ISSN 2712-0554
Original Research

Heritage and Sustainable Development
Vol. 5, No. 2, September 2023, pp.253-279
https://doi.org/10.37868/hsd.v5i2.226

Toward a conceptual framework for designing sustainable cyber-physical system architectures: A systematic mapping study

Luisa Fernanda Restrepo Gutierrez [1*], Pablo Bernal Moreno [1], Elizabeth Suescún Monsalve [1], Jose Aguilar Castro [1,2,3], César Jesus Pardo Calvache [4]

[1] GIDITIC Research Group, EAFIT University, Medellín, Colombia
[2] CEMISID, University of the Andes, Mérida, Venezuela
[3] Dept. Automatic, University of Alcalá, Alcalá de Henares, Spain
[4] GTI Research Group, University of Cauca, Popayán, Cauca, Colombia

* Corresponding author E-mail lrestr61@eafit.edu.co

Received Jun. 1, 2023
Revised Sep. 8, 2023
Accepted Sep. 14, 2023

Abstract

Cyber-physical systems (CPS) represent devices whose components enable interaction between machines and processes. One of the biggest challenges of these systems today is the ability to adjust to changes at the time of execution as they are implemented in environments with a multidimensional complexity, this challenge is currently addressed from the design of the systems themselves by integrating sustainability. With this problem in mind, the present document describes a systematic mapping study of the literature with the goal of demonstrating the current panorama of the frameworks, designs, and/or models used at the time of initiating the development of a cyber-physical system. As a result, it has been concluded that there is a lack of guidelines to construct sustainable, and evolvable cyber-physical systems. To address these issues, a framework for designing sustainable CPS architectures is outlined.

© The Author 2023.
Published by ARDA.

Keywords: Cyber-physical systems, Framework, Design, Sustainability, Architecture, Architecture design

1. Introduction

"Cyber-physical systems (CPS) are integrations between computation and physical processes" [1]. These systems control the physical processes, and in turn, these processes affect the CPS algorithms. In comparison to traditional embedded systems, CPS are evolving to be more dynamic, modular, and scalable, increasing dependence on software to such an extent that today it is normal to speak of software-intensive cyber-physical systems [2]. For this reason, new-generation CPS come with great challenges in relation to software design and implementation, in part due to the immense diversity of the platforms on which they must be implemented and the immense diversity in their applicability [3]. Another important issue for CPS is sustainability, due to the physical nature of these systems. A sustainable software design must account for component obsolescence and upgrades, as well as allow for the replacement and introduction of new components in a deployed system with minimal impact on the existing applications [4]. Traditional approaches such as designing for the worst-case scenario will not be useful with these new challenges and the new requirements are security, reliability, sustainability, efficiency, and predictability of the software and the system. Given these challenges, a systematic

This work is licensed under a Creative Commons Attribution License (https://creativecommons.org/licenses/by/4.0/) that allows others to share and adapt the material for any purpose (even commercially), in any medium with an acknowledgement of the work's authorship and initial publication in this journal.

mapping study (SMS) was carried out with the main objective of identifying related works and from its analysis establishing the main models, frameworks, and/or architectures to additionally propose a framework for designing sustainable CPS architectures that help to solve the problems raised where sustainability is addressed.

Apart from this introductory section, the study is structured in the following manner: In Section 2, we describe the developed research process and specify the study's research topics. Section 3 provides the findings and solutions to the posed questions along with a discussion and outlines a framework with which to address the principal issues identified regarding sustainable CPS, and finally, Section 4 outlines the conclusions and next work.

2. Research method

A systematic mapping study (SMS) of literature is a method used to identify, assess, and synthesize current knowledge on the subject issue. The present SMS was carried out following the protocols and methods established in [5], [6], and different tools such as Parsifal (https://parsif.al) and Microsoft Excel were used to manage the selected works. The search strategy is described in the following subsections (see Figure 1). Provide enough information to allow the work to be duplicated. Any previously published methods should be acknowledged with reference. While only relevant alterations must be stated.

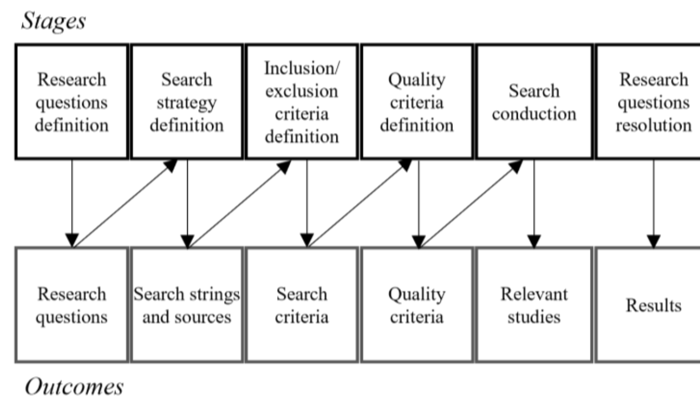

Figure 1. Planning activities carried out in the SMS, taken from [5], [6]

2.1. Research questions definition

This SMS's main objective was to evaluate the state of the art in designing CPS architectures with a certain emphasis on sustainability. With this objective in mind, five research questions were formulated to inquire about strategies, methodologies, and/or frameworks used for both the design of software architectures and CPS architectures. Additionally, we aimed to establish a relationship between architecture definitions and sustainability:

- Q1. What kind of strategies, methods, and/or frameworks are used for the design of software architectures?
- Q2. What types of modeling strategies and patterns exist to represent software architectures?
- Q3. What kind of strategies, methods, and/or frameworks are used for the design of cyber-physical system architectures?
- Q4. What types of modeling strategies and patterns exist to represent cyber-physical systems?
- Q5. Is there a link between the definition of architecture and sustainability?

2.2. Search strategy definition

We established three search strings containing specific terms and sentences for the search process. Subsequently, we fine-tuned the search strings by incorporating additional keywords found in relevant studies related to our research area. The keyword list that was utilized to locate an answer to the research queries is shown in the definitive search strings presented in Table 1.

For each of the selected databases, this process was narrowed down and further refined: ACM, Google Scholar, Scopus, ISI Web of Science, IEE Digital Library, Elsevier and Springer as shown in Table 2.

Table 1. Definitive Search Strings

Research Questions	Search Strings
Q1&Q2	(Software) AND (framework OR architecture OR structure OR model OR frameworks) AND (methodologies OR approach OR methodology OR method OR concept) AND (agroindustry OR agricultural OR agricultural industry OR rural industry)
Q3&Q4	(cyber-physical OR embedded OR IoT OR Internet of things) AND (Systems) AND (self-adaptive OR adaptable OR flexible OR adaptivity) AND (framework OR architecture OR structure OR model OR frameworks) AND (methodologies OR approach OR methodology OR method OR concept) AND (agroindustry OR agricultural OR agricultural industry OR rural industry)
Q5	(framework OR architecture OR structure OR model OR frameworks) AND (Sustainability)

Table 2. Search strings utilized for each database:

Data Base	Search String
ACM	(Cyber-physical software OR Embedded software OR Software Systems) AND (Framework OR Architecture OR Structure OR Model OR Frameworks) AND (Self adaptive OR Adaptable OR Flexible OR Adaptivity) AND (Methodologies OR Approach OR Methodology OR Method OR Concept)
Google Scholar	allintitle: Systems Software Framework OR Architecture OR Methodologies OR Model ”Cyber physical” allintitle: Adaptive Framework OR Architecture OR Methodologies OR Model ”Cyber physical” allintitle: Adaptive Framework OR Architecture OR Methodologies OR Model ”Embedded systems” allintitle: software Framework OR Architecture OR Methodologies OR Model ”Embedded systems” allintitle: architecture sustainable ”cyber physical” allintitle: architecture sustainable ”software” allintitle: Systems Software Framework OR Architecture OR Methodologies OR Model
Scopus	(”Cyber-physical software” OR ”Embedded software” OR ”Software Systems”) AND (”Framework” OR ”Architecture” OR ”Structure” OR ”Model”) AND (”Self adaptive” OR ”Flexible” OR ”Adaptivity”) AND (”Methodologies” OR ”Approach” OR ”Methodology” OR ”Concept”)
ISI Web of Science	TI=(Cyber-physical Architecture*) OR TI(Software Architecture*)
IEE Digital Library	(”Cyber-physical software” OR ”Embedded software” OR ”Software Systems”) AND (”Framework” OR ”Architecture” OR ”Structure” OR ”Model”) AND (”Self adaptive” OR ”Flexible” OR ”Adaptivity”) AND (”Methodologies” OR ”Approach” OR ”Methodology” OR ”Concept”)
Elsevier	(Cyber-physical software OR Embedded software OR Software Systems) AND (Framework OR Architecture OR Structure OR Model OR Frameworks) AND (Self adaptive OR Adaptable OR Flexible OR Adaptivity) AND (Methodologies OR Approach OR Methodology OR Method OR Concept)

2.3. Inclusion/exclusion criteria definition

The inclusion and exclusion criteria were established in Table 3 y 4. The criteria were devised to identify the most pertinent papers that could provide answers to the research questions while excluding those that are not relevant to this field or do not contribute to solving the research inquiries. Conversely, articles meeting any of the exclusion criteria listed in Table 4 were disregarded.

Table 3. Inclusion criteria

Data Base	Search String
IC1	Articles, chapters, dissertations, books, and lectures published since 2010
IC2	Articles, dissertations, book chapters and conferences presenting methods, models, and representations of cyber-physical and software systems.
IC3	Articles, chapters, dissertations, book, and conferences with titles related to software architectures
IC4	Articles, chapters, dissertations, book, and conferences whose title, abstracts, and conclusions contain one or more keywords.

Table 4. Exclusion Criteria

Data Base	Search String
EC1	Articles, dissertations, book chapters and conferences whose domain is a subject other than software engineering or development.
EC2	Articles, dissertations, book chapters and conferences published before 2010.
EC3	Duplicate articles, book chapters, dissertations, and conferences.
EC4	Articles, dissertations, book chapters and conferences whose texts are not available or accessible.

2.4. Quality criteria definition

A questionnaire was developed to gauge the quality of the selected studies. It employed a scoring system with three values: 1 for 'Yes,' 0.5 for 'Partially,' and 0 for 'No.' These values were carefully calibrated to ensure that studies with negative scores were not disregarded for future research. The evaluation process involved assessing the information gathered from each database search, including the title, abstract, and keywords, to determine the inclusion of studies among the relevant ones. This evaluation was conducted by the authors, who then thoroughly analyzed the resulting studies to choose those who satisfied at least one of the listed criteria outlined in Table 5.

Table 5. Quality assessment of studies according to inclusion criteria

Ref	IC1	IC2	IC3	IC4	Total
[7]	1	1	0.5	1	3.5
[8]	1	0	1	1	3
[9]	1	1	1	1	4
[10]	1	1	1	1	4
[11]	1	0.5	1	1	3.5
[12]	1	1	0	1	3
[13]	1	1	1	1	4
[14]	1	1	0.5	1	3.5
[15]	1	0	1	1	3
[1]	1	0.5	0	1	2.5
[16]	1	1	0	1	3
[17]	1	1	1	1	4
[18]	1	1	1	1	4

Ref	IC1	IC2	IC3	IC4	Total
[19]	1	0	1	1	3
[20]	1	0	1	1	3
[21]	1	0	1	1	3
[22]	1	0.5	1	1	3.5
[2]	1	1	0.5	1	3.5
[23]	1	1	0.5	1	3.5
[24]	1	1	1	1	4
[25]	1	1	0.5	1	3.5
[4]	1	1	1	1	4
[26]	1	1	1	1	4
[27]	1	1	1	1	4
[28]	1	0	1	1	3
[29]	1	1	1	1	4
[30]	1	1	0	1	3
[31]	1	1	0	1	3
[32]	1	1	1	1	4
[33]	1	1	1	1	4
[34]	1	0	1	1	3
[35]	1	0	1	1	3
[3]	1	1	0.5	1	3.5
[36]	1	1	1	1	4
[37]	1	0.5	1	1	3.5

2.5. Search conduction

The data extraction strategy aims to maintain consistency across all selected studies by employing uniform data extraction criteria. This involves streamlining their classification using potential answers corresponding to each of the research questions, as outlined in Table 6.

Table 6. Classification Scheme

Research Question	Answers
Q1. What kind of strategies, methodologies and/or frameworks are used for the design of software architectures?	a. Software Architecture b. Methodology c. Software Design
Q2. What types of representations exist to represent software architectures?	a. Software Representations b. Software Systems
Q3. What kind of strategies, methodologies and/or frameworks are used for the design of cyber-physical system architectures?	a. Cyber-physical Architecture b. Methodology c. Cyber-physical Design
Q4. What types of representations exist to represent cyber-physical systems?	a. Cyber-physical Representations b. Cyber-physical Systems
Q5. Is there a link between the definition of architecture and sustainability?	a. Sustainable Architecture b. Sustainable Systems c. Adaptivity

Information from the chosen primary studies was collected and organized based on the following specifications: basic details (title, author, year), summary, and relevant aspects crucial for addressing the research questions. These pertinent aspects included definitions, characteristics, types, methods, models, frameworks, and

applications in both the private and the public sector. Table 7 lists the studies that were chosen, a total of 35. Initially, a context search and data collection were carried out using the databases mentioned in Table 2 with the strategy described in the search strategy section, as well as a search through other means (teachers, peers, etc.). Afterward, three iterations were conducted to fine-tune the method used to search each database. Table 8 displays the outcomes achieved following the execution of the search strings in each database, along with the quantity of studies that were gathered from various sources.

Regarding the quality criteria, each study's overall quality score is determined by the sum of scores obtained for each question, resulting in a value ranging from 0 to 6. Table 5 shows the findings of the studies' evaluation in accordance with the quality assessment questions. The studies selected were those with a score higher than 3. Figure 2 summarizes the study selection process with the corresponding values for each stage of the SMS, and results are described in the section that follows.

Table 7. Contribution of main studies

Ref	Q1	Q2	Q3	Q4	Q5
[7]	X	-	X	-	X
[8]	X	X	X	X	-
[9]	X	-	X	-	-
[10]	X	X	X	-	X
[11]	X	X	X	X	-
[12]	X	-	X	-	-
[13]	X	X	X	-	X
[14]	-	X	-	X	X
[15]	X	-	X	-	-
[1]	-	X	-	X	-
[16]	X	-	-	X	-
[17]	-	X	X	X	-
[18]	-	X	-	X	-
[19]	X	-	-	X	-
[20]	X	-	X	-	-
[21]	X	-	X	-	-
[22]	-	-	X	X	-
[2]	-	X	-	X	-
[23]	-	X	-	X	X
[24]	X	X	X	-	X
[25]	X	-	X	X	-
[4]	X	-	X	-	-
[26]	X	X	X	-	X
[27]	X	X	X	-	X
[28]	X	-	X	-	-
[29]	X	X	X	-	X
[30]	-	X	-	X	-
[31]	X	-	X	-	X
[32]	X	-	X	X	X
[33]	X	X	X	-	X
[34]	X	-	-	X	-
[35]	X	-	X	-	X
[3]	-	X	-	X	-
[36]	X	-	X	-	X
[37]	X	-	-	X	-

Table 8. Classification scheme

Source	Found	Pertinent	Duplicates	Pertinent without Access	Total
Other sources	20	18	0	0	18
ACM	39	16	10	2	4
Google Scholar	850	29	21	1	5
Scopus	40	20	10	7	3
ISI Web of Science	15	6	5	0	1
IEEE Digital Library	100	15	10	2	3
Elsevier	22	10	8	1	1
Overall	1086	114	64	13	35

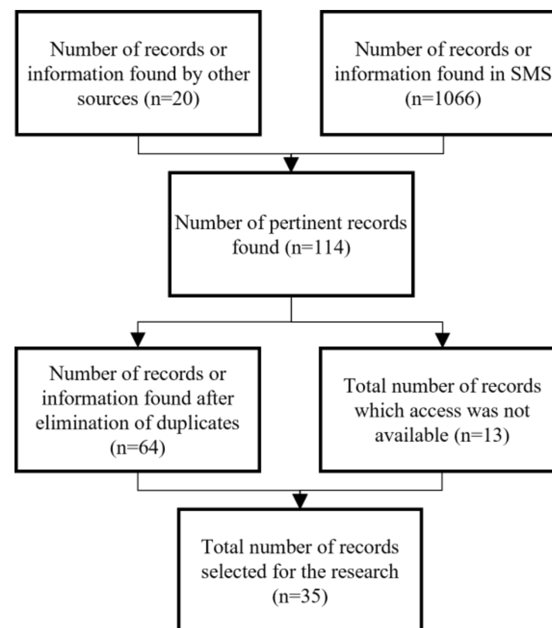

Figure 2. SMS results

3. Results and discussion

The outcomes for each of the identified research topics are shown below.

3.1. First question: What kind of strategies, methods, and/or frameworks are used for the design of software architectures?

A system's software architecture is a set of structures essential for understanding and analyzing the system. It encompasses software elements, their interrelationships, and associated properties. It comprises software elements [8] and it is important for a variety of reasons, from carrying out the quality attributes, seeing the qualities of the system, and seeing the system constraints, to being the basis for the evolution of the system. Seeing the importance of software architectures, we agree that the design of these is vital for a system to function properly and meet its objectives, in the design of architectures, decisions are made to transform the purpose, requirements, constraints, and other concerns in structures which are used to guide the project [37].

So, what to do when starting the activity of designing architecture? It may seem an infinitely complex task, but over the years design principles have been developed whose function is to guide (rather than force) the creation of high-quality designs. These concepts are oriented to the achievement of specific quality attributes (modifiability, availability, scalability, among others) and work as the building blocks from which the structures that are built that make up the architecture [37]. Ultimately, if the structure is poorly founded, the architecture does not matter much [34]. Table 9 explains the relevant design principles and patterns found in the main studies.

These practices work as the cornerstone for the design of software architecture since they provide a transfer of knowledge about architectures used throughout the history of software development [37]. Although these practices are primarily employed during the architectural design phase, it is important to note that their application extends beyond this specific phase. Architecture is a pervasive process that transverses the complete life cycle, from the risk identification phase to the delivery in each iteration of software development. In the same way that these practices are the cornerstone for the design of architectures, there are some strategies that within the software development community are taken as basic and necessary for this type of study. The strategies found will be described as follows:

1) 4+1 Model: Software architecture encompasses various aspects such as abstraction, decomposition, composition, style, and aesthetics. Describing a software architecture involves using a model that consists of various points of view or perspectives. To tackle large and complex architectures effectively, a proposed model comprises five main views: A model is used to describe a software architecture with many viewpoints or perspectives is used. A suggested model has five key viewpoints that can efficiently handle huge and complicated architectures: (i) The logical view: Represents the design's object model, particularly when object-oriented design is used. The concurrency and synchronization features of the design are captured by the process view (ii). (iii) The physical view: Highlights the distributed nature of the software and describes how it maps onto the hardware. (iv) The development view: Explains how the software is statically organized within its development environment. (v) The "+1" view: Encompasses architecture decisions, organized around the four previous views, and illustrated through selected use cases or scenarios. These scenarios play a crucial role in shaping architecture as it evolves over time [10].

2) Top Down: This method starts with the complete system at its most fundamental level before beginning a process of breakdown and gradually descending into more precise layers. At the beginning, the highest level of abstraction is present. The design gets more specific as the deconstruction goes along until the component level is reached [24]. The public interfaces of these components have a significant role in the design even when the intricate design and implementation details are not directly engaged. We can make inferences about how components will interact with one another thanks to public interfaces [24].

3) Bottom Up: In contrast to the top-down approach, this alternative method starts with the necessary components required for the solution. The design then progresses upwards, moving into higher levels of abstraction. Components behave as building components, collaborating to produce other components, eventually resulting in larger structures. This iterative process continues until all requirements are fulfilled. In contrast to the top-down approach that begins with a predefined high-level structure, the bottom-up approach doesn't have an upfront architecture design. Instead, architecture gradually emerges as more work is accomplished, adapting, and evolving with each step of the process. Consequently, this is also known as emergent design or emergent architecture [24].

4) Domain Driven Design: This style of strategic design provides development teams and business analysts with guidance on how to break down the domain of their software system into sub-areas known as “bounded contexts” [27]. Under domain-driven design, the software code's structure and language are aligned with the business domain. Within a bounded context, all business language concepts are clearly and unambiguously defined. There are concepts in every domain that can be uniquely assigned to a bounded context, on the other hand, the same concept can exist with a slightly different definition in two separate bounded contexts.

Table 9. Practices to design

Principle	Description
Single responsibility principle	SRP works as active reasoning of Conway's law: The social structure of the organization to which it belongs has a considerable influence on the appropriate structure for a software system [34]. It implies that each software module has a single (and unique) cause to change.
Open-closed principle	OCP aims for software systems to be easy to update by allowing behavior to be modified by adding new code rather than changing current code [34].
Liskov substitution principle	LSP states that to build software systems from interchangeable parts, such parts must follow a contract that implies that each part can be substituted for any other [34].
Interface segregation principle	ISP seeks that designers avoid dependency on things (modules, components, classes, objects, among others) which are not used [34].
Dependency inversion principle	According to DIP, code implementing high-level regulations should not rely on code implementing low-level details [34]. The details should depend on the policies and not the other way around.
Reuse/ReleaseEquivalence Principle	REP component cohesion principle (along with CCP and CRP) dictates that for software components to be reusable they must be traceable through a release process and have corresponding release numbers [34].
Common closure principle	CCP states that components should not have multiple reasons for change, thus providing an incentive to group classes that will most likely change for the same reason into a single component [34], thereby minimizing release related workload.
Common reuse principle	According to CRP, classes and modules that are frequently reused together belong to the same component [34]. This ensures that the dependency is more manageable and avoids unnecessary deployments due to erroneous dependencies.
Separation of concerns	It states that any system should be separated into different sections that address a concern. Architects build layered architectures to cut down on incidental coupling, creating isolation layers [20].
Inversion of control	IoC It is a design principle in which the flow of execution of a program is reversed with respect to traditional programming methods, desired responses are specified leaving the architecture to carry out the actions required to reach that response, it is not the same as dependency inversion [34].
Cloud native	The cloud-native principles encompass a set of core ideas and practices that guide the development and deployment of applications in cloud environments. These principles include microservices architecture, containerization, dynamic orchestration, and DevOps practices, among others.
On premise infrastructure	The key design is that infrastructure is located within the organization's premise which provides customizability, security control, data privacy, cost control but requires a comprehensive disaster recovery plan, regular maintenance, updates, and upgrades, and internal expertise in server administration [28].
Other patterns	**Description**
Unit of work pattern	Control is maintained over everything done during a negotiation transaction that may affect the database [35] so that changes to the database and the resolution of concurrency problems can be coordinated.

Gateway pattern	An object that encapsulates access to an external system or resource [35]. This avoids having several system resources accessing external resources on their own and facilitates the understanding of the code.
Mapper pattern	An object that establishes communication between two independent objects [35]. This avoids the creation of unnecessary dependencies between entities.
Layer supertype pattern	A type that acts as a ruler of all types in its layer [35]. In other words, a parent element that contains the set of types in common of its children, so that they inherit from it without the need to repeat code/structure.
MVC	Model-View-Controller is a method of organizing code's key functions into nicely arranged boxes. This makes considering your app, revisiting it, and sharing it with others easier and cleaner. The model component represents real-world objects, the view part is everything that interacts with the user, and the controller part serves as a bridge between the view and the model, receiving user input and choosing what to do with it.
Other Methods	**Description**
Invariant Refinement Method	It is a goal-oriented design process that generates low-level restrictions that are operationalized by system components [23]. It is based on the concept of iteratively refining system objectives. Unlike other object-oriented methodologies, IRM focuses on the system components and how they contribute to the achievement of the goals.
Attribute Driven Design	ADD is a method composed of different strategies, these strategies or steps are the following: (1). Select a system element to design, (2). Determine the Architecturally Significant Requirements (ASRs) for the element of interest, (3). Create a design solution for the selected piece, (4). Inventory remaining requirements and choose the next iteration's input (5). Repeat until all ASRs are satisfied. The result of this procedure is not an architecture that is complete in every aspect, but an architecture in which the fundamental design approaches have been picked and vetted [8].
Model-Based Design	Models are used throughout the manufacturing process (design, simulation, code creation, and verification). It enables early validation and verification, which serves as a foundation for automated software synthesis [16].
Model Driven Development	It is based on the idea that domain models should be created from which the code is generated automatically. Developers create a platform-independent model (PIM) that is combined with a platform-definition model (PDM) to generate code [22]. PIM would be the realization of the functional requirements while PDM would be the quality attributes and platform specifics.

5) Attribute-driven design (ADD): ADD is an approach for developing software architectures that take into account the software's quality attributes. It is a step-by-step architecture design method that relies on an iterative process of selecting a specific part of the system to design. Subsequently, suitable architectural styles, patterns, and tactics are chosen to fulfill important architectural requirements for that part. Each ADD iteration's outcome may be saved in a separate view packet [13]. Since ADD is a sequential, five-step method [8]. These are: (i) Choose a specific element of the system to be designed. (ii) Determine the architecturally important criteria (ASRs) for the chosen element. (iii) Generate a design solution tailored to the selected element. (iv) Assess and document remaining requirements while determining the next iteration's input. (v) Repeat steps 1 to 4 until all the ASRs have been adequately addressed. Keeping a record of the design chronology, including the sequence of decisions made, can be valuable for future reference or when modifications to a design decision are needed.

By examining the decisions made before and after a particular choice, it becomes easier to assess the potential impact and necessity of modifying subsequent design decisions.

6) Clean Architecture: This architecture is based on the concept of the “Dependency Rule”, which dictates that: “Source code dependencies must point only inward, toward higher-level policies.” as shown in 3. In the architectural principle being discussed, information within an inner circle must remain oblivious to anything in an outer circle. Specifically, code in an inner circle should not reference the names of entities declared in an outer circle, such as classes, variables, functions, or any additional specified software entity. Similarly, an inner circle should not use data formats stated in an outer circle., especially if these formats are generated by a framework located in an outer circle. The purpose of this principle is to maintain clear and strict boundaries between different circles of the system, promoting better modularity and separation of concerns [28].

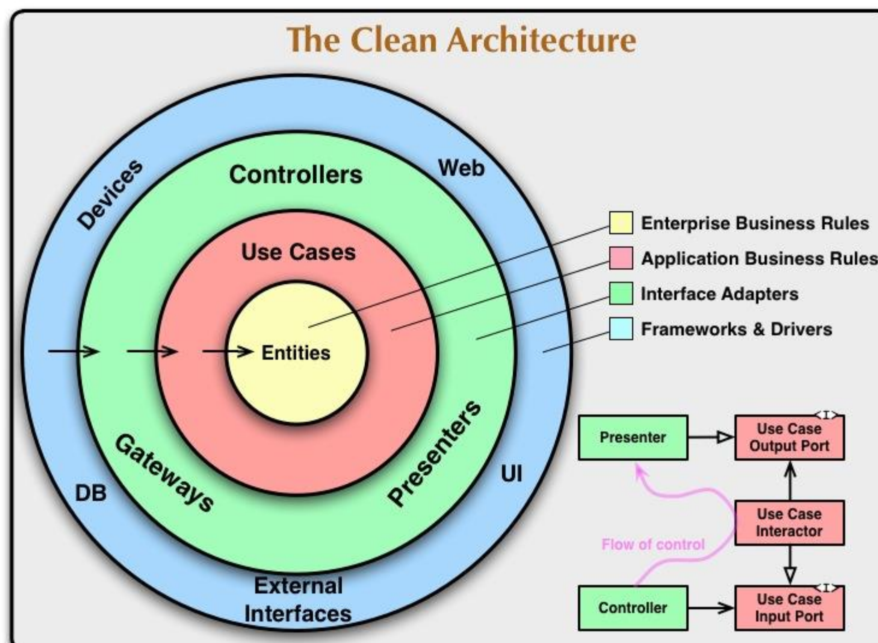

Figure 3. Clean architecture proposed by Robert Martin [28]

3.2. Second question: What types of modeling strategies and patterns exist to represent software architectures?

There are no excellent or terrible architectures, these are simply more or less suitable for the objective required by the system, in this sense each system has a unique architecture that meets (or does not) the objective of the system [22]. Considering the above, the representations for software architectures and their types depend on the objective of the system, and as this is unique for each system, but there are some representations (better-called architecture patterns) that help with the creation of these designs, they help by creating an outline that allows the user to define a structure (or schema) for any software system. Also, one of the stronger benefits (if not the strongest) is that these patterns and models are reusable, this refers to a predefined set of subsystems, roles, and responsibilities that are offered by the system. Listing all of these is a task that is beyond the scope of this work, however, through the systematic mapping studies, some representations were found that (for this approach) were considered the most relevant.

1) C4 Model: It is a graphical notation used to model the architecture of software systems, based on a structural decomposition of the system into containers and components. It leverages UML (Unified Modeling Language) and/or ERD (Entity-Relationship Diagrams) for a more detailed decomposition of the architectural building blocks. This model documents the architecture by showing multiple points of view, these are organized by hierarchical level: Context diagrams (level 1), Container diagrams (level 2), Component diagrams (level 3) and Code diagrams (level 4). Figure 4 shows this hierarchy.

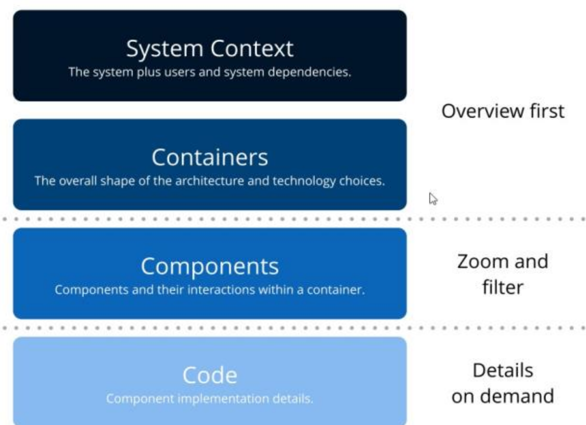

Figure 4. C4 Model proposed by Neal Ford et al. [20]

2) UML: Unified Modeling Language is a pictorial language used to make software blueprints [20]. It is used to visually represent, specify, build, and document a software system. The parts are like components that can be connected in various ways to form a complete UML picture, known as a diagram, there are many types of diagrams, the most known are: Class diagrams, Component diagrams, Use-Case diagrams, Activity diagrams, and Sequence diagram. Understanding the various diagrams is crucial for implementing knowledge in real-life systems. These diagrams fall into two main categories: Structural Diagrams and Behavioral Diagrams, each of which comprises several subcategories.

3) Layered Architecture Pattern: As its name says, it separates the architecture into different layers. The most common usage of this pattern involves four distinct layers: presentation, business, persistence, and database. However, it is not limited to these specific layers, and users have the flexibility to include additional layers, such as an application layer, service layer, data access layer, or any other layer as needed for their application.

This pattern is notable for its clear distinction of roles for each layer within the application, and each layer is marked as closed. This implies that a request must pass through the layer directly beneath it before reaching the subsequent layer. Another significant concept of this pattern is "layers of isolation," which allows modification of components within one layer without impacting the other layers. This ensures a modular and maintainable design, promoting flexibility and ease of development. Figure 3 shows the basic structure of this pattern.

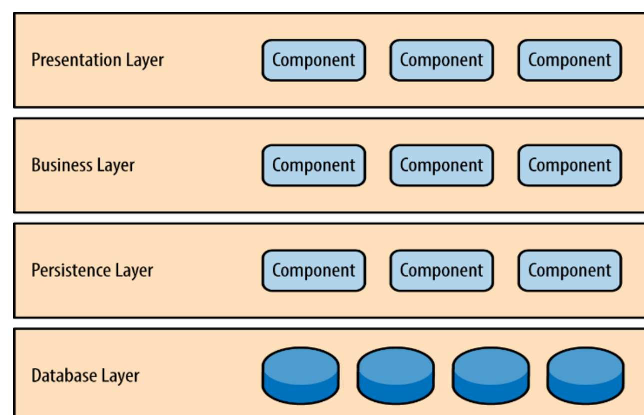

Figure 5. Layered architecture proposed by Neal Ford et al. [20]

4) Service Oriented Architecture (SOA): SOA (see Figure 6) is a strategy that focuses on discrete services instead of an indivisible unit called monolithic design. Services adhere to common interface standards and an architectural pattern, enabling seamless integration into new applications with ease. The utilization of service interfaces promotes loose coupling, allowing them to be called without requiring extensive knowledge of their underlying implementation. This, in turn, minimizes dependencies between applications, facilitating flexibility

and modularity across the system. This interface is a service contract between the service provider and the service consumer [20].

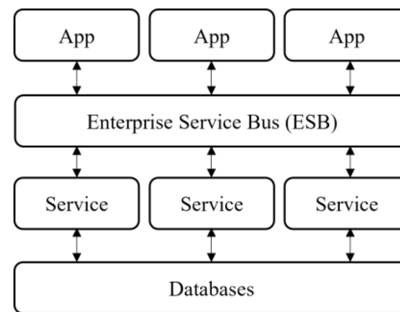

Figure 6. Service oriented architecture (SOA) [38]

5) Server-less: A recent change in the equilibrium of software development is the use of server-less architectures where the server-side logic and infrastructure management are abstracted away from developers [20]. Key characteristics of serverless architectures include: (i) Incremental change: involve redeploying code, as all the infrastructure concerns are abstracted away under the "serverless" framework. (ii) Guided change via fitness functions: Due to the criticality of coordination between services, developers can anticipate composing a higher share of overall fitness functions. These functions must operate in relation to various integration points to keep third-party APIs aligned and do not deviate from expected behavior. (iii) Appropriate coupling: There are two main meanings for serverless FaaS (Function as a service) and BaaS (Backend as a service), architects should have a deep understanding of the two to have the appropriate coupling in the system.

6) Micro-services Architecture Pattern: Micro-services are independently releasable services based on a business domain with the purpose of being technologically and functionally independent. A service encapsulates functionality and links it to additional services via networks, then the designer constructs a more sophisticated system from these building blocks [29] as shown in 7. How should a micro-service architecture be defined? The documented requirements, like with any software development endeavor, serve as the beginning point but with the twist of decomposing these into services. The architecture of an application is designed to manage and process requests. The initial step in defining this architecture involves extracting and distilling the application's requirements into the core requests it must handle. Subsequently, the second step is to determine the decomposition of these requests into distinct services. The final stage in designing the application's architecture is to determine the API (Application Programming Interfaces) for each service [33].

Microservices adopt a "share nothing" architecture, where each service operates independently removing technical coupling. This approach enables granular changes, as the main objective is to isolate domains through physically bounded contexts, emphasizing a thorough knowledge of the problem domain. Consequently, the fundamental building block of this architecture is the service itself, making it a model of evolutionary architecture. A significant advantage of this approach is that if one service requires evolution, such as changing its database schema, no other service is impacted. This is because services are not permitted to have knowledge of each other's implementation details, ensuring a high degree of isolation, and promoting a more robust and flexible system. Of course, the creators of the altering service will need to transmit the identical data via the point of integration between the services, giving the developers of the calling service the luxury of being unaware of the change [20].

The main advantage of this architectural style lies in its complete avoidance of coupling at the technical architecture layer. However, individuals who criticize coupling typically refer to" inappropriate coupling.", indeed, a software system with absolutely no coupling would lack functionality and capabilities. The concept of "share nothing" essentially translates to "avoiding entangling coupling points." While microservices promote low coupling, there are essential aspects that still require sharing and coordination, such as tools, libraries, frameworks, and more. For instance, functionalities like logging, monitoring, service discovery, etc., need to be shared and implemented consistently across microservices. Failing to include crucial monitoring capabilities for

a service could lead to disastrous consequences during deployment. In a microservices architecture, a service that cannot be effectively monitored may become invisible and difficult to manage, resembling a "black hole" within the system. Hence, proper coordination and sharing of essential components are vital for the success and operability of microservices.

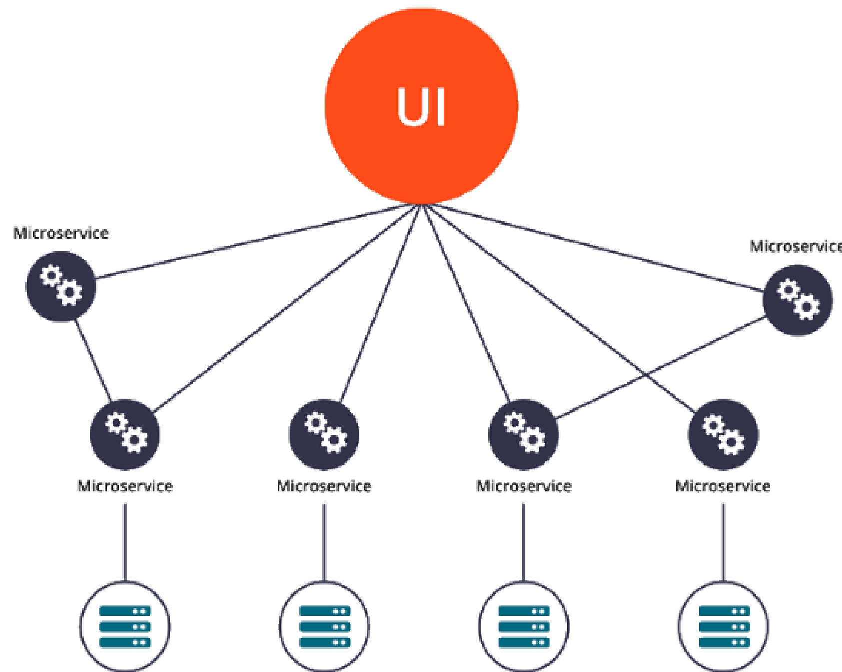

Figure 7. Microservices architecture proposed by S. Newman [29]

7) Cloud Native: The general term to define cloud computing is: “Cloud computing is the on-demand delivery of compute power, database storage, applications, and other IT resources through a cloud services platform via the internet with pay-as-you-go pricing.”; however, this merely scratches the surface of what it takes to become cloud-native. Even if it is the most mature service available, there is much more to it than simply utilizing the underlying cloud architecture [26]. Both automation and application are critical in this process. The cloud's API-driven design facilitates extensive scale automation, enabling not only the creation of individual instances or systems but also the seamless rollout of an entire corporate landscape without any human intervention. As a result, cloud-native architecture heavily relies on the approach employed to design a particular application, ensuring its compatibility and optimal utilization within the cloud environment.

8) Data Centered Architecture: Is an architectural style in which the data is designed first, followed by the design, creation, and use of applications. The architecture focuses on the movement of information within the organization, and then modifies the workflows to improve that movement. The method necessitates a full understanding of the data: where it originated, who owns it, what is the master and what is a copy, who uses it and how, how long it must be held, when it must be archived, how confidential it is, and so on [39].

9) Component Based: This approach places significant emphasis on separating concerns related to the functionality of a system. It revolves around a reuse-based methodology, involving the definition, implementation, and composition of loosely coupled, independent components into cohesive systems. Structured as a collection of components services, can be both isolated using hardware and/or software techniques or combined into a single address space [36], thus deriving a configuration from a collection of high-level services described by the developers. One of the many benefits is the creation of widely reusable software components.

10) Three Layer Framework: is a component-based approach that enables software components to autonomously arrange their interactions and accomplish a system's main objective [12]. This architecture arranges applications into three distinct logical and physical computing layers: the presentation layer,

responsible for the user interface; the application layer, where data is processed; and the data layer, dedicated to storing and managing the application's data. One of the key advantages is that each layer has its own infrastructure, allowing for parallel development by separate teams. Additionally, each layer can be updated or scaled independently without causing any disruptions to the other layers. This separation of concerns enables efficient and flexible development and maintenance of the system.

11) Rainbow Framework: It keeps track of a running software system's runtime properties using an abstract model [15]. It evaluates the model for a violation of constraint and carries out modifications to the operating system. In principle, externalized control mechanisms separate the concerns of functionality from the concerns of “exceptional behaviors”, providing several benefits, including analysis, modularity, applicability to legacy systems, and reuse. One of Rainbow's goals is to offer a low-cost method for integrating self-adaptation features into a variety of systems.

12) KAMI Framework: System designers check a model against the required requirements and utilize the model's structure to guide the implementation process. If the model's parameters do not align with the actual system behavior, it is possible that the software will not work as expected, resulting in unsatisfactory outcomes or failures. To address this challenge, run-time adaptation of non-functional properties becomes essential. This adaptation allows the system to dynamically adjust its behavior to match the real-world conditions, accounting for potential differences or environmental changes. Consequently, models for non-functional requirements should coexist and continuously interact with the system's implementation during run time to ensure accurate and effective performance. So, the Kami Framework continuously updates the reliability parameter and building performance models based on data collected during runtime [15].

13) Kieker Framework: This Framework comprises two main components: the monitoring part and the analysis part. In the monitoring phase, monitoring probes gather measurements, which are represented as monitoring records. These monitoring records are then passed to a configured monitoring log or stream by a monitoring writer. Monitoring readers ingest pertinent monitoring records from the monitoring log or stream for analysis and send them via a set of programmable analysis plugins with a pipe and filter architecture. Kieker, which focuses on application-level monitoring, contains monitoring probes for gathering timing and trace information from distributed program executions [15].

14) Bus-based Software: Bus-based software is based on SOA and refers to a software architecture or design pattern that utilizes a bus-like structure for communication and integration between different software components or services promoting low coupling between components since it is not necessary for the source of an event to be aware of where, how, or why this information will be handled. Then, SOA is a higher-level architectural concept, while bus-based software is a specific implementation approach for communication and integration [1].

3.3. Third question: What kind of strategies, methods, and/or frameworks are used for the design of cyber-physical system architectures?

The description that follows encompasses strategies, methodologies, and frameworks that are directly aligned with the research objectives of this paper. It is acknowledged that the range of available options extends beyond those discussed here, the following descriptions highlight the most pertinent approaches.

1) MAPE-K: A fundamental paradigm for this type of system, especially when self-adaptation is required (as in the case of this study) is the MAPE-K feedback loop, whose acronym translates as Monitor, Analyze, Plan and Execute over a shared Knowledge [7] (see Figure 8). It is the integration of distributed computing resources with self-management capabilities that can adapt to unexpected changes while concealing inherent complexity from operators and customers. This method draws its inspiration from the autonomic nerve system of the human body, which regulates vital bodily processes (including blood pressure and heart rate) automatically and without conscious thought.

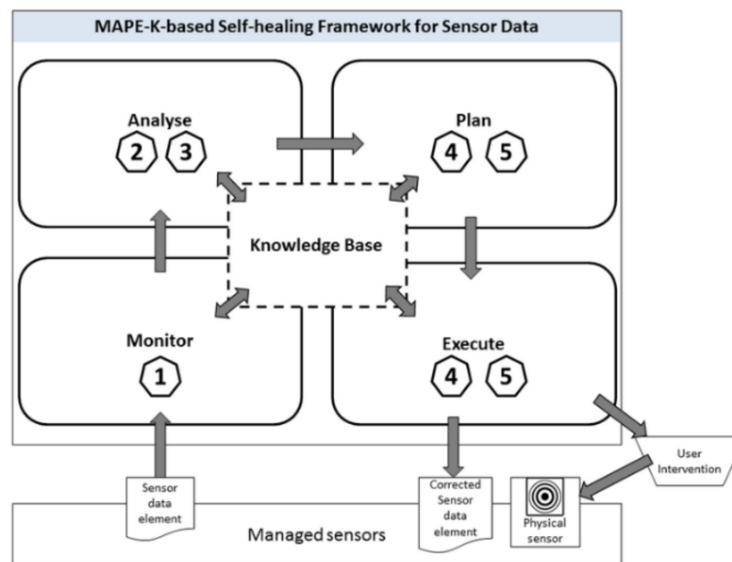

Figure 8. MAPE-K Loop proposed by Paolo Arcaini et al. [7]

2) 5C: Known as five-level it consists of the following levels: Connection, Conversion, Cyber, Cognition, and Configuration see Figure 9 [16]. At each level of the hierarchy, distinct analytical procedures are employed to extract valuable information and system knowledge from the data. Various analytical procedures are used to aggregate data from lower levels, and crucial high-level information is passed back down the hierarchy. To create a CPS in production system-based manufacturing, a 5-level structure known as the 5C architecture offers a clearly stated rule. This sequential workflow approach ensures a more detailed and transparent construction of a CPS. In this context, advanced interconnection is vital for the gathering of real-time data, facilitating the connection between the physical world and specific processes while incorporating feedback from cyberspace. This seamless integration allows for enhanced control and optimization of CPS operations.

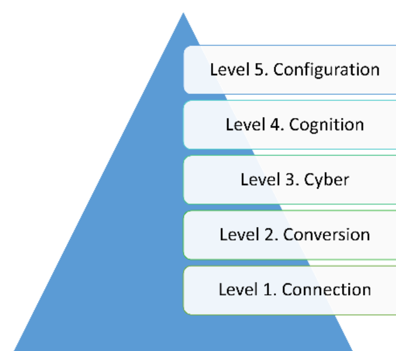

Figure 9. 5C architecture proposed by Ioan Dumitrache et al. [16]

3) Safe State Space: When the controller is unable to maintain control of the controlled plant inside a specified a subset of its Safe State Space (SSC), a cyber-side failure occurs in a CPS [36]. This technique directs the user or application engineer in the specification of a set of restrictions (Safety Space Constraints) that to be regarded operationally safe, the plant must meet certain criteria. A system is in a safe state space if the plant satisfies the SSCs at the present time, this helps the system know if there is any need to apply some controller input and/or helps to mitigate the consequences of a mistake input hence maintaining the correct performance of the plant and thus achieving adaptive fault tolerance.

4) Self Aware Monitoring: Current efforts to enhance automated systems' efficiency, collaboration, and resilience in the industrial sector underscore the significance of self-awareness within these systems. Self-awareness allows a system to keep track of itself and its surroundings to better analyze its position and produce more suitable judgments [35]. This methodology is applied to the architecture as a logical layer to keep track of the system's health deterioration, but some studies have found that it was possible to implement as an agent in

a multi-agent architecture [11] or in a more typical hierarchical architecture thus giving the system the ability to keep track of its own internal and external conduct to make sound decisions.

5) Cognitive Systems Architecture: A cognitive architecture's purpose is to construct a framework for developing human-like intelligence in systems; they give a framework that allows a system to evolve over time by incorporating perception, reasoning, action, and learning mechanisms [35]. Cognitive systems frequently exhibit social behavior to overcome problems caused by poor environmental perception and the inability to accomplish global tasks separately, involving communication, collaboration, and negotiation. This method is used in cognitive radio networks to increase spectrum sensing performance by collaborating selectively with numerous remote sensors and optimize radio frequency spectrum utilization.

6) Dynamic Clustering Architecture: Effective communication among system components is pivotal for achieving efficient performance in distributed systems. Clustering was developed in response to the requirement to adjust to growing industrial control systems' high complexity and dynamic nature [35]. Every cluster represents a dynamically formed community of intelligent system components collaborating to gather sufficient information for problem-solving. Each cluster member possesses a set of algorithms enabling problem recognition and solution discovery.

7) Invariant Refinement Method for Self-Adaptation: The idea is to evolve on the idea of IRM by identifying and mapping applicable configurations to make it adaptable to given situations this is done by developing design alternatives for achieving system requirements so they can be employed for architecture adaption at run-time. Three recurring steps are used in self-adaptation [7]: (i) Determine the present circumstance. (ii) Choose one of the available configurations. (iii) Reconfigure the architecture to match the chosen configuration.

8) Model-Based Software Development Method for Automotive Cyber-Physical Systems: The primary workflow of MoBDAC's development comprises four key steps. First, It entails deriving software specifications from system specifications. Second, modeling tools are utilized to construct models in the problem domains (MPD), which are then subjected to simulation for verification purposes. Third, the MPD is transformed into models in the implementation domains (MID). Finally, the MID is employed to generate the actual code for the system. It should be noted that system specifications are used to extract non-functional needs as well as interactions with the physical environment. Analysis tools employ non-functional requirements to determine whether the software's non-functional requirements are satisfied, and the information on the interaction with the physical environment is used by MID to generate correct code [22]. Figure 10 illustrates the MobDAC architecture.

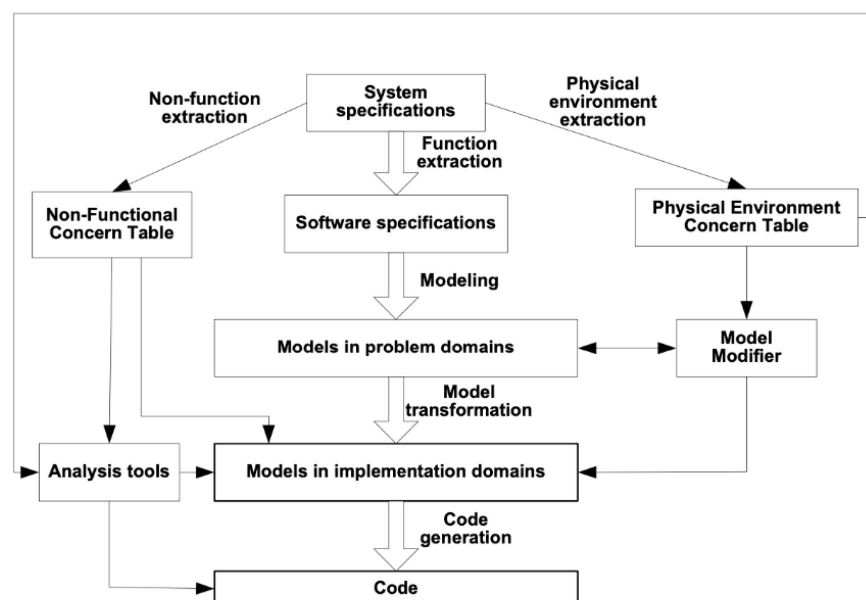

Figure 10. MoBDAC Architecture proposed by Zhigang Gao et al. [22]

3.4. Fourth question: What types of modeling strategies and patterns exist to represent cyber-physical systems?

Representing cyber-physical systems requires modeling strategies and patterns that can capture the complex interactions between physical process and computational components [14]. Here are some modelling strategies and patterns used to represent CPS:

1) Traditional Hierarchical Architecture: Most traditional manufacturing methods fall into this category. These systems are based on centralized and staggered control techniques, offering efficient outputs due to their optimization capabilities. However, their rigid multilevel structure hinders agile responses to potential variations. Hierarchical architectures, such as pyramid-like Computer Integrated Manufacturing (CIM), demonstrate limited autonomy, making the system susceptible to disturbances and resulting in weak responses when facing disruptions. This rigidity raises the development expenses and results in a system that is difficult to maintain [14] even though it produces a system with maintainability issues it also refers to a systematic way of thinking, working, and communicating.

2) Multi Agent System (MAS): The core of this design are the autonomous components, known as agents, that are taught to collaborate through negotiation protocol structures [11]. MAS approach eliminates every kind of hierarchy, granting all power to the essential modules. By removing the system's hierarchical links, the components work together equally, instead of designating subordination and supervisory connections, the consequence is a flat design [14] (See Figure 11).

3) Holonic Manufacturing System (HMS): HMS is made up of holons that are autonomous, intelligent, flexible, dispersed, and cooperative. With this design, the production process is driven by the product cases themselves, leading to complete decentralization of coordination through holons. Manufacturing based on holarchies (levels of holons) anticipates future actions, in contrast to previous decentralized setups and utilizes proactive efforts to avoid coming problems [14]. Therefore, one of the most promising aspects of HMS is their ability to represent a change from wholly hierarchical to hetero-hierarchical structures.

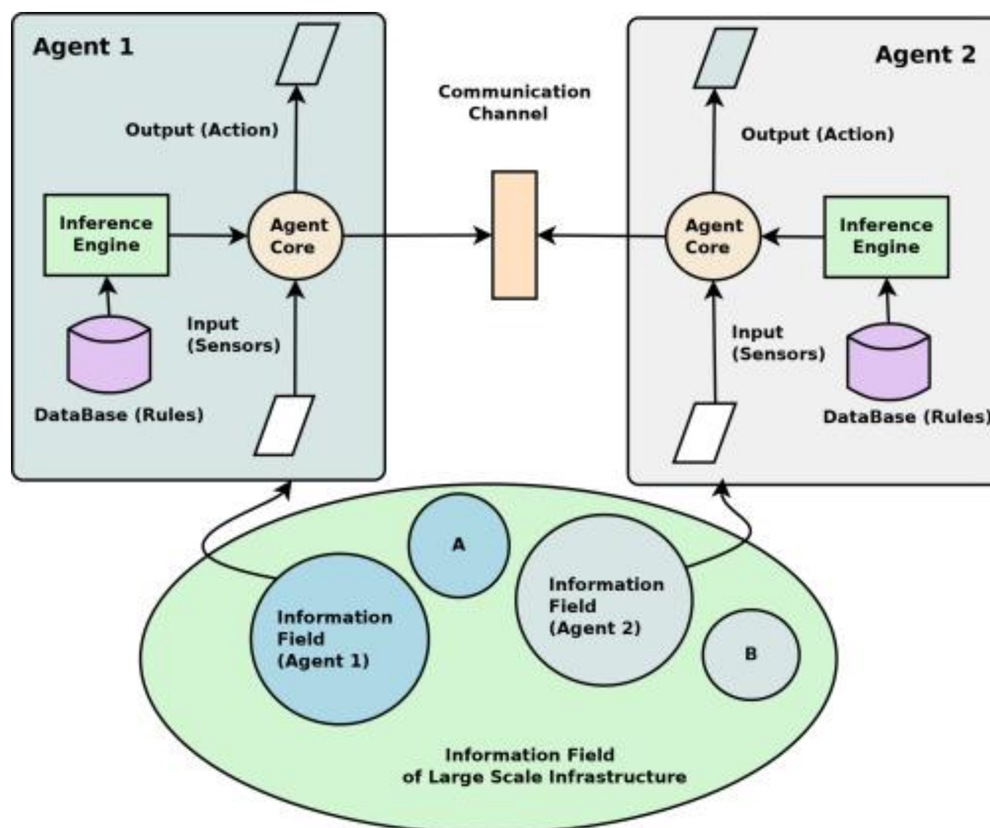

Figure 11. MAS Architecture proposed by Sagit Valeev et al. [40]

4) Traditional Embedded System (TES): Regarding the physical world, TES implements an asymmetric control relationship [31], only the computational processes launch the monitor-and-control interaction with the application but not vice versa. TES emphasizes the importance of an integrated software framework in which basic adaption functionality is linked with high-level application functionalities which preclude quick incremental software changes and configurations.

5) Adaptive Fault Tolerance (AdaFT): AdaFT framework consists of two major parts: the first approach concentrates on generating the sub-spaces and employing a machine learning technique for sub-space classification. On the other hand, the second approach utilizes the subspace classifier's outputs and conducts system simulation alongside reliability analysis. AdaFT applies the adaptive fault-tolerance technique after taking the physical side data of the controlled plant as input, ensuring the same level of safety as the conventional way while maintaining the most effective use of computing resources, thereby improving the computing platform's long-term reliability [25].

6) SAMBA: "SAMBA's logical unit of an entity is an Autonomous Cooperating Object (ACO)" [35] (see Figure 10). Each ACO autonomously learns the nuances of its surroundings and the options available to it. Moreover, it displays social conduct as it interacts with other ACOs within the same surroundings. When deciding on actions to execute, it considers its own goals, the environmental situation, and demands from other ACOs. The collective conduct of the ACOs derives from their interactions, giving rise to the overall global behavior of the system.

7) Reconfiguration Framework for distributed embedded systems for Software and Hardware (ReFrESH): ReFrESH is a four-layer framework designed to facilitate self-adaptation in both hardware and software components (see Figure 13). Its layers are (i) the Resource Layer, which provides actual hardware resources as well as capability indicators to aid robot actions, (ii) the Interface Layer, which offers component interfaces for driving and requesting hardware resources, (iii) the Component Layer, which consists of task-execution components, an evaluator, and an estimator to evaluate the performance of running and incoming components, and lastly, (iv) The management unit Task Layer produces configuration candidates and select a suitable setup to carry out one or more jobs [15].

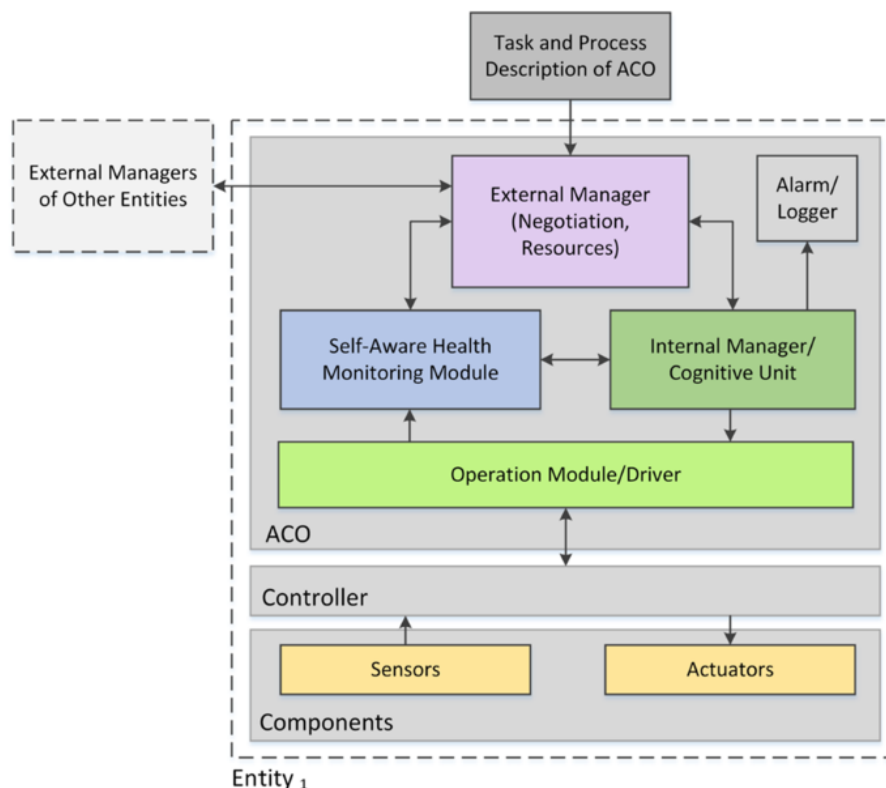

Figure 12. SAMBA Framework proposed by Lydia Siafara [35]

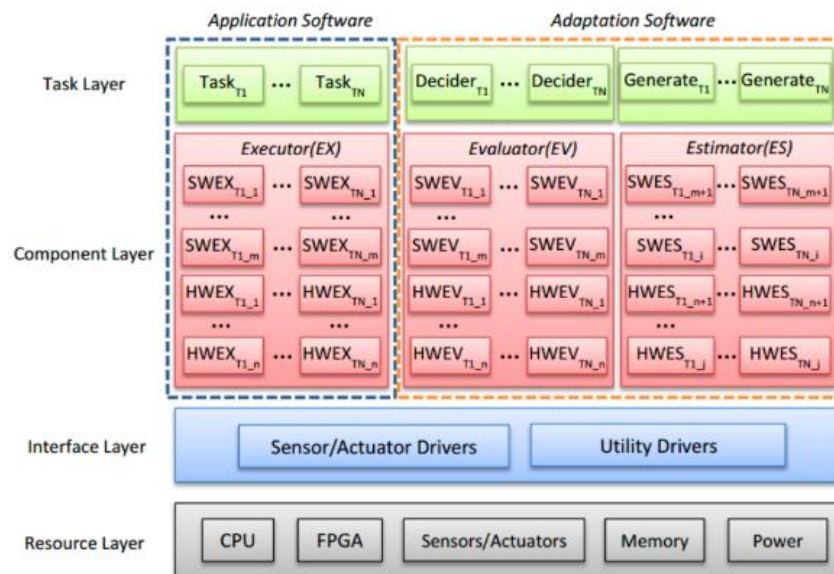

Figure 13. 4-layer framework proposed by Yanzhe Cui et al. [15]

3.5. Fifth question: Is there a relation between the definition of architecture and sustainability?

With the increasing software dependency of cyber-physical systems, a noticeable trend has emerged wherein control tasks are shifted from isolated controllers to an integrated computation platform. Historically, enormous always-on redundancy was used to assure consistent controller efficiency [36]. In numerous cases, the controlled plant operates deep within its permitted state space, which means that minor controller failures do not result in malfunction of the controlling plant. This encourages a flexible approach to maintaining sustainability.

In systems engineering, sustainability refers to adopting and implementing iterative and incremental methodologies that foster the long-term development of technologies at a low cost and with reduced effort, seeing this as an important aspect of which the development of both software and cyber-physical systems is migrating to sustainable development which at the same time is related to architecture. Sustainability entails the creation of products that are both technically sound and economically beneficial. Even though sustainability has typically been linked to the environmental aspect, its significance is growing in the broader context of engineering, including software engineering. For software systems integrated into Cyber-Physical Systems (CPS), sustainability is closely tied to nonfunctional attributes, especially maintainability, and nonfunctional attributes with sustainability dimensions.

"There are five dimensions of sustainability: (i) environmental, (ii) social, (iii) economic, (iv) technical, and (v) individual" [32]. Restrepo et. al. [32] define that a sustainable-system architecture is attained after the system is ready for maintenance and evolution., an attribute that -indirectly- encompasses the ideas of lifespan and cost-effectiveness. Due to the constant evolution of these systems, the attribute of self-adaptation or adaptability comes in as a necessity. Self-adaptation is considered an essential characteristic of systems that function in dynamic environments and manage operating situations that are continually changing [7].

To attain high quality and versatility in manufacturing processes, high-efficiency production demands a high level of adaptability, and reactivity [35]. In efforts to shorten the lead time, past approaches have emphasized automated manufacturing environments that are strict and deterministic that aim to reduce operational disruptions. Nevertheless, with the rising structural complexity of manufacturing systems, driven by the inclusion of more CPS and heterogeneous components distributed, the determinism of manufacturing processes is diminishing, and requires an adaptive approach to gain (or maintain) sustainability without decreasing its flexibility.

Considering that CPS is implemented in dynamic environments, with several variables where uncertainty dominates, it is clear the need to have in mind an architecture that gives priority to sustainable development, not

only in its design but throughout the entire life cycle of the system. In the event of a new and unexpected occurrences, the system will adapt accordingly not only the items' existing statuses, but also their future plans, while efficiently communicating these revised plans with all relevant users. Users should be given updated plans to help them manage their tasks and provide feedback (engineers, workers, drivers, among others.) [23]. Control-theoretic feedback loops are frequently used to achieve adaptability [31] to process system outputs in relation to actuator signals produced by the controller. These designs have remarkable resistance to change and may be continuously altered to their surroundings [14].

A relationship between the definition of architecture and sustainability does exist since self-adaptation is accomplished through the implementation of adaptation techniques, such as the MAPE-K feedback loop. From a technical standpoint, sustainability is related to the creation of reusable software components, with a focus on achieving the maintainability attribute. Economic sustainability is related with the development of algorithms that lower expenses in analysis, data collection, and energy use [32]. Technical sustainability is also linked to the utilization of layered, microservices, and cloud-based architectures, which enables scalability of the system.

Discussion, challenges, and gaps

Designing of software architectures: Architecture design practices bring benefits for technology heterogeneity but the consulted literature reports challenges at the level of system complexity, changing requirements, security, scalability, and maintainability. Designing software architectures can be complex and challenging since involves making a wide range of decisions than can have an impact in the quality of the software system also because there are numerous options available [12], [41], designers must carefully evaluate and select the most appropriate technologies, patterns, and approaches. To address these challenges a framework could be designed to help select the best framework for a certain domain or specific feature requirement, also making decisions at the design level, adopting best practices, continuous improvements of design, and the use of a robust process can help.

Representations of software architectures: Software representations face several challenges such as consistency between representations that lead to misunderstandings, communication problems, inadequate representation or documentation, and implementation errors, also software representations can be time consuming and resource intensive, managing different versions can be complex, and representation of non-functional aspects are overlooked or not visible to stakeholders. To address these challenges management solutions, flexible representations, automatic tools, and best practices can help to reduce the problems.

Designing of CPS architectures: Designing CPS presents challenges both computational and physical elements, as these systems require high degree of scalability and monolithic architectures may not always be well-suited for CPS because the lack of modularity, scalability issues, inefficient use of resources, and other. To address these challenges to propose a framework that consider hybrid approaches may provide a more suitable solution, achieving better component heterogeneity, high interoperability, low power consumption, also employ robust designs approaches through co-engineering, which involves collaboration between different disciplines can help with these challenges. These disciplines may include software engineers and domain specialists.

Representations of CPS: Represent CPS is challenging due to real-time constraints and behaviors, also because involves a mix of hardware, software, sensors, etc., there are no standards that allow the homogeneous representation of components, interdisciplinary, and in many cases lack of modeling tools for physical and computational aspects, also representations have limited extensibility to other domains since often require domain-specific knowledge, to address these challenges modular an adaptable representation could be proposed.

Architecture and sustainability: The gap found in the literature was a lack of guidelines for constructing sustainable and evolvable CPS has significant implications for the development and long-term viability of these complex systems since (i) developers face a greater challenge when designing and implementing CPS than can result in ad-hoc solutions leading to longer development cycles, and higher costs. (ii) Sustainability is a critical aspect especially in the resource utilization such as energy consumption and responsible use of resources, also

there is an environmental impact and designing without sustainability can lead to consume more resources and contributing to environmental issues. (iii) Without guidelines maintaining and updating CPS over time becomes more difficult and error-prone can lead to systems that quickly become obsolete or require resources to adapt. To address these challenges a conceptual framework focused on sustainability for the design of CPS architectures can be proposed, also consider the entire life cycle of the systems [32] to show a holistic approach that considers the social, environmental, technical, and ethical aspects of sustainability.

3.6. Towards a conceptual framework for designing sustainable CPS architectures

Methodological proposals must address the product life cycle, requirements, technologies, domains, adaptability, and execution contexts, among other challenges, having correspondence between what you want to build and how you are going to build the solution, there are traditional and conservative architectural proposals and others that emerge to respond to current challenges where design frameworks provide flexibility, agility for changes, adaptability, scalability, high evolution, technological independence, reduce coupling, among others. traditionally, as evidenced in the SMS, the architectural design of software and CPS has been performed using monolithic structures where the functional aspects are coupled and subject to the same solution, generating long-term problems at the level of evolution, changes in requirements, technology, and others. Having detected the need to create agile solutions that respond to flexible, modular designs and a reduced development effort from the field of software architectural design, microservices architectures have gained popularity since they propose a structure that better meets the characteristics of the challenges and in real environments, have better behavior and provide advantages such as modularity, versatility, and small code base [32]. Knowing the methodological and architectural design implications and the different gaps to approach CPS, we identified the opportunity to embrace the characteristics of this proposal in the context of the architectural design of CPS, so with the following proposal we take the first step and represent what would be a framework that addresses the best practices that the literature is offering us.

The proposal proposes a conceptual framework that makes visible the domain and its decomposition, where it is important to isolate the domain via physical contexts, this approach emphasizes to fill this good practice of software development to the context of the CPS since it allows to understand the domain of the problem. The basic structure and the highest level of this proposal are shown in Figure 14, each section will be explained below.

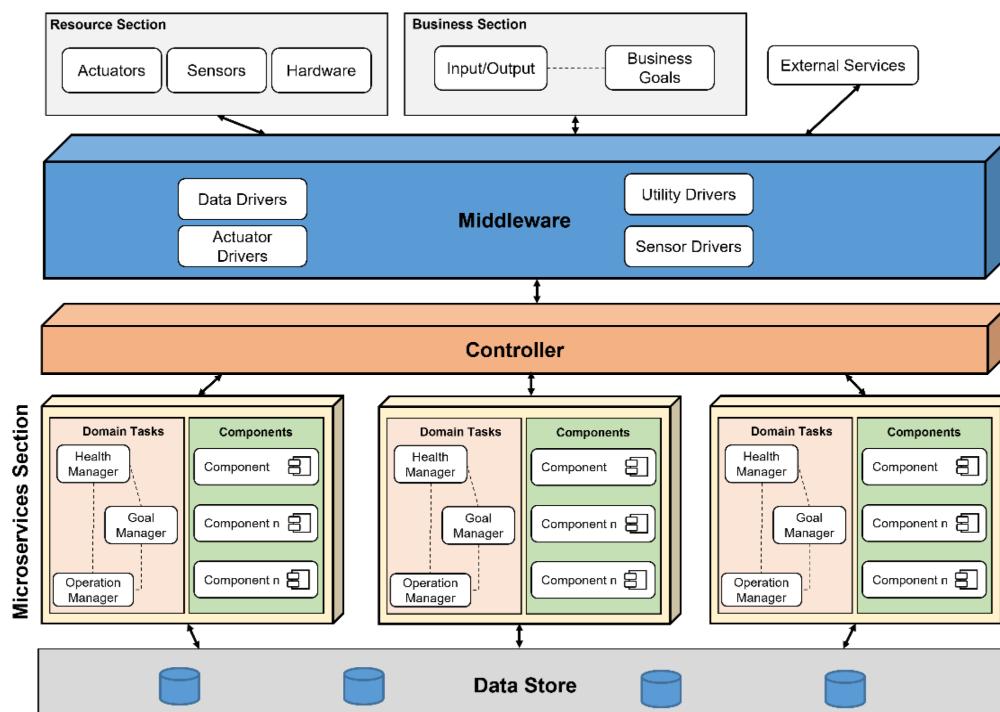

Figure 14. Proposed framework

• A *resource section* that defines the physical specifications of each system that implements this framework, it symbolizes the physical component of the cyber-physical system, here we will find (among several specific components of each system, necessary for each task) two components that are the most important for us, these are the actuators and sensors.

• A *business section* where the user has access to the system to enter the necessary values that would become (with interaction with its environment) the business goals, these are the ones that function as conductors for the entire framework.

• *External services* allow microservices to interact with other services, systems, and data sources outside of their own context.

• *Middleware* section contains the drivers and means necessary to provide and manage the transfer of data in a reliable way to both the upper section (client side) and the lower section (microservices and data store section). This section works as a middleware for the communication of the system, in this way, we achieve that the components are loosely coupled.

• *Controller* section is responsible for managing the incoming request and routing them to the appropriate microservice for processing providing a unified point for all incoming requests and performing basic input validation, authentication, and authorization before forwarding the request [33], [41].

• *Microservices* section is composed of Domain tasks and components. (i) Domain tasks oversees generating feasible instructions (tasks) for the component section while maintaining the best configuration. Three major parts make up this section, the first is the health manager, which receives the states of the components, and the information of the actuators, among others, and analyzes these states together with the restrictions of the system to act always in favor of the health of the system. The objective manager seeks to act in favor of the business goals, and its objective is to create tasks that maximize the utility of the system to meet the objectives. The third and last part is the operations manager, its main function is to be the brain of this section, it must take the inputs given by its twin parts (health manager and objectives manager) and in this way join forces to create the operations (tasks) that give the best result for the business objective without compromising the health of the system. (ii) Components is where all the software components are located, following the Separation of Concerns pattern (explained above) divided by the concern to avoid incidental coupling, creating layers of isolation. Within this section there are no hierarchical relationships, following the multi-Agent paradigm, thus allowing each component to function as equals.

• *Data sources* section enables microservices to access and manipulate data efficiently and reliably. This layer can be implemented in different ways such as sue a database for all microservices or using a separate database for each microservices providing flexibility and scalability but requiring more resources [42].

Proposed framework can be applied across various application domains where CPS technology is utilized such as smart cities, energy management, manufacturing, agriculture, healthcare, transportation, environmental monitoring, water management, disaster management, supply chain management, renewable energy, building automation, wearable technology, education, defense, and security. Application of the framework can vary within these domains, but the overarching goal is to design systems that positively impact sustainability. As an example the conceptual framework can be applied in the in the context of a Smart Manufacturing System to enhance the efficiency and flexibility of a manufacturing facility the microservices section can be implemented in edge devices for local data processing handling tasks like anomaly detection, and real-time control, or also could be implemented in cloud services to provide remote monitoring that will used for predictive maintenance, quality control, and process optimization, and in the middleware section high-speed and low latency communication protocols will used to facilitate data exchange between microservices. This results in enhanced adaptability, evolvability, and scalability, allowing for the easy integration of new processes.

3.7. Threats to validity and limitations of the research

The SMS focused on gathering information on Software and CPS architectures and representations. In this context, and given the nature of the study, it is important to relate the threats to the validity and limitations of this SMS.

Construction Validity: Several steps were put in place to mitigate construction dangers: (i) A well-established approach proposed in the literature led the search strategy and process. (ii) In constructing the search strings, a comprehensive range of terms related to CPS and software architecture were carefully considered. (iii) Table 6 shows how the study topics were answered using a categorization approach.

Internal Validity: We looked through six online digital databases. These libraries house a large quantity of high-quality field publications. The absence of other libraries, on the other hand, may introduce a bias in locating primary research. In addition, to limit the potential of missing significant articles, we applied the snowballing technique [43] as a supplemental search strategy. In addition, the search strategy was carefully established and reviewed. Ultimately, a clear and detailed account of the research review methods is presented to allow readers to acquire an informed opinion of the review's scientific rigor and the robustness of its conclusions.

External Validity: All the papers studied were chosen for their relevance to the CPS and software architectures. The omission of these papers might impact our findings' generalizability. The consistency of our study methodology, which is a systematic procedure that allows for repetition and mitigates this concern [44].

Conclusion Validity: To ensure conclusion validity and minimize bias in the extraction of data, cross-checking was employed. This approach helps mitigate potential discrepancies in data interpretation and reduces the influence of subjective judgment.

Some of the limitations encountered are: (i) given the vast and ever-evolving nature of software and CPS architectures, it was challenging to encompass the entire breadth of relevant research. The study may have missed emerging trends or underrepresented certain architectural aspects due to scope constraints. (ii) There is always a possibility that some relevant papers were missed leading to potential biases in the included literature, and (iii) the categorization of architectural aspects and the selection of relevant studies involved a level of subjectivity. While efforts were made to ensure rigor and objectivity, the absence of expert consensus or certain categorizations may introduce bias. However, to minimize these threats and avoid data extraction biases, as mentioned the entire process was executed by cross-checking between the authors.

4. Conclusions

This SMS discussed many methods for creating both software and cyber-physical architectures. The study considered the sustainability requirements for this kind of system. 35 articles were selected for this SMS.

For the design of software architectures, it was found that there are several practices from various sources, as well as several types of representations for this kind of system. However, this was not the case for cyber-physical systems, where fewer representations and design strategies were found. Since CPS is a relatively new technology since is something that is still being contributed. Also, it is missing a framework that allows for designing sustainable CPS architectures.

Finally, this SMS contributed to the creation of a framework for designing sustainable cyber-physical systems architectures which are based on the concept of microservices architecture allowing to construct of a framework of highly decentralized decreasing coupling which also promotes the evolvability of the system at a granular level, with technological independence. Having sustainability as the main non-functional requirement.

It is evident the heterogeneity of the methodological proposals at the level of software architecture design and CPS. Therefore, it is necessary to make a proposal that embraces the best practices of both proposals where elements such as agility for changes and sustainability are directional axes. As a future work, the proposed framework will be refined and is considered important to carry out studies on the application of the proposed

framework and thus compare the effectiveness of applying this proposal on other architectures and considering the aforementioned application domains.

Declaration of competing interest

The authors state that they have no known competing financial or non-financial interests in any of the topics mentioned in this research.

Funding information

Universidad EAFIT in Colombia provided funding for this study.

References

[1] P. Derler, E. A. Lee, and A. Sangiovanni Vincentelli, "Modeling Cyber–Physical Systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 13–28, 2012, doi: 10.1109/JPROC.2011.2160929.

[2] I. Gerostathopoulos *et al.*, "Self-adaptation in software-intensive cyber–physical systems: From system goals to architecture configurations," *Journal of Systems and Software*, vol. 122, pp. 378–397, 2016, doi: https://doi.org/10.1016/j.jss.2016.02.028.

[3] R. West and G. Parmer, "A software architecture for next-generation cyber-physical systems," Sep. 2006.

[4] A. Larab, E. Conchon, R. Bastide, and N. Singer, "A sustainable software architecture for home care monitoring applications," in *2012 6th IEEE International Conference on Digital Ecosystems and Technologies (DEST)*, 2012, pp. 1–6. doi: 10.1109/DEST.2012.6227928.

[5] S. Keele and others, "Guidelines for performing systematic literature reviews in software engineering." Technical report, ver. 2.3 ebse technical report. ebse, 2007.

[6] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Inf Softw Technol*, vol. 64, pp. 1–18, 2015, doi: https://doi.org/10.1016/j.infsof.2015.03.007.

[7] P. Arcaini, R. Mirandola, E. Riccobene, and P. Scandurra, "A Pattern-Oriented Design Framework for Self-Adaptive Software Systems," in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2019, pp. 166–169. doi: 10.1109/ICSA-C.2019.00037.

[8] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012.

[9] K. Beck, *Implementation Patterns*. in Addison-Wesley Signature Series. Upper Saddle River, NJ: Addison-Wesley, 2007. [Online]. Available: http://my.safaribooksonline.com/9780321413093

[10] P. Kruchten, "Architectural Blueprints – The '4+1' View Model of Software Architecture," *IEEE Softw*, vol. 12, no. 6, Sep. 1995.

[11] D. Carnì, D. Grimaldi, L. Nigro, P. F. Sciammarella, and F. Cicirelli, "Agent-based software architecture for distributed measurement systems and cyber-physical systems design," in *2017 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*, 2017, pp. 1–6. doi: 10.1109/I2MTC.2017.7969977.

[12] H. Cervantes and R. Kazman, *Designing Software Architectures: A Practical Approach*, 1st ed. Addison-Wesley Professional, 2016.

[13] D. Garlan *et al.*, *Documenting Software Architectures: Views and Beyond*, 2nd ed. Addison-Wesley Professional, 2010.

[14] S. L. A. Cruz and B. Vogel-Heuser, "Comparison of agent-oriented software methodologies to apply in cyber physical production systems," in *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*, 2017, pp. 65–71. doi: 10.1109/INDIN.2017.8104748.

[15] Y. Cui, R. M. Voyles, and M. H. Mahoor, "ReFrESH: A self-adaptive architecture for autonomous embedded systems," in *2013 IEEE International Conference on Automation Science and Engineering (CASE)*, 2013, pp. 850–855. doi: 10.1109/CoASE.2013.6654042.

[16] I. Dumitrache, S. I. Caramihai, I. S. Sacala, and M. A. Moisescu, "A Cyber Physical Systems Approach for Agricultural Enterprise and Sustainable Agriculture," in *2017 21st International Conference on Control Systems and Computer Science (CSCS)*, 2017, pp. 477–484. doi: 10.1109/CSCS.2017.74.

[17] M. Engelsberger and T. Greiner, "Software architecture for cyber-physical control systems with flexible application of the software-as-a-service and on-premises model," in *2015 IEEE International Conference on Industrial Technology (ICIT)*, 2015, pp. 1544–1549. doi: 10.1109/ICIT.2015.7125316.

[18] M. Erder and P. Pureur, *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2015.

[19] M. C. Feathers, *Working Effectively with Legacy Code*. in Martin, Robert C. Prentice Hall Professional Technical Reference, 2004. [Online]. Available: https://books.google.com.co/books?id=CQlRAAAAMAAJ

[20] N. Ford, R. Parsons, and P. Kua, *Building Evolutionary Architectures: Support Constant Change*. Beijing: O'Reilly, 2017. [Online]. Available: https://www.safaribooksonline.com/library/view/building-evolutionary-architectures/9781491986356/

[21] M. Fowler, *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*. Addison-Wesley, 2012.

[22] Z. Gao, H. Xia, and G. Dai, "A Model-Based Software Development Method for Automotive Cyber-Physical Systems," *Comput. Sci. Inf. Syst.*, vol. 8, pp. 1277–1301, Sep. 2011, doi: 10.2298/CSIS110303059G.

[23] S. S. and N. D. and S. P. O. Gorodetsky V. I. and Kozhevnikov, "The Framework for Designing Autonomous Cyber-Physical Multi-agent Systems for Adaptive Resource Management," in *Industrial Applications of Holonic and Multi-Agent Systems*, P. and R. G. and Z. A. and A.-K. G. and T. A. M. and K. I. Mařík Vladimír and Kadera, Ed., Cham: Springer International Publishing, 2019, pp. 52–64.

[24] J. Ingeno, *Software Architect's Handbook: Become a Successful Software Architect by Implementing Effective Architecture Concepts*. Packt Publishing, 2018.

[25] M. Kit, I. Gerostathopoulos, T. Bures, P. Hnetynka, and F. Plasil, "An Architecture Framework for Experimentations with Self-Adaptive Cyber-physical Systems," in *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2015, pp. 93–96. doi: 10.1109/SEAMS.2015.28.

[26] T. Laszewski, K. Arora, E. Farr, and P. Zonooz, *Cloud Native Architectures: Design high-availability and cost-effective applications for the cloud*. Packt Publishing, 2018. [Online]. Available: https://books.google.com.co/books?id=QshsDwAAQBAJ

[27] C. Lilienthal and Dpunkt.Verlag, *Sustainable Software Architecture: Analyze and Reduce Technical Debt*. in WPS, workplace solutions. dpunkt.verlag, 2019. [Online]. Available: https://books.google.com.co/books?id=4euMwwEACAAJ

[28] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. in Martin, Robert C. Prentice Hall, 2018. [Online]. Available: https://books.google.com.co/books?id=8ngAkAEACAAJ

[29] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015. [Online]. Available: https://books.google.com.co/books?id=jjl4BgAAQBAJ

[30] L. T. X. Phan and I. Lee, "Towards a Compositional Multi-modal Framework for Adaptive Cyber-physical Systems," in *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2011, pp. 67–73. doi: 10.1109/RTCSA.2011.82.

[31] K. Ravindran and R. Sethu, "Model-Based Design of Cyber-Physical Software Systems for Smart Worlds: A Software Engineering Perspective," in *Proceedings of the 1st International Workshop on Modern Software Engineering Methods for Industrial Automation*, in MoSEMInA 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 62–71. doi: 10.1145/2593783.2593785.

[32] L. Restrepo, J. Aguilar, M. Toro, and E. Suescún, "A sustainable-development approach for self-adaptive cyber–physical system's life cycle: A systematic mapping study," *Journal of Systems and Software*, vol. 180, p. 111010, 2021, doi: https://doi.org/10.1016/j.jss.2021.111010.

[33] C. Richardson, *Microservices Patterns: With examples in Java*. Manning, 2018. [Online]. Available: https://books.google.com.co/books?id=UeK1swEACAAJ

[34] L. Sha and J. Meseguer, "Design of complex cyber physical systems with formalized architectural patterns," in *Software-Intensive Systems and New Computing Paradigms*, Springer, 2008, pp. 92–100.

[35] L. C. Siafara, H. Kholerdi, A. Bratukhin, N. Taherinejad, and A. Jantsch, "SAMBA–an architecture for adaptive cognitive control of distributed Cyber-Physical Production Systems based on its self-awareness," *e & i Elektrotechnik und Informationstechnik*, vol. 135, no. 3, pp. 270–277, 2018.

[36] Y. Xu, I. Koren, and C. M. Krishna, "AdaFT: A Framework for Adaptive Fault Tolerance for Cyber-Physical Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 3, Sep. 2017, doi: 10.1145/2980763.

[37] H. and W. Z. and Z. Q. Zheng Bowen and Liang, "Model-Based Software Synthesis for Safety-Critical Cyber-Physical Systems," in *Safe, Autonomous and Intelligent Vehicles*, X. and M. R. M. and R. S. and T. C. J. Yu Huafeng and Li, Ed., Cham: Springer International Publishing, 2019, pp. 163–186. doi: 10.1007/978-3-319-97301-2_9.

[38] "Service-Oriented Architecture," in *Services Computing*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 89–113. doi: 10.1007/978-3-540-38284-3_5.

[39] P. López Martínez, R. Dintén, J. M. Drake, and M. Zorrilla, "A big data-centric architecture metamodel for Industry 4.0," *Future Generation Computer Systems*, vol. 125, pp. 263–284, 2021, doi: https://doi.org/10.1016/j.future.2021.06.020.

[40] S. Valeev and N. Kondratyeva, "Chapter 7 - Risk control and process safety management systems," in *Process Safety and Big Data*, S. Valeev and N. Kondratyeva, Eds., Elsevier, 2021, pp. 271–294. doi: https://doi.org/10.1016/B978-0-12-822066-5.00005-4.

[41] S. and L. A. L. and M. M. and M. F. and M. R. and S. L. Dragoni Nicola and Giallorenzo, "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*, B. Mazzara Manuel and Meyer, Ed., Cham: Springer International Publishing, 2017, pp. 195–216. doi: 10.1007/978-3-319-67425-4_12.

[42] A. Sadri, A. Rahmani, M. Saberikamarposhti, and M. Hosseinzadeh, "Fog data management: A vision, challenges, and future directions," *Journal of Network and Computer Applications*, vol. 174, p. 102882, Sep. 2021, doi: 10.1016/j.jnca.2020.102882.

[43] C. Wohlin, "Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, in EASE '14. New York, NY, USA: Association for Computing Machinery, 2014. doi: 10.1145/2601248.2601268.

[44] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of Systems and Software*, vol. 101, pp. 193–220, 2015, doi: https://doi.org/10.1016/j.jss.2014.12.027.