evaluated based on the three core criteria defined earlier. First, whether the system can handle the three main requirements and their respective change scenarios without requiring code-level modifications. Second, whether the target qualities that guided the system design, such as configurability, adaptability, integrability, and others, are realized in the architecture and validated through relevant change scenarios. For each quality, the evaluation examined the architectural concepts that support it, and whether their intended goals are confirmed by the system's behavior when executing the scenarios. Third, whether the system maintains architectural integrity and correct behavior when applying changes or when additional components are added to address specific requirements.

All these criteria were evaluated using the change scenarios constructed from the system requirements (shown in section 3.2). Next to per-requirement evaluation discussed in section 6.2, a summary of a per-system-quality evaluation is provided in Table 6.1 where each quality is linked to the change scenarios used for its evaluation, the problem it addresses, the architectural concepts applied to overcome it, and the outcome of the evaluation. Finally, an analysis of the proposed system design and the proof-of-concept implementation was conducted, and system boundaries were identified and summarized in Table 6.2. For all identified points, appropriate mitigations exist to overcome the limitations. It is important to note that most of these points concern the proof-of-concept implementation and can be addressed in future development iterations. For the few limitations related to the system design, both are explicitly discussed. In these cases, the architecture supports effective mitigation: new UFs and adaptors can be added without impacting other components, and transactional consistency can be addressed at the workflow level using

workflow-specific mechanisms. In summary, there was not identified any blocking limitation originating on the proposed system design highlighting its effectiveness on solving the given challenge.

Based on the above results, it can be concluded that all three research questions and their respective sub-questions have been successfully addressed. The system design fulfills the identified requirements, the architectural concepts effectively realize the targeted qualities, and the system demonstrates the required level of evolvability through the systematic execution of change scenarios. The findings confirm that the proposed approach provides a well designed architecture of an evolvable CWS.

## 7.2 Discussion

The current project makes a significant contribution to the design of an evolvable CWS. The contribution spans from the requirement identification (workflow configurability, CWS adaptability to external variability, and deployment flexibility) to the design of the system (UFs, adaptors, message broker, execution engine, and editor), proof-of-concept implementations, and verification through change scenarios and qualities. While existing approaches address individual aspects of these stages (e.g., BPMN for workflow structuring), the proposed solution offers a more integrated and generalizable design, explicitly tailored for clinical workflow variability and evolution. Additionally, the concepts used for applying each quality into the CWS have been presented, which to our knowledge has not been addressed as comprehensively in past studies. Furthermore, a clear design methodology for designing an evolvable CWS has been presented, which can be applied to other related studies as well. Lastly, the boundaries of the presented system design have also been discussed.

The change scenarios and qualities evaluated in the study have shown that the system design is capable of handling all the identified evolving requirements without requiring code-level changes, making the goal of the current study successful. Specifically, the system supports workflow reconfiguration through the use of configuration files to define workflows, a workflow execution engine for their interpretation and execution, and UF services that integrate functionality into the workflows. Regarding CWS adaptability, achieved through the use of adaptors, the system has demonstrated that by enabling the appropriate adaptor type component, the system can adapt to the external systems available in each hospital environment without code-level modifications. Furthermore, by leveraging the modularity of its components and its ability to execute multiple workflows concurrently, using identifiers to differentiate between them, the system can be deployed in both centralized and per-patient models. This flexibility is further supported by the system's ability to scale each component independently as workload demands increase.

Comparing the current study's results with previous works, there are two main contributions to designing an evolvable CWS. Firstly, introducing UFs makes workflow modeling extensible, scalable and easily maintainable. Previous studies did not focus on clearly separating functionality from workflow definitions, as their design proposals remained at very abstract level, typically only suggesting that workflows should be defined in configuration files and executed by an engine. In contrast, the current study established a strict separation between the functionality executed and the workflow definition, where otherwise such functionality would have been embedded as code snippets into the workflow definition. This earlier approach would lead to poorly maintainable workflows, especially as the number of workflows grows, and would lack a clear separation of concerns between workflow definition, task execution, and external system interaction. Secondly, this study explicitly addressed a key limitation of previous works, which often assumed standardization across external systems. In practice, hospital environments exhibit high variability in the types and interfaces of external systems. The proposed design overcomes this by introducing adaptors that isolate this variability, enabling the system to integrate with different external systems without requiring code-level modifications. This capability is essential for achieving the adaptability required of an evolvable CWS.

The implications of providing an evolvable CWS are significant. Such a system offers tangible

benefits to clinical workflow system providers, their engineers, and end users such as caregivers. For providers, the proposed design enables a single product release to serve multiple clients by allowing workflow customization and activation of relevant adaptors per hospital, all without requiring code-level changes, although workflow testing and (syntax and semantic) validation may still be required to ensure correct behavior. Furthermore, the system's maintainability, extensibility, and integrability reduce the engineering effort needed for system maintenance or evolution. On the other hand, caregivers benefit from faster delivery of tailored workflows at lower cost. Notably, the system is not confined to a clinical context; it can be adapted to other sectors with similar requirements, such as customer care, by replacing the UF services with domain-specific functionality and the adaptor services with interfaces to domain-specific external systems types. The proposed architecture demonstrates a generalizable design principle for building evolvable systems through the explicit separation of behavior (workflow definitions), function (UF services), and data (patient-specific context during workflow instantiation). This cross-domain applicability enhances the value and versatility of the design.

Regarding the system boundaries, an extensive analysis has already been presented in section 6.4. In summary, the identified system boundaries of the design were not significant and could be mitigated with explained mitigation strategies. The rest of the identified boundaries were about the proof of concept implementation which can be addressed through future development. Lastly, a consideration not yet discussed is that while the PESTEL-based requirement analysis provided valuable insights into external trends, it cannot guarantee completeness in identifying all future requirements. While this analysis was informed by literature reviews, brief market analysis, and expert discussions, it remains an approximation, and unforeseen requirements may still arise over time.

An important discussion point regarding workflow reconfiguration is that, beyond ensuring syntactic validity according to the selected workflow modeling language, the system must also verify the semantic correctness of the resulting clinical workflow. Even if reconfiguration actions comply with syntax rules, a sequence of valid edits may still result in a workflow that is not clinically meaningful. Achieving semantic validation is more complex than enforcing syntax, but it is equally, if not more, important in a clinical context. Therefore, while not yet implemented in the proof-of-concept, the proposed architecture supports future integration of techniques in the workflow editor to ensure that workflows remain both syntactically correct and semantically valid. Examples include embedding clinical constraints or validation rules (e.g., disallowing specific task orders), integrating domain-specific logic checks, or leveraging medical knowledge bases. The editor could invoke these validations during workflow authoring or before deployment to catch clinically invalid workflows.

Lastly, future work needs to be conducted on various aspects of the proposed system and, more broadly, on evolvability assessment. First, regarding workflow reconfiguration, although the proposed architecture supports semantic validation, further research is required to define and implement semantic verification techniques that ensure a reconfigured workflow remains not only syntactically correct according to the modeling language but also clinically valid. Second, the proof-of-concept implementation should be extended to further evaluate mixed deployment scenarios and assess the effectiveness of interactions between different CWS deployments. Finally, regarding the evaluation of system qualities, the introduction of new quantitative techniques could significantly enhance the ease, reproducibility and consistency of assessing both individual quality attributes and overall system evolvability, in contrast to the more subjective scenario-based evaluation currently used.

# Chapter 8

# Conclusion

The current thesis project addressed the challenge of designing an evolvable CWS capable of adapting to changing clinical workflows, external system variability, and diverse deployment environments without code-level changes.

Clinical workflows are constantly evolving, driven by medical advances, regulatory changes, hospital-specific processes, and variations in available technology. Healthcare environments are highly heterogeneous, with hospitals using diverse external systems and IT infrastructures. Existing CWS solutions often lack explicit decoupling of functionality from workflow definitions and require either continuous workflow maintenance or code-level changes to adapt to external systems. This results in high maintenance costs, the need to maintain different product releases for each hospital, slow adaptation to new requirements, and difficulties in deploying systems across multiple hospitals. To meet evolving healthcare needs, an evolvable CWS that addresses these issues without requiring code-level changes is essential.

A structured approach has been followed to address this problem. Firstly, an extensive literature review was conducted to understand the context and existing solutions. Then, the current and foreseeable requirements of the CWS were identified, and a set of change scenarios was constructed to operationalize these requirements. Based on the requirements, the key qualities were selected to guide the system design. The system design was then proposed, including several sequence diagrams to explain its behavior, covering both the requirements and the trade-offs encountered during the design process. A proof of concept was implemented, and the constructed change scenarios were demonstrated on the implementation, proving that the system fully satisfies them. The selected qualities were evaluated and found to be correctly integrated into the system design achieving the system goals. Thus, the evaluation criteria, based on change scenarios and qualities, were defined and assessed. Finally, the current system's design boundaries, its contributions, and directions for further work were discussed. The results of this work demonstrate that the proposed system design effectively meets the targeted requirements and achieves the intended level of evolvability.

The proposed system design successfully covers the requirements identified for workflow reconfiguration, system adaptation to external system variability, and deployment flexibility, all without requiring code-level changes. This is enabled by decoupling the workflow definition from the functionality to be executed through the use of UFs, by isolating interoperability concerns within the adaptors, and by decoupling UFs and adaptors through the use of a message broker. This system design integrates various qualities such as workflow configurability, system flexibility, adaptability, and integrability, as well as other related qualities that contribute to overall evolvability as defined by the requirements.

In summary, this thesis presents a novel modular architecture for evolvable CWS, introduces the concept of UFs for modular task execution, and provides adaptors for system decoupling from external system variability. It also offers a design methodology based on change scenarios and targeted qualities. The research questions posed at the start of this study were fully addressed through system design, proof-of-concept implementation, and scenario-based evaluation. Identified

boundaries of the proposed system do not pose significant limitations as they all can be mitigated using appropriate discussed strategies. However, while the proposed architecture is complete and, with the proposed components, addresses all identified requirements without posing any significant limitations, its current implementation remains a proof-of-concept. Further work is needed to possibly enhance aspects such as dynamic adaptor mechanisms, semantic clinical workflow validation, multi-tenant deployment support, and access control integration. Ultimately, this work provides a solid proposal for building CWS solutions that are not only technically robust but also capable of evolving alongside the dynamic nature of clinical environments.

# Bibliography


[1] M. Tanzini, J.I. Westbrook, S. Guidi, et al. Measuring clinical workflow to improve quality and safety. In L. Donaldson, W. Ricciardi, S. Sheridan, et al., editors, *Textbook of Patient Safety and Clinical Risk Management*, chapter 28, page Available from: https://www.ncbi.nlm.nih.gov/books/NBK585597/. Springer, Cham (CH), 2021. Published online: 2020 Dec 15. doi:10.1007/978-3-030-59403-9_28. 1

[2] Mustafa Ozkaynak, Kim Unertl, Sharon Johnson, Juliana Brixey, and Saira N. Haque. *Clinical Workflow Analysis, Process Redesign, and Quality Improvement*, pages 103–118. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-030-93765-2_8. 2, 11, 19

[3] Health Quality BC. Bc sepsis network. https://healthqualitybc.ca/improve-care/sepsis/bc-sepsis-network/. 1

[4] Health Quality BC. Emergency Department Sepsis Guidelines 2022, 2022. URL: https://healthqualitybc.ca/improve-care/sepsis/emergency-department-sepsis-guidelines/. 2, 3, 11

[5] Yousef Mehdipour and Hamideh Zerehkafi. Hospital information system (his):at a glance. *Asian Journal of Computer Science and Information Technology*, 01:2321–5658, 08 2013. 2, 11, 13, 15

[6] MO Akanbi, AN Ocheke, PA Agaba, and et al. Use of electronic health records in sub-saharan africa: Progress and challenges. *J Med Trop*, 14(1):1–6, 2012. 3

[7] Oluyemi Adetoyi and Abdulhameed Raji. Electronic health record design for inclusion in sub-saharan africa medical record informatics. *Scientific African*, 7:e00304, 02 2020. doi:10.1016/j.sciaf.2020.e00304. 3

[8] Lesley Dornan, Kanokporn Pinyopornpanish, Wichuda Jiraporncharoen, Ahmar Hashmi, Nisachol Dejkriengkraikul, and Chaisiri Angkurawaranon. Utilisation of electronic health records for public health in asia: A review of success factors and potential challenges. *BioMed Research International*, 2019:1–9, 07 2019. doi:10.1155/2019/7341841. 3

[9] P. Daniel Borches and G. Maarten Bonnema. A3 architecture overviews : focusing architectural knowledge to support evolution of complex systems. In *20th Annual International Symposium of the International Council on Systems Engineering, INCOSE 2010*, 20th Annual International Symposium of the International Council on Systems Engineering, INCOSE 2010, pages 354–369, 2010. 20th Annual International Symposium of the International Council on Systems Engineering, INCOSE 2010, INCOSE 2010 ; Conference date: 12-07-2010 Through 15-07-2010. 5, 6, 7, 8, 9, 10, 15

[10] P. Daniel Borches and G. Maarten Bonnema. Coping with system evolution - experiences in reverse architecting as a means to ease the evolution of complex systems. In *19th Annual International Symposium of the International Council on Systems Engineering (INCOSE 2009) in conjunction with the 3rd Asia-Pacific Conference on Systems Engineering APCOSE 2009*, pages 955–969, 2009. 19th Annual International Symposium of the International Council on

Systems Engineering, INCOSE 2009, INCOSE 2009 ; Conference date: 20-07-2009 Through 23-07-2009. doi:10.1002/j.2334-5837.2009.tb00994.x. 5, 6, 7, 8, 9, 15

[11] Brendan P. Sullivan, Monica Rossi, and Sergio Terzi. A review of changeability in complex engineering systems. *IFAC-PapersOnLine*, 51(11):1567–1572, 2018. 16th IFAC Symposium on Information Control Problems in Manufacturing INCOM 2018. URL: https://www.sciencedirect.com/science/article/pii/S2405896318313971, doi:10.1016/j.ifacol.2018.08.273. 5, 6, 15

[12] Eun Suk Suh, Michael R. Furst, Kenneth J. Mihalyov, and Olivier L. de Weck. Technology infusion: An assessment framework and case study. 2008. URL: https://api.semanticscholar.org/CorpusID:110304993. 5, 8, 15

[13] E. Halilović. Proposing a method for improved decision making on extensibility in system architectures, April 2024. 5, 7, 8, 15

[14] Boyang Zhang. Evolvability in system architecture: Methodology and mechanisms. Master's thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, August 2024. 5, 7, 8, 9, 10, 15

[15] Pierre America, P. Laar, G. Muller, Teade Punter, Nico Rooijen, Joland Rutgers, and David Watts. *Architecting for Improved Evolvability*, pages 21–36. 10 2010. doi:10.1007/978-90-481-9849-8_2. 5, 7, 8

[16] Adam Ross and Donna Rhodes. Using natural value-centric time scales for conceptualizing system timelines through epoch-era analysis. *INCOSE International Symposium*, 18, 06 2008. doi:10.1002/j.2334-5837.2008.tb00871.x. 5, 9, 15

[17] Bente Anda. Assessment of software system evolvability. pages 71–74, 09 2007. doi:10.1145/1294948.1294966. 5, 9, 15

[18] Edoardo F. Colombo, Gaetano Cascini, and Olivier L. de Weck. Classification of change-related ilities based on a literature review of engineering changes. *Journal of Integrated Design and Process Science*, 20(4):3–23, 2017. arXiv:https://doi.org/10.3233/jid-2016-0019, doi:10.3233/jid-2016-0019. 5, 6, 10, 15

[19] Ernst Fricke and Armin Schulz. Design for changeability (dfc): Principles to enable changes in systems throughout their entire lifecycle. *Systems Engineering*, 8, 12 2005. doi:10.1002/sys.20039. 5, 7, 8, 9, 10, 15

[20] Arnold B. Urken, Arthur "Buck" Nimz, and Tod M. Schuck. Designing evolvable systems in a framework of robust, resilient and sustainable engineering analysis. *Advanced Engineering Informatics*, 26(3):553–562, 2012. Evolvability of Complex Systems. URL: https://www.sciencedirect.com/science/article/pii/S1474034612000535, doi:10.1016/j.aei.2012.05.006. 5, 6, 10, 15

[21] Rick Steiner. 4.1.3 system architectures and evolvability: Definitions and perspective. *INCOSE International Symposium*, 8(1):137–141. URL: https://incose.onlinelibrary.wiley.com/doi/abs/10.1002/j.2334-5837.1998.tb00020.x, arXiv:https://incose.onlinelibrary.wiley.com/doi/pdf/10.1002/j.2334-5837.1998.tb00020.x, doi:10.1002/j.2334-5837.1998.tb00020.x. 5, 7, 8

[22] Jianxi Luo. A simulation-based method to evaluate the impact of product architecture on product evolvability. *Research in Engineering Design*, 26(4):355–371, 2015. doi:10.1007/s00163-015-0202-3. 5, 9, 15

[23] Hanfei Wang, Yuya Mitake, Yusuke Tsutsui, Salman Alfarisi, and Yoshiki Shimomura. A requirement analysis method for the design of the upgradable product-service system. *Procedia CIRP*, 119:402–407, 2023. The 33rd CIRP Design Conference. URL: https://www.sciencedirect.com/science/article/pii/S2212827123004882, doi:10.1016/j.procir.2023.01.006. 5, 8, 15

[24] Muztoba Ahmad Khan and Thorsten Wuest. Towards a framework to design upgradable product service systems. *Procedia CIRP*, 78:400–405, 2018. 6th CIRP Global Web Conference – Envisaging the future manufacturing, design, technologies and systems in innovation era (CIRPe 2018). URL: https://www.sciencedirect.com/science/article/pii/S2212827118312551, doi:10.1016/j.procir.2018.08.326. 5, 6, 8, 10

[25] Scott Selberg and Mark Austin. Toward an evolutionary system of systems architecture. *INCOSE International Symposium*, 18, 06 2008. doi:10.1002/j.2334-5837.2008.tb00863.x. 5, 7

[26] Justus Bogner, Alfred Zimmermann, and Stefan Wagner. Towards an evolvability assurance method for service-based systems. In M. Fazio and W. Zimmermann, editors, *Advances in Service-Oriented and Cloud Computing - Workshops of ESOCC 2018, Revised Selected Papers*, Communications in Computer and Information Science, pages 131–139. Springer Science and Business Media Deutschland GmbH, 2020. 7th European Conference on Service-Oriented and Cloud Computing, ESOCC 2018 ; Conference date: 12-09-2018 Through 14-09-2018. doi:10.1007/978-3-030-63161-1_10. 5, 8, 9, 15

[27] Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. Assuring the evolvability of microservices: Insights into industry practices and challenges. 10 2019. doi:10.1109/ICSME.2019.00089. 5, 8, 9, 15

[28] P. Daniel Borches and Gerrit Maarten Bonnema. On the origin of evolvable systems; evolvability or extinction. 2008. URL: https://api.semanticscholar.org/CorpusID:5792598. 5, 9

[29] David Lowe, John Leaney, and David Rowe. Defining systems evolvability - a taxonomy of change. In *Proceedings of the IEEE International Conference on Engineering of Computer-Based Systems*, page 45, Los Alamitos, CA, USA, April 1998. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/ECBS.1998.10027, doi:10.1109/ECBS.1998.10027. 5, 6, 7, 10

[30] Paolo Falcarin, Patricia Lago, and Maurizio Morisio. Dynamic architectural changes for distributed services. 01 2003. 5, 7, 8, 15

[31] R.L. Krikhaar. Reverse architecting approach for complex systems. In *1997 Proceedings International Conference on Software Maintenance*, pages 4–11, 1997. doi:10.1109/ICSM.1997.624225. 5, 7, 8, 9, 15

[32] Vojtech Huser, Luke Rasmussen, Ryan Oberg, and Justin Starren. Implementation of workflow engine technology to deliver basic clinical decision support functionality. *BMC medical research methodology*, 11:43, 04 2011. doi:10.1186/1471-2288-11-43. 11, 13, 15, 37

[33] Silvia Quaglini, Mario Stefanelli, Angela Cavallini, Giuseppe Micieli, Claudio Fassino, and Claudio Mossa. Guideline-based careflow systems. *Artificial Intelligence in Medicine*, 20(1):5–22, August 2000. doi:10.1016/s0933-3657(00)00050-6. 11, 12, 13, 15, 26

[34] Simona Ferrante, Stefano Bonacina, Giuseppe Pozzi, Francesco Pinciroli, and Sara Marceglia. A design methodology for medical processes. *Applied Clinical Informatics*, 7:191–210, 03 2016. doi:10.4338/ACI-2015-08-RA-0111. 11, 15, 26

[35] Peter J. Kennedy, Colin M. Leathley, and Carmel F. Hughes. Clinical practice variation. *Medical Journal of Australia*, 193(S8):S97–S99, October 2010. doi:10.5694/j.1326-5377.2010.tb04021.x. 11, 15, 26

[36] Silvana Quaglini, Mario Stefanelli, Giordano Lanzola, Vincenzo Caporusso, and Silvia Panzarasa. Flexible guideline-based patient careflow systems. *Artificial intelligence in medicine*, 22:65–80, 05 2001. doi:10.1016/S0933-3657(00)00100-7. 11

[37] J. Tummers, H. Tobi, C. Catal, et al. Designing a reference architecture for health information systems. *BMC Medical Informatics and Decision Making*, 21:210, 2021. doi:10.1186/s12911-021-01570-2. 11, 13, 15

[38] Peter Dadam, Manfred Reichert, and Klaus Kuhn. Clinical workflows — the killer application for process-oriented information systems? In Witold Abramowicz and Maria E. Orlowska, editors, *BIS 2000*, pages 36–59, London, 2000. Springer London. 11, 13, 15, 37

[39] E. C. de Carvalho, M. K. Jayanti, A. P. Batilana, et al. Standardizing clinical trials workflow representation in uml for international site comparison. *PLoS One*, 5(11):e13893, 2010. Published 2010 Nov 9. doi:10.1371/journal.pone.0013893. 11, 13

[40] Mor Peleg and Peter Haug. Chapter 10 - guidelines and workflow models. In Robert A. Greenes and Guilherme Del Fiol, editors, *Clinical Decision Support and Beyond (Third Edition)*, pages 309–348. Academic Press, Oxford, third edition edition, 2023. URL: https://www.sciencedirect.com/science/article/pii/B9780323912006000085, doi:10.1016/B978-0-323-91200-6.00008-5. 11, 12, 13, 15

[41] Mor Peleg. Computer-interpretable clinical guidelines: A methodological review. *Journal of Biomedical Informatics*, 46(4):744–763, 2013. URL: https://www.sciencedirect.com/science/article/pii/S1532046413000841, doi:10.1016/j.jbi.2013.06.009. 11, 13, 15

[42] Emma Carter and Fiona Mcgill. The management of acute meningitis: an update. *Clinical Medicine*, 22:396–400, 09 2022. doi:10.7861/clinmed.2022-cme-meningitis. 11

[43] Gilbert Habib, Patrizio Lancellotti, Manuel J. Antunes, et al. 2015 esc guidelines for the management of infective endocarditis: The task force for the management of infective endocarditis of the european society of cardiology (esc). *European Heart Journal*, 36(44):3075–3128, 2015. Endorsed by: European Association for Cardio-Thoracic Surgery (EACTS), the European Association of Nuclear Medicine (EANM). doi:10.1093/eurheartj/ehv319. 11

[44] Michael T Fitch and Diederik van de Beek. Emergency diagnosis and treatment of adult meningitis. *The Lancet Infectious Diseases*, 7(3):191–200, 2007. URL: https://www.sciencedirect.com/science/article/pii/S1473309907700506, doi:10.1016/S1473-3099(07)70050-6. 11

[45] Stavros V. Konstantinides, Guy Meyer, Cecilia Becattini, et al. 2019 esc guidelines for the diagnosis and management of acute pulmonary embolism developed in collaboration with the european respiratory society (ers). *European Heart Journal*, 41(4):543–603, 2020. doi:10.1093/eurheartj/ehz405. 11

[46] O. G. Mustafa, M. Haq, U. Dashora, E. Castro, K. K. Dhatariya, and Joint British Diabetes Societies (JBDS) for Inpatient Care Group. Management of hyperosmolar hyperglycaemic state (hhs) in adults: An updated guideline from the joint british diabetes societies (jbds) for inpatient care group. *Diabetic Medicine*, 40(3):e15005, 2023. doi:10.1111/dme.15005. 11

[47] Kidney Disease: Improving Global Outcomes (KDIGO) CKD Work Group. Kdigo 2024 clinical practice guideline for the evaluation and management of chronic kidney disease. *Kidney International*, 105(4S):S117–S314, 2024. doi:10.1016/j.kint.2023.10.018. 11

[48] R. A. Byrne, X. Rossello, J. J. Coughlan, et al. 2023 esc guidelines for the management of acute coronary syndromes. *European Heart Journal*, 44(38):3720–3826, 2023. Published correction appears in Eur Heart J. 2024 Apr 1;45(13):1145. doi: 10.1093/eurheartj/ehad870. doi:10.1093/eurheartj/ehad191. 11

[49] F.V. van Daalen, W.G. Boersma, E.M.W. van de Garde, F.M. van der Mooren, N. Roescher, J.A. Schouten, E. Sieswerda, D. Snijders, T. van der Veer, J.M. Prins, and W.J. Wiersinga. Management of community-acquired pneumonia in adults: the 2024 practice guideline from the dutch working party on antibiotic policy (swab) and dutch association of chest physicians (nvalt), 2024. URL: https://swab.nl/nl/exec/file/download/300. 11

[50] Miguel A. Laguna and Yania Crespo. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Science of Computer Programming*, 78(8):1010–1034, 2013. Special section on software evolution, adaptability, and maintenance & Special section on the Brazilian Symposium on Programming Languages. URL: https://www.sciencedirect.com/science/article/pii/S0167642312000895, doi:10.1016/j.scico.2012.05.003. 14

[51] Antônio Tadeu Azevedo Gomes, Artur Ziviani, Bruno Souza Pinto Marques Correa, Iuri Malinoski Teixeira, and Vinícius Macedo Moreira. Splice: a software product line for healthcare. In *Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium*, IHI '12, pages 721–726, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2110363.2110447. 14

[52] Robert Lindohf, Jacob Krüger, Erik Herzog, and Thorsten Berger. Software product-line evaluation in the large. *Empirical Software Engineering*, 26(2):30, 2021. doi:10.1007/s10664-020-09913-9. 14

[53] Clément Quinton, Rick Rabiser, Michael Vierhauser, Paul Grünbacher, and Luciano Baresi. Evolution in dynamic software product lines: Challenges and perspectives. 07 2015. 14

[54] Clément Quinton, Michael Vierhauser, Rick Rabiser, Luciano Baresi, Paul Grünbacher, and Christian Schuhmayer. Evolution in dynamic software product lines. *Journal of Software: Evolution and Process*, 33(2):e2293, 2021. doi:10.1002/smr.2293. 14

[55] Maíra Marques Samary, Jocelyn Simmonds, Pedro Rossel, and M. Bastarrica. Software product line evolution: a systematic literature review. *Information and Software Technology*, 105:190–208, 01 2019. doi:10.1016/j.infsof.2018.08.014. 14

[56] Carlos Cetina, Vicente Pelechano, Pablo Trinidad, and Antonio Ruiz-Cortés. An architectural discussion on dspl. pages 59–68, 01 2008. 14

[57] Kai Zheng, Raj M. Ratwani, and Julia Adler-Milstein. Studying workflow and workarounds in electronic health record-supported work to improve health system performance. *Annals of Internal Medicine*, 172(11 Suppl):S116–S122, June 2020. doi:10.7326/M19-0871. 19

[58] Melinda M. Davis, Rose Gunn, Maribel Cifuentes, Parinda Khatri, Jennifer Hall, Emma Gilchrist, C. J. Peek, Mindy Klowden, Jeremy A. Lazarus, Benjamin F. Miller, and Deborah J. Cohen. Clinical workflows and the associated tasks and behaviors to support delivery of integrated behavioral health and primary care. *The Journal of Ambulatory Care Management*, 42(1):51–65, Jan/Mar 2019. doi:10.1097/JAC.0000000000000257. 19

[59] Jiacun Wang and William Tepfenhart. *Petri Nets*, pages 201–243. 06 2019. doi:10.1201/9780429184185-8. 19

[60] PharmiWeb. Clinical workflow solutions market overview (2024-2035). https://www.pharmiweb.com/press-release/2025-02-17/clinical-workflow-solutions-market-overview-2024-2035, 2025. 25

[61] Dilip Nath. Artificial intelligence (ai) will transform the clinical workflow with the next-generation technology. https://www.healthtechmagazines.com/artificial-intelligence-ai-will-transform-the-clinical-workflow-with-the-next-generation-tec 2024. 25

[62] M. Scheffler and E. Hirt. Wearable devices for telemedicine applications. *Journal of Telemedicine and Telecare*, 11(Suppl 1):11–14, 2005. doi:10.1258/1357633054461994. 27

[63] Albert Y. Zomaya and Rizos Sakellariou, editors. *Software-defined systems*. Springer, 2020. URL: https://link.springer.com/book/10.1007/978-3-030-35945-3, doi:10.1007/978-3-030-35945-3. 29

[64] Donald J. Trump. Withdrawing the united states from the world health organization, January 2025. URL: https://www.whitehouse.gov/presidential-actions/2025/01/withdrawing-the-united-states-from-the-worldhealth-organization/. 30

[65] Youakim Badr. Service-oriented workflow. *JDIM*, 6:120–127, 01 2008. 37

[66] Marco A. Rodriguez and Rajkumar Buyya. A taxonomy and survey on scheduling algorithms for scientific workflows in iaas cloud computing environments. *Concurrency and Computation: Practice and Experience*, 29(8), 2017. doi:10.1002/cpe.4041. 37

[67] Jia Yu and Rajkumar Buyya. A taxonomy and survey of workflow management systems for grid computing. In *Journal of Grid Computing*, volume 3, pages 171–200. Springer, 2005. 37

[68] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015. URL: https://www.sciencedirect.com/science/article/pii/S0167739X14002015, doi:10.1016/j.future.2014.10.008. 37

[69] Amazon Web Services. What is pub/sub messaging? - pub/sub messaging explained, n.d. URL: https://aws.amazon.com/what-is/pub-sub-messaging/. 42

[70] Michael P Papazoglou and Willem-Jan Van Den Heuvel. Service-oriented architectures: approaches, technologies and research issues. *The computer journal*, 48(5):378–398, 2005. 43

[71] Seda Kul and Ahmet Sayar. A survey of publish/subscribe middleware systems for microservice communication. In *2021 5th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, pages 781–785, 2021. doi:10.1109/ISMSIT52890.2021.9604746. 43

[72] IBM Institute for Business Value. Healthcare interoperability, n.d. URL: https://www.ibm.com/thought-leadership/institute-business-value/report/healthcare-interoperability. 48

[73] Oracle Corporation. Interoperability in healthcare, n.d. URL: https://www.oracle.com/health/interoperability-healthcare/. 48

[74] IBM. Interoperability in healthcare, n.d. URL: https://www.ibm.com/think/topics/interoperability-in-healthcare. 48

[75] Health Level Seven International. Fhir - fast healthcare interoperability resources, n.d. URL: https://hl7.org/fhir/. 48

[76] Norman Wilde. A data flow approach to software architecture. *IEEE Software*, 9(4):56–62, 1992. 53

[77] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, Jim Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Bruce Berriman, Joseph Good, et al. Pegasus: A workflow management system for science automation. *Future Generation Computer Systems*, 25(5):528–540, 2009. 53

[78] Grafana Labs. Grafana tempo: Open source distributed tracing backend. `https://grafana.com/oss/tempo/`. 85

[79] OpenTelemetry Project. Opentelemetry. `https://opentelemetry.io/`. 85

[80] Grafana Labs. Grafana: The open observability platform. `https://grafana.com/`. 85

[81] Camunda. Camunda platform 8 documentation. `https://docs.camunda.io`. 103

# Appendix A

# Detailed Comparison of Workflow Modeling Alternatives

## A.1 Workflow Modeling Language (Detailed)

In the domain of workflow modeling, two major paradigms are commonly considered, the process-oriented (prescriptive) and data-oriented approaches. Process-oriented models, like those defined using the Business Process Model and Notation (BPMN), predefine all tasks, their execution order, and decision logic during the design time. The workflow proceeds through a structured sequence of steps stated in the workflow definition, which makes it a valid approach for predictable, repeatable processes that must be executed consistently every time. On the other hand, data-oriented models, such as those found in dataflow systems, focus on data driven execution. Tasks are triggered as soon as their required inputs become available. This paradigm is suited for scientific workflows, IoT systems, and streaming data processing, where the structure of the process is secondary to the availability and flow of data.

Case-oriented modeling approaches, an example of which is the Case Management Model and Notation (CMMN), fall between these two paradigms. While not purely data-driven, CMMN differentiates from the rigid sequencing of BPMN by allowing task activation to be determined dynamically at runtime based on the evolving state of case data, external events, or user decisions (declarative). It supports both declarative execution and optional control flow elements, making it appropriate for unpredictable processes where not all behavior can be predefined. In addition to BPMN and CMMN, other workflow modeling approaches exist, like Serverless Workflow (CNCF), AWS Step Functions, and Netflix Conductor. A high-level comparison of these modeling languages is provided in Table A.1. This table illustrates the characteristics (rows) of each modeling language (columns).

While this study includes an analysis of high-level workflow requirements and identifies the desired overall system behavior, it does not address low-level implementation details, as these lie outside the scope of the system design. Importantly, the goal of the proposed system architecture is to remain implementation-independent, and one of its key strengths is its flexibility. Regardless of which modeling approach is selected, the architecture can support all of them, as they share a standard underlying structure. Each of the approaches, including in the table above, involves an editor to define workflows using specific modeling rules, a configuration file to store the workflow definition, and an execution engine to interpret and execute the workflow. In addition, there is a mechanism to maintain workflow state and a user interface to visualize the workflow's progress and current status. Finally, the notion of workers is prevalent in all approaches as tasks can be linked with an implemented functionality to be executed.

Now, considering the boundaries and timeline of the project, the approach that offers the most complete and practical foundation for demonstrating the system's capabilities will be selected. One of the most critical requirements for delivering a proof of concept within the given constraints

is the availability of a ready-to-use, publicly available editor that enforces the necessary modeling rules and execution engine that is able to execute the workflows as developing custom tooling for modeling and executing workflows is beyond the project's scope. As shown in Table A.1, CNCF Serverless Workflow, AWS Step Functions, and Netflix Conductor present significant drawbacks for this specific use case. They also lack native support for human-in-the-loop tasks, which is essential in clinical workflows. These platforms are primarily designed for service orchestration in cloud-native systems and are not well-suited for modeling processes involving manual task execution. However, with additional implementation additions, they could also support manual tasks.

In summary, while several workflow modeling alternatives provide useful capabilities, the combination of modeling language standardization, mature graphical tooling, strong human-in-the-loop support, formal execution semantics, overall simplicity, and the points discussed in section 5.1 makes BPMN the most appropriate option for clinical workflows. However, as mentioned above, if another modeling language is deemed more suitable for the final system implementation, the architecture can fully support it. This analysis informed the selection of BPMN for the proof-of-concept implementation presented in chapter 5.

| **Feature** | **BPMN** | **CMMN** | **CNCF Serverless Workflow** | **AWS Step Functions** | **Netflix Conductor** |
|---|---|---|---|---|---|
| **Purpose** | Prescriptive, structured process modeling | Case-based processes with tasks triggered by data, events, or users | Event-driven, cloud-native workflows | AWS-native service orchestration | Microservice orchestration |
| **Standardized?** | ✓ OMG standard (ISO/IEC 19510:2013) | ✓ OMG standard | ✓ CNCF specification | ✗ Proprietary | ✗ Open-source, but not a formal standard |
| **Graphical Modeling** | ✓ Strong (e.g., Camunda Modeler) | ✓ Graphical case notation (e.g., Camunda CMMN) | - Limited (Kogito, VSCode plugin) | - AWS Console editor | - JSON UI or third-party tools |
| **Execution Semantics** | Formal, token-based with rich constructs (e.g., timers, gateways, tasks) | Declarative, event and condition driven based on evolving case context | Event and state driven execution using defined state transitions | State machine-based execution driven by events and transitions | Task queue based orchestration with external workers managing tasks independently |
| **Human-in-the-loop Support** | ✓ Strong (user tasks, human actors) | ✓ Strong (discretionary tasks, user-triggered stages) | ✗ Not designed for it | ✗ Not natively | ✗ Not natively |
| **Model Type** | Process-oriented (procedural, structured), XML Based | Case-oriented (event/data-driven), XML based | Hybrid (JSON/YAML) | Process-oriented (JSON) | Process-oriented (JSON) |
| **Structure** | Strict control flow, predefined sequence | Flexible, event- and context-driven | Moderate | Structured, state-based | Flexible but predefined |
| **Human Discretion** | Low: predefined task paths | High: tasks triggered at runtime by context or user | Moderate | Low | Low |
| **Typical Use Case** | Order fulfillment, invoice approval, employee onboarding, customer service workflows, Clinical Workflows | Case management, diagnostics, incident handling | Event-based orchestration in microservices | Backend service workflows | Microservice coordination in large-scale systems |

Table A.1: Comparison of Workflow Modeling Alternatives

# Appendix B

# Components Implementation

## B.1 Camunda Related Components Implementation

Camunda Modeler, the Zeebe Cluster, Elasticsearch, and the UI components (Tasklist and Operate) are used as provided by Camunda, without any customization. Even though the needed implementation concepts and functionality of these components has been explained, the official documentation provides detailed implementation information [81]. All components, except Camunda Modeler, which is not involved during the workflow execution, are deployed as Docker containers for convenient management and independence from the underlying hardware.

## B.2 UFs' Implementation

UFs have been implemented as independent services for this proof of concept. They are written in Java to leverage existing libraries for interacting with the Zeebe Client, which is also part of Camunda's core implementation. However, it should be noted that Camunda also supports additional programming languages for the worker's implementation. The decision to implement UFs as standalone services was made to highlight the benefits of this approach, particularly in terms of maintainability, scalability, and modular management of each UF. However, this approach may introduce extra overhead, as each UF is deployed as a separate service and, in this case, within its own container. To overcome this disadvantage, an alternative would be to group multiple UFs into fewer containers or deploy them all as a single container. In this case, the trade-off revolves around the desired level of independence between UFs, acceptable downtime when maintenance is needed, and other operational considerations. For the proof of concept, the overhead is not considered significant, however in the final implementation of the system, this should be carefully considered, and the approach should be selected based on the available resources and operational requirements.

More specifically, UFs are independent Spring Boot applications, each implementing a specific functionality. Each UF has a set of core functions that allow it to poll the Zeebe Client for available jobs, publish messages to the appropriate topics (to be handled by the adaptors via the RabbitMQ message broker), and receive responses by subscribing to the relevant topics. During workflow execution, the Zeebe Engine knows how to link the created job with the specific UF because, as shown in Figure 5.2, the name of the UF's function that handles the job is specified, allowing Zeebe Engine to create the job for the correct worker. Finally, UFs can respond to the Zeebe Engine via the Zeebe Client with the results of the executed job and mark the task as completed if the task is completed successfully. On top of this functionality, core pre- and post-conditions that do not need to differ across hospitals can be encapsulated in a UF as they are not considered a variation point that needs to be customized.

In addition to UFs that perform clinical tasks, a dedicated UF has been implemented to trigger workflow executions based on requests originating from external systems. This UF listens

for workflow execution requests on a specific topic, which are received and forwarded by the corresponding adaptor of the external system.

Finally, if a new UF needs to be added to the system, it must declare a unique function name (job type). Once it is deployed and registers with the message broker, it can be referenced in BPMN task definitions for execution.

## B.3 Message Broker Implementation

In the proof of concept, RabbitMQ serves as the publish/subscribe mechanism that facilitates asynchronous and decoupled communication between UFs and adaptors. It uses a topic exchange to route messages based on routing keys. Upon initialization, each UF and adaptor declares a queue and binds it to the exchange using a binding keys that specifies its topics of interest. These binding keys can be specific strings or patterns using wildcards like * (to match a single word) or  (to match multiple words). The topic exchange constructs an internal mapping of which queues are interested in which routing keys based on these bindings. The queues act as message buffers, holding messages until the subscribers consume them. When a message is published to the exchange with a routing key, for example *Temperature.Request*, the exchange compares this key against all declared binding keys. It then routes the message to every queue whose binding key matches the routing key, following the matching rules of the topic exchange.

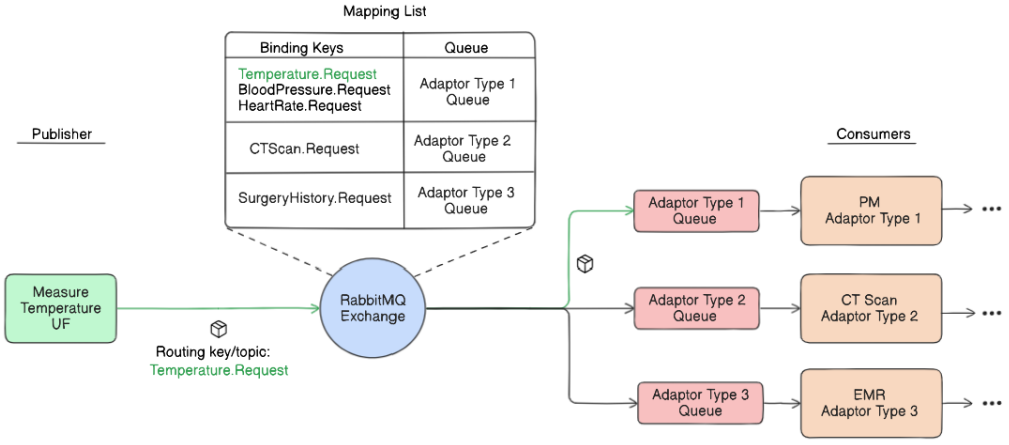



Figure B.1: Example of how RabbitMQ Publishes Messages

For example, Figure B.1 illustrates how the *MeasureTemperature UF* publishes a message with the routing key *Temperature.Request* to the exchange. While RabbitMQ supports multiple exchanges to separate concerns across different groups of routing keys, a single exchange is used in this proof of concept for simplicity. The exchange evaluates the message against the binding keys in the mapping list it has constructed during the initialization phase. As can be seen, the routing key matches with the binding key of only the *Adaptor Type 1 Queue* which receives the message. Other queues, such as the *Adaptor Type 2 Queue* and *Adaptor Type 3 Queue*, are bound to different keys and therefore do not receive this message. This example highlights how RabbitMQ routes messages only to relevant consumers based on the match between routing keys and binding keys.

In the previous example, only the one-way message flow from the UF to the adaptor was shown for simplicity. However, the same mechanism applies in the opposite direction, where adaptors can also publish messages toward the UFs for the responses of the requests received as has been shown already in the system behavioral diagrams in the previous chapter. An important point to clarify is how messages are associated with a specific workflow or patient. The topics or routing keys are used solely for routing purposes, to direct the message to the correct destination. All other contextual information, such as the patient ID, workflow process ID, or any other relevant

data, is encapsulated within the message body itself. Finally, RabbitMQ like UFs is deployed in a container as well.

## B.4 Adaptors' Implementation

The adaptors are implemented similarly to the UFs. They are independent services deployed in separate containers. They encapsulate functionality specific to the external systems they interact with. Like the UFs, adaptors are responsible for subscribing to routing keys related to the functionality of the corresponding external system they are dedicated to and setting up the appropriate queues and routing keys for messaging via RabbitMQ. Based on the incoming requests, they invoke different exposed external system services to execute and retrieve the needed information. Once the response from the external system has been received, they do the necessary information translation and publish the retrieved information to the exchange on the response routing key related to the routing key of the incoming message. For example, if the incoming message had the *Temperature.Request* as routing key, the adaptor will publish the response on the *Temperature.Response* to be received by the dedicated UF related to the temperature measurement and complete the job.

In this proof of concept, three different adaptors have been implemented: one for interacting with *PM (Type 1)*, one for *PM (Type 2)*, and one for the *Medical Record Database*. The first two adaptors implement functions for invoking different services provided by the corresponding PMs. They also implement endpoints that allow the respective PMs to send workflow execution trigger requests to the CWS. Finally, each adaptor maintains information that maps which PM is responsible for each patient, in order to determine whether the linked PM should handle any relevant request or not.

## B.5 External Systems' Implementation

For the system proof of concept implementation, a minimal implementation of a number of external systems have been implemented. Specifically, a server simulating a PM has been implemented, exposing a series of endpoints to serve requests related to vital signs measurements for a patient, such as temperature, heart rate, and other data needed to execute the Sepsis workflow discussed. The same PM implementation has been used for both PMs acting as external systems, however they are linked to different addaptors to simulate that the CWS can handle different types of external systems while leveraging the respective adaptors. Additionally, a database has been created to simulate the role of an EMR, providing patients' medical records, such as surgery history and other relevant information needed for sepsis workflow execution.

## B.6 Docker Container Deployment

| | Name | Image | Container ID | Port(s) | CPU (%) | Last started | Actions |
|---|---|---|---|---|---|---|---|
| | rabbitmq | rabbitmq:3-management | 13fdb5935794 | 8089:15672<br>Show all ports (2) | 0.46% | 5 hours ago | |
| | cws | - | - | - | 37.74% | 3 minutes ago | |
| | WorkflowRequestControllerUF-1 | workflow-request-controller | 6fb4f1c73d64 | 8015:8080 | 0.12% | 3 minutes ago | |
| | MeasureTempertureUF-1 | uf-temperature | b5864c80f5d5 | 8011:8080 | 0.13% | 3 minutes ago | |
| | CheckSurgeryHistoryUF-1 | uf-surgeryhistory | 3eb1e6734884 | 8009:8080 | 0.13% | 3 minutes ago | |
| | MeasureSystolicBloodPressureUF-1 | uf-sbp | c5679ba1c645 | 8010:8080 | 0.15% | 3 minutes ago | |
| | MeasureRespirationRateUF-1 | uf-respirationrate | 6c394a2782c8 | 8008:8080 | 0.12% | 3 minutes ago | |
| | MeasureMeanArterialPressureUF-1 | uf-map | c1e4b7f521f6 | 8007:8080 | 0.1% | 3 minutes ago | |
| | EvaluateImmuneCompromisationUF- | uf-immunecompromisation | 74658ff6e9ee | 8006:8080 | 0.13% | 3 minutes ago | |
| | MeasureHeartRateUF-1 | uf-heartrate | feeb96198707 | 8005:8080 | 0.12% | 3 minutes ago | |
| | DummyActionUF-1 | uf-dummyaction | fab14aadd700 | 8004:8080 | 0.1% | 3 minutes ago | |
| | CheckChronicIllnessUF-1 | uf-chronicillness | d859a375858a | 8003:8080 | 0.14% | 3 minutes ago | |
| | MeasureBloodPressureUF-1 | uf-bloodpressure | ada66a327933 | 8002:8080 | 0.13% | 3 minutes ago | |
| | RetrieveAgeUF-1 | uf-age | f3d3f4b4056b | 8001:8080 | 0.13% | 3 minutes ago | |
| | PatientMonitorAdaptorType2-1 | patient-monitor-adaptor-type-2 | e0defc1d0cca | 8014:8080 | 0.21% | 4 minutes ago | |
| | PatientMonitorAdaptorType1-1 | patient-monitor-adaptor-type-1 | c0d4db6168d4 | 8013:8080 | 0.16% | 4 minutes ago | |
| | PatientMedicalRecordAdaptor-1 | patient-medical-record-adaptor | 5d8a3b745c02 | 8012:8080 | 0.16% | 4 minutes ago | |
| | elasticsearch | elasticsearch/elasticsearch:7.17.0 | 5c10cb401d74 | 9200:9200<br>Show all ports (2) | 4.51% | 5 hours ago | |
| | zeebe | camunda/zeebe:8.0.0 | 3bf23ad89786 | 26559:26500<br>Show all ports (2) | 8.02% | 5 hours ago | |
| | tasklist | camunda/tasklist:8.0.0 | 2ed68d5806bb | 8082:8080 | 0.22% | 4 hours ago | |
| | operate | camunda/operate:8.0.0 | d48049c93c80 | 8081:8080 | 0.88% | 4 hours ago | |
| | Patient_DB_EMR | postgres:15 | 9a3ba79fbf83 | 5432:5432 | 0% | 1 hour ago | |
| | PM-Type2 | cws-monitor2 | 53dbe3bc7044 | 8091:8090 | 0.01% | 4 minutes ago | |
| | PM-Type1 | cws-monitor1 | 835006997a38 | 8090:8090 | 0.02% | 4 minutes ago | |

Figure B.2: Docker dashboard showing deployed and running CWS components' & external systems' containers in the proof-of-concept setup.

# Appendix C

# Evaluation

## C.1 Workflow Configurability

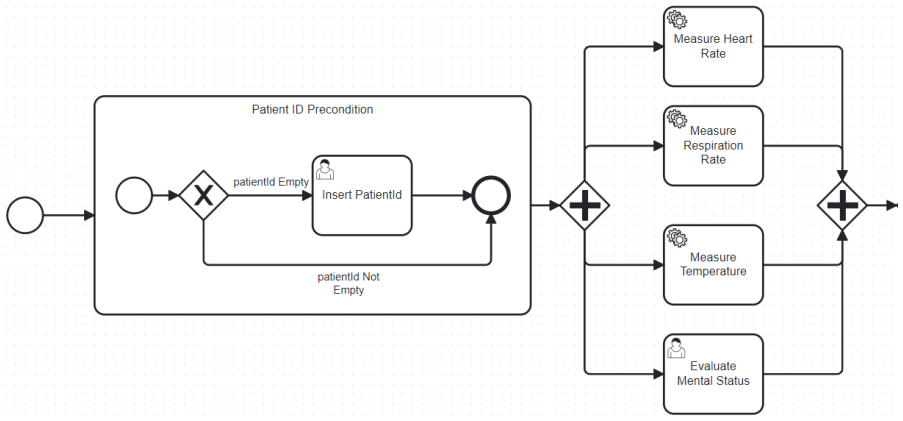



Figure C.1: Base workflow design with configurable precondition and measurement steps

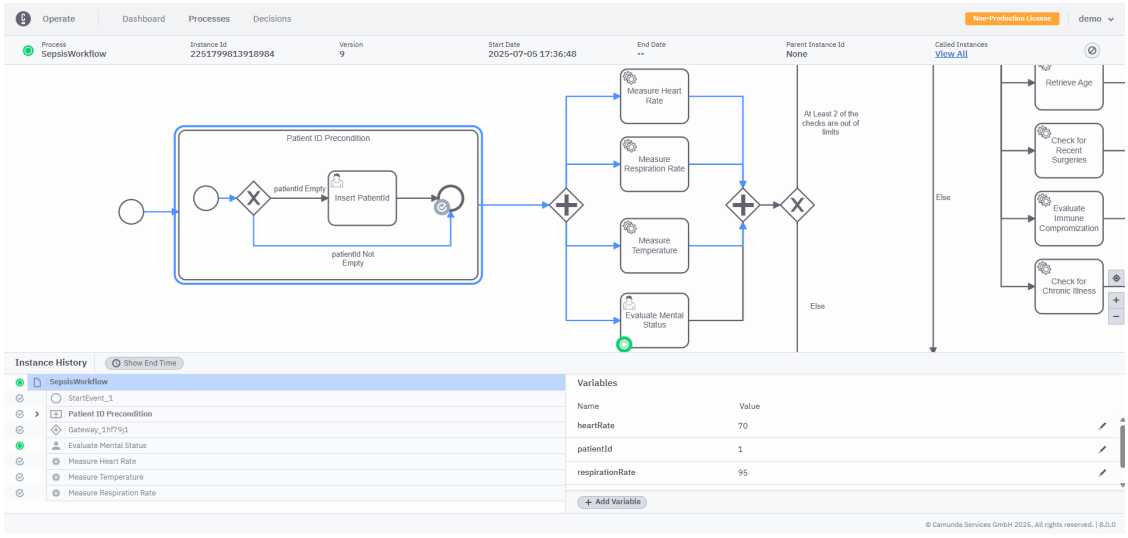



Figure C.2: Successful execution of the base workflow configuration

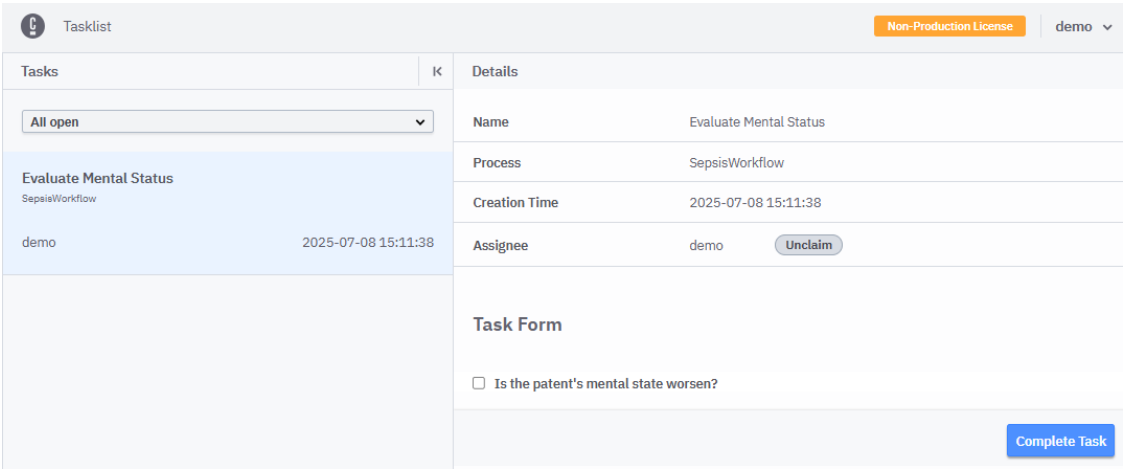



Figure C.3: *Evaluate Mental Status* user task