UDC 004.414.2:681.5.04
DOI: 10.15587/1729-4061.2026.359147

DEVELOPMENT OF SYSTEM-TYPE-CENTRIC METHOD FOR ARCHITECTURAL DESIGN OF COMPUTER-SOFTWARE AUTOMATION SYSTEMS

Ihor Polataiko
PhD Student*
ORCID: https://orcid.org/0000-0002-9111-0162
Leonid Zamikhovskyi
Corresponding author
Doctor of Technical Sciences, Professor,
Head of Department
E-mail: leozam@ukr.net
ORCID: https://orcid.org/0000-0002-6374-8580
*Department of Information and Telecommunication Technology and Systems
Ivano-Frankivsk National Technical University of Oil and Gas
Karpatska str., 15, Ivano-Frankivsk, Ukraine, 76019

The object of the study is the architectural design process and architectural documentation model of computer-software systems in the automation domain.

Existing architectural design methodologies are predominantly agnostic about the nature of computer-software systems, lacking the specific guidance required to address the fundamentally divergent architectural natures of heterogeneous computer-software systems in modern automation.

Within the scope of this study, a system-type-centric method of architectural design of computer-software automation systems is proposed. The proposed method introduces a comprehensive viewpoint-oriented architectural documentation model and a reproducible iterative architectural design process.

The method hierarchically structures architectural concerns at different levels, ensuring a systematic transition from abstract to concrete design. By defining a viewpoint applicability matrix across the different system types, the method ensures the creation of relevant, standardized, structured, consistent, traceable and interoperable architectural artifacts.

The method is based on the system-type-centric paradigm and the concepts of standards and approaches focused on architectural viewpoints. This combination is the main distinguishing feature of this method, which ensures the applicability of the method to a wide range of computer software systems, addressing the differences in the architectural nature of different types of systems.

The proposed method increases the interoperability of architectural design artifacts, defines a prescriptive process that structures architectural design and forms the basis for tools for automated AI-assisted architecture generation

Keywords: software architecture, automation systems, architectural documentation model, architectural design process

Received 04.02.2026
Received in revised form 02.03.2026
Accepted date 17.04.2026
Published date 30.04.2026

How to Cite: *Polataiko, I., Zamikhovskyi, L. (2026). Development of system-type-centric method for architectural design of computer-software automation systems. Eastern-European Journal of Enterprise Technologies, 2 (2 (140)), 65–84. https://doi.org/10.15587/1729-4061.2026.359147*

1. Introduction

The rapid evolution of Industry 4.0, Internet of Things (IoT), and Industry 5.0 has driven a fundamental shift in the development of automation solutions. Consequently, modern automation systems involve a complex, deeply intertwined combination of information technology (IT) and operational technology (OT) systems [1, 2]. The inherent difficulty of defining universally applicable architectural design methods with interoperable artifacts lies in the vast variability of the fundamental architectural aspects and natures of heterogeneous computer-software systems (CSS) in automation [3, 4]. Existing architectural design methodologies and approaches do not account for fundamental differences in the architectural nature of different types of systems used in automation. This critical gap prevents the formation of standardized and uniform design processes with interoperable artifacts capable of covering the divergent and heterogeneous aspects of modern computer-software systems, while remaining practical and prescriptive.

The presence of a structured and flexible method of the architectural design of computer-software systems in automation, would enable a more standardized and efficient architectural design for automation solutions and ensure the interoperability of architectural design artifacts. Furthermore, the establishment of such an architectural design method would contribute directly to leveraging the software architecture design process via automated tools. These tools could generate (design and develop) computer-software systems for automation or serve as an intelligent co-pilot for architects designing these complex systems.

Study devoted to processes and models of architectural design of computer-software systems is relevant and fundamentally important for forming effective and standardized approaches. This subject has become especially relevant in the era of rapid development of AI-based systems for software design and development, as it has acquired new application vectors for ensuring the efficiency and direction of such systems' operation.

2. Literature review and problem statement

The body of knowledge regarding architectural design and documentation is extensive, encompassing standardized

Copyright © 2026 Authors. This is an open access article under the Creative Commons CC BY license

meta-models, structural templates, and process-oriented frameworks. Industry-standard documentation templates such as the arc42 template [5] and the C4 model [6] provide robust structures for communicating software architecture. However, while these tools excel at organization and stakeholder communication, they are intentionally system-agnostic. They neither prescribe a design process nor explicitly address the fundamental differences in architectural nature across heterogeneous computer-software system types.

Standardization efforts led by ISO/IEC/IEEE 42010 have established a formal meta-model for architecture description, introducing essential concepts such as stakeholders, concerns, viewpoints, model kinds, and views. While it serves as the foundational basis for architectural description, it intentionally avoids prescribing concrete viewpoints or specific frameworks to maintain universal applicability. In contrast, ISO/IEC/IEEE 42020 shifts the focus toward architecture processes, defining activities and relationships across governance, management, and technical execution. Although it provides a high-level process framework, it remains abstract and does not prescribe a concrete, step-by-step method for designing heterogeneous system architectures.

Traditional view-based methodologies, including SEI views and beyond [7], viewpoints and perspectives [8], and RM-ODP (ISO/IEC 10746-1:1998; ISO/IEC 10746-2:2009; ISO/IEC 10746-3:2009), offer comprehensive conceptual toolsets for multi-perspective design. Classic models such as the “4 + 1” view model [9] describe a concrete set of system views but remain too generic, as they do not account for the fundamental variances between different types of software systems. Extensions of this model into process-plane frameworks, such as RUP [10] and RUP-SE [11], define design processes for software and systems engineering respectively. However, these are insufficiently adapted to differences in system architectural nature and, in practice, are often either overly heavyweight or lacking the specificity required for direct application in real-world contexts. Despite their utility, these approaches do not explicitly address architectural heterogeneity across IT and OT system boundaries, failing to account for the divergent architectural natures that occur at the intersection of information technology and operational technology in automation systems.

Attribute-driven design (ADD) [12] describes the method of architectural design centered on quality attributes of the system as the primary architectural drivers, and is aligned with the concepts described in [7]. While valuable for elaborating quality requirements and guiding architectural decisions, the method remains abstract. It lacks explicit guidance for adaptation to systems with fundamentally different architectural natures, leaving such tailoring to individual organizations.

In the domain of systems engineering, the functional architecture for systems (FAS) [13] proposes a structured approach to designing the functional architecture of a system and its functional decomposition in a manner independent of implementation details. The system architecture framework (SAF) [14], in turn, proposes a general, domain-independent framework for modeling and describing the architecture of technical systems in the context of MBSE (model-based systems engineering). Specialized frameworks such as the robotic architecture framework [15] propose domain-specific approaches and artifacts. These are complemented by lifecycle-centric resources such as the NASA Systems Engineering Handbook [16], as well as study in the field of model-based systems architecture (MBSA) [17], which examines approaches and practices of architectural design of systems in the context of MBSE.

The IoT-A architectural reference model [18] serves as a domain-specific reference model aligned with viewpoint-oriented documentation approaches according to ISO/IEC/IEEE 42010 and [8]. It provides structured guidance for architecture generation, including explicit activities and view derivation dependencies, yet remains methodology-agnostic.

Furthermore, various studies have proposed methods for deriving architectures from problem descriptions [19] or quality requirements [20]. A common limitation of these derivation methods is their siloed nature and focus on simplified transitions. They often lack a unified, iterative process that ensures interoperability across the heterogeneous system scales required in modern automation. Study [21] proposes an architecture-centric software development lifecycle, while study [22] analyzes widely adopted methods to propose a high-level process model. Both studies describe the framing of the process rather than offering a prescriptive workflow that can be followed to systematically transform requirements and constraints into architectural artifacts.

Works [5–22] present current approaches, standards, and research results regarding existing methods of architectural design and documentation of computer-software systems. It is shown that these approaches provide reliable conceptual foundations, process structures, and frameworks. However, issues remain unresolved related to the absence of prescriptive (concretized) processes and models of architectural design that could be directly applied to a wide range of systems. Existing methods are predominantly either focused on generic aspects, resulting in fragmented coverage, or provide only high-level framework models. This results in overly abstract approaches that require unique adaptation and elaboration for each specific case.

The reason for this may be the objective difficulties associated with covering architecturally significant aspects of heterogeneous systems (in particular, at the boundary of IT and OT systems). This is related to fundamental differences in the architectural nature of disparate systems, which makes the development of a single, detailed, universal method impractical without taking into account the fundamental architectural differences of various categories of systems.

A way to overcome these difficulties may be the application of an approach that classifies systems by their fundamental architectural nature and operational model. Such an approach is used in work [23], where the system-type-centric paradigm is proposed, which defines a taxonomic classification of computer-software systems, as well as a model of computer-software system elements and levels. However, while it establishes the necessary theoretical foundations and emphasizes the distinctions between CSS types, it does not define a concrete, standardized method or a reproducible process for the design and documentation of architecture.

All this allows asserting that it is appropriate to conduct the study devoted to the development of a prescriptive standardized method of architectural design of heterogeneous computer-software automation systems that takes into account fundamental differences in the architectural nature of systems of different types.

3. The aim and objectives of the study

The aim of the study is to develop a uniform, flexible, and standardized method for the architectural design and architectural documentation model of computer-software automation systems. Rooted in the system-type-centric paradigm, the proposed method aims to ensure structural consistency, artifact interoperability, end-to-end traceability, and universal applicability across heterogeneous computer-software automation system types. This will enable a structured and standardized transition from requirements and constraints to interoperable architectural design artifacts, which can be used both by architects and by automated AI-driven architecture generation tools.

To achieve this aim, the following objectives have been set:

– to formalize the conceptual foundations and structural principles of a system-type-centric architectural design method, including its viewpoint-oriented documentation model, hierarchical architectural scales, and design progression;

– to define a comprehensive, standardized set of architectural viewpoints covering architectural concerns at different scales, together with their inputs, dependencies, and recommended representation formats, forming a complete architectural documentation model;

– to define a prescriptive architectural design process by establishing the ordered application of architectural viewpoints and explicitly mapping their applicability across fundamentally different system types;

– to define formal rules governing the architectural design process across the system lifecycle, ensuring consistency, traceability, and controlled evolution of architectural decisions;

– to perform a quantitative assessment of the integral effectiveness of the proposed method by formalizing qualitative characteristics into measurable indices and to conduct a comparative analysis against generally accepted industry standards, methodologies, and frameworks of architectural design;

– to demonstrate the applicability of the proposed method on the example of architectural design of a heterogeneous computer-software industrial automation system.

4. Materials and methods

The object of the study is the architectural design process and architectural documentation model of computer-software systems (CSS) in the automation domain.

The main hypothesis of the study is that integrating the system-type-centric paradigm [23] with architectural standards (ISO/IEC/IEEE 42010) and viewpoint-oriented approaches [7, 8] facilitates the creation of a formalized, reproducible architectural design method. The method is capable of ensuring the production of structurally consistent, traceable, and interoperable artifacts across fundamentally heterogeneous computer-software systems in automation, including behavior-oriented, data-flow-oriented, continuous agentic, and continuous control systems.

Accepted assumptions in the work: the study assumes that formal problem-space artifacts – specifically, functional requirements, non-functional requirements (NFRs), and constraints – are available as standardized inputs to the design process. It is also assumed that requisite business domain exploration practices are conducted externally and successfully inform these inputs to the proposed method.

The scope of application of the method covers only the aspect of architectural design of computer-software systems, which means that other phases and aspects of architectural decision-making are outside the scope of this method. Such aspects include: the definition of business goals, domain knowledge exploration, requirements elicitation and management, planning of the system architecture implementation project, etc. This also means that the method can be considered modular and used within higher-level models and methodologies or in combination with them for the design and documentation of the computer-software systems architecture in automation. For example, in the areas of solution architecture, enterprise architecture, industrial process design, etc.

At the same time, the architectural model defined within the method allows flexibility in terms of the use of tools, lower-level models, and approaches for architecture modeling. This method has a clear focus and boundaries. It serves as a link between high-level models and standards and low-level tools. High-level standards do not cover concrete software architectural design processes. Low-level models and tools are used as direct means of architectural modeling.

The study was conducted utilizing qualitative, constructive theory-building methodologies. Because the study focuses on prescriptive process engineering and architectural documentation model formation, it did not rely on physical laboratory equipment or specific software simulation tools. Instead, the results were obtained through the systematic theoretical synthesis and adaptation of existing systems engineering standards.

The foundational design of the method was constructed by cross-referencing the system-type-centric paradigm [23] with the formal meta-models established in international standards (ISO/IEC/IEEE 42010, ISO/IEC/IEEE 42020).

To develop the architectural documentation model, a hierarchical decomposition method was applied. The architectural concerns and solution-space artifacts were divided into three strict hierarchical scales: system-level, subsystem-level, and component-level. Within these scales, a transition mechanism from abstract (logical) to concrete (physical) design was formulated by analyzing the required inputs and dependencies for multi-perspective design concepts.

The formulation of the viewpoint applicability matrix was achieved through a comparative mapping process. First, the fundamental architectural natures and corresponding aspects of four system types [23] were isolated. Then, they were theoretically matched against the structural, behavioral, and infrastructural concerns required for their implementation.

Finally, the rules governing the architectural design process were developed by applying lifecycle governance principles. This involved defining rigid sequential logic for viewpoint application and establishing mechanisms for continuous iteration, governance, and architectural compliance verification.

5. Results of the development of the system-type-centric method for architectural design of computer-software automation systems

5. 1. Core concepts behind the method and high-level structure

The core concept of the method lies in combining the concepts of standards (ISO/IEC/IEEE 42010) and approaches [7, 8] of architectural design oriented towards architectural viewpoints, and the concepts of the system-type-centric paradigm of architectural design [23].

In particular, the method contains:

– the architectural documentation model in the form of defined architectural viewpoints, that provide a set of standardized artifact definitions of architectural description of the computer-software automation systems. The resulting set of architectural artifacts for a particular system should contain a set of architectural views expressed through corresponding models, describing the architecture of the computer-software system in a standardized and interoperable way. The architectural documentation model is sufficient and versatile in the sense that it describes the details necessary for the implementation of the architecture as well as allows reasoning about the architecture of the system and performing an architectural analysis. The set of viewpoints is designed in the way to cover the system architecture on both application and infrastructural levels [23] at the scale of global-scope system, subsystems, and individual components requiring the design (custom application-level components; application-level of platform-like non-custom subsystems; non-custom platform-like application-level components);

– the description of the architectural design process defines a standardized sequence of stages and steps to be followed in order to design the architecture of the computer-software automation system. Following the process ensures the creation of system-specific architectural documentation artifacts describing the designed architecture. The concept behind the process aligns with principles of architecture conceptualization, elaboration, and evaluation defined in ISO/IEC/IEEE 42020, ensuring a systematic transition from problem characterization to a verified architectural description. The process has a common set of inputs to the architectural design process, which consists of requirements to the system, both functional requirements and non-functional requirements (NFRs) as well as a set of constraints. The goal of the process is to gradually move towards the fully documented system architecture by designing, at each step, artifacts that are parts of the system architectural documentation as well as serve as an input to the subsequent design steps.

Both the architectural documentation model and the architectural design process are adherent to the system-type-centric paradigm [23] and address the system-type-specific concerns and aspects, caused by the fundamental architectural nature.

The method hierarchically addresses the computer-software automation system architectural concerns starting from the highest-level (system-level-scale) to the lowest level (application-level components internal design and logic details). The following scale levels are part of the method:

– system-level-scale – at the scale of a global system being designed;

– subsystem-level-scale – at the scale of an individual subsystem within a global system;

– component-level-scale – at the scale of an individual custom or modified off-the-shelf application-level component. As per [23], an application-level component is defined as an independent deployment unit (a separate software application);

– unit-level-scale viewpoints specifying the logic details at the level of a single exposable logic unit, named behavior / processing, in scope of this study (only for required details).

At the scales of system / subsystems / components the method is structured as the movement from logical (abstract) design to physical (concrete) design. It allows considering first abstract technology-agnostic aspects and structure at the corresponding scale and then turn it into concrete technical design, ensuring a gradual approach where subsequent design steps can be based on the results of the previous design steps without missing logical links. Fig. 1 presents the high-level concept of the architectural design method.

The resulting architectural design artifacts are expressed in the form of views, representing the architecture from specific angles (different viewpoints) addressing particular concerns. The viewpoints are defined to be oriented towards global structural (at higher scale) and technical engineering concerns, representing the system architecture in the way that allows both – implementation of the system based on the architectural documentation as well as reasoning about the architecture itself.

At each scale, a particular set of viewpoints is defined to cover the relevant structural and technical engineering concerns (Table 1).

Fig. 2 presents the architectural design method's concerns at different scales.

Table 1

Architectural stakeholder concerns at different scale levels

Scale	Stakeholder	Concern
1	2	3
System-scale	Business Analysts / Enterprise Architects / System Analysts / System Engineers	Problem space definition: business requirements and problem statements
	Business Analysts / Enterprise Architects / System Analysts / System Engineers	Identification and clarification of business-level capabilities informing system capabilities
	Business Analysts / System Analysts / System Engineers	Functional completeness of requirements against identified capabilities
	Business Analysts / System Analysts / System Engineers / System Architects	User experience flows
	Enterprise Architects / Business Analysts / System Analysts / System Engineers / System Architects	Non-functional requirements and constraints specification
	Enterprise Architects / System Architects	Alignment of the system within the enterprise landscape
	Enterprise Architects / System Architects	Sourcing strategy: SaaS vs. custom vs. off-the-shelf decisions
	Enterprise Architects / System Architects	Definition of system boundaries and external integrations
	Enterprise Architects / System Architects	System topology and subsystem decomposition
	Enterprise Architects / System Architects	System type classification and architectural nature (behavior-oriented, data-flow, continuous)
	Enterprise Architects / System Architects	Global technical constraints and system-level quality attributes

Continuation of Table 1

1	2	3
Subsystem-scale	System Architects / Software Architects	Mapping of capabilities to logical subsystem structures
	System Architects / Software Architects	Classification of logical components and sourcing strategy at subsystem-level
	System Architects / Software Architects	Technology stack selection
	System Architect / Software Architects / Infrastructure Engineers	Infrastructure platform and cloud services selection
	System Architect / Software Architects / Infrastructure Engineers	Deployment topology and environment isolation
	System Architects / Software Architects	Definition of API standards and communication protocols
	Software Architects / Control Engineers	Modeling of control dynamics and feedback loops
	System Architects / Software Architects	Definition of missions for continuous agentic systems
	Control Engineers / System Architects / Software Architects	Analysis of physical constraints influencing control logic
	System Architect / Hardware Engineers	Hardware components design
Component-scale	Software Architects / Software Engineers	Definition of application-level interface contracts in form of APIs and software-to-hardware communication specifications
	Software Architects / Control Engineers / Software Engineers	Logical semantics of I/O signals and control states
	Software Architects	Design of orchestration, selection of component-global patterns and styles
	Software Architects / Software Engineers	Internal modular decomposition of components
	Software Architects / Software Engineers	Execution model: concurrency, tasks, and runtime behavior
	Software Architects / Control Engineers / Software Engineers	States and stages modeling and design
	Software Architects / Data Engineers / Software Engineers	Design of in-application and persistent data models
Unit-level (detailed design)	Software Engineers	Implementation details of procedures and transformations
	Test Engineers	Testability of logic, states, and edge cases
	Test Engineers	Verifiability of behaviors, processings, and stages
	Data Engineers	Correctness of calculations and data transformations

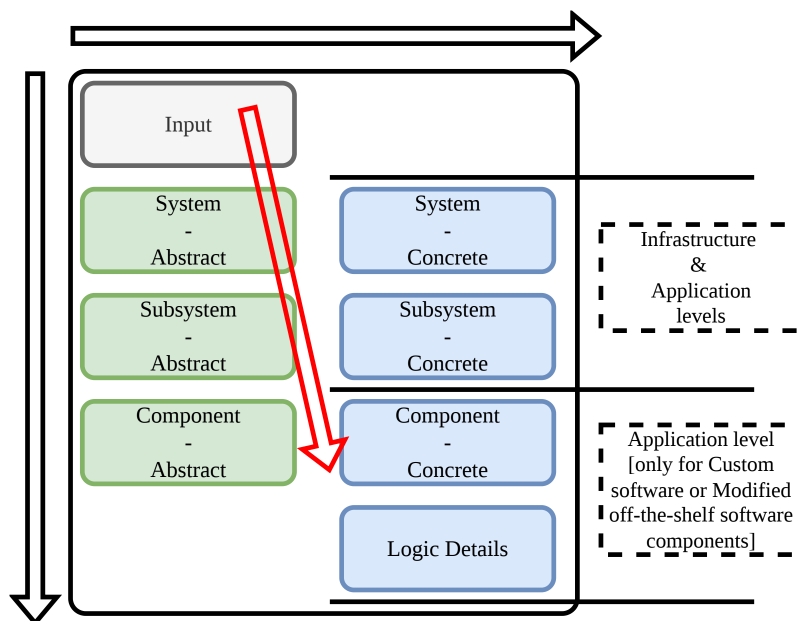

Fig. 1. High-level concept of architectural design method

System capabilities, which are identified based on the requirements, serve as a decomposition subject, being mapped to subsystems on system-scale and to logical (abstract) components on subsystem-scale. System capabilities mapping should ensure cohesion and compliance with NFRs and constraints. Another decomposition criterion on the system-scale is system-type-enforced boundaries, which are based on the fundamental architectural nature of the system parts [23].

Since the viewpoints are defined generically by the method proposed in this study, it's important to leave the space for system-specific governance to be defined. For this specific purpose, terminology compliant with ISO/IEC/IEEE 42010 was extended by the notion of governance documents. The governance document represents the global rules behind the viewpoint artifacts creation, maintenance and modifications within the scope of the system (covering architectural aspects of architectural governance, standardization, and maintainability).

Fig. 3 outlines the relationship between architectural description artifacts and respective governance documents.

Each view created within the scope of the architectural artifacts of the designed system should have a link to the corresponding system-specific governance document for the respective viewpoint. Generally, the relationship within the scope of the system's artifacts: many views (of a certain viewpoint) – one governance (of a corresponding viewpoint).

Fig. 2. Architectural design method's concerns at different scales

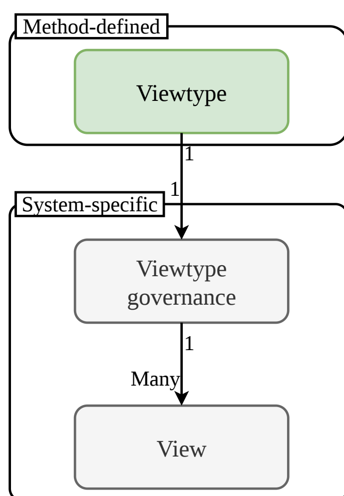

Fig. 3. Relationship between architectural description artifacts and respective governance documents

5. 2. Architectural viewpoints defined in scope of the method

A number of viewpoints was defined in scope of the study, which are applicable either globally across system types or specifically to certain types – the applicability of specific viewpoints defined is described later in this article.

Each viewpoint description also mentions "recommended documentation formats", which can be considered conceptually similar to "model kinds" defined by ISO/IEC/IEEE 42010.

More than one representation can be created per viewpoint.

System-scale inputs into the architectural design, which are described as specifications, are listed in Table 2. These specifications are artifacts that describe the problem space of the architectural design.

Table 3 describes a viewpoint operating on System-scale abstract level and oriented onto capabilities definition. The viewpoint serves as a link between problem space and solution space, forming the first round of architectural inputs processing.

Table 4 presents a viewpoint defined at System-scale concrete level, outlining decomposition of the system into subsystems and mapping of capabilities to eachsubsystem.

Table 2

System — Input

ID	Specification	Description	Scale	Based upon viewpoints
I-1	Functional requirements viewpoint	Specification of functional requirements. Recommended formats: requirements list, use-case list, feature catalog, etc.	System	–
I-2	NFR and constraints viewpoint	Specification of NFRs and constraints (both technical and business). Recommended formats: quality attribute scenarios, constraints catalog, NFR matrix, etc.	System	–

Table 3

System — Abstract

ID	Viewpoint	Description	Scale	Based upon viewpoints
SA-1	Capabilities viewpoint	High-level capabilities (functionalities) of the system, based on the requirements. Capabilities are defined according to the system type: – behavior-oriented systems: a cohesive group of operational functions (behaviors) demanded from the system, which should belong to the same component; – data-flow-oriented systems: a cohesive group of data-processing functionality demanded from the system; – continuous systems: a function or control task demanded from the system. Types of capabilities: – business-related capabilities – primary; arise from business requirements, system functional requirements, business capability models; – technical capabilities – derived; capabilities which are necessary to deliver primary capabilities; these are new capabilities, necessary to deliver business-related capabilities, that emerge during the design of system's architecture. Capabilities definition, from the problem-space perspective, may be informed and refined by business domain exploration practices such as Domain-Driven Design subdomain discovery [24] and Event Storming [25] aimed at analysis of the business domain targeted for automation. These practices aim to identify business subdomains and domain events, which contribute to clarification and refinement of functional requirements and constraints. The outputs of such discovery activities are expected to be reflected in the «System – Input» specifications (I-1, I-2). The architectural method assumes that problem-space exploration precedes or iteratively co-evolves with architectural design, and that resulting insights are incorporated into the input specifications before capabilities are formalized in the SA-1 Capabilities viewpoint. Business architecture discovery practices are considered outside the scope of the present method and are treated as external problem-space processes supplying structured input into architectural design. Recommended formats: capability map diagram, capability hierarchy diagram, etc.	System	I-1, I-2

Table 4

System — Concrete

ID	Viewpoint	Description	Scale	Based upon viewpoints
SC-1	Subsystems viewpoint	Representation of the subsystems' composition. The view should include mapping of capabilities to subsystems. "Emergent capabilities" / "technical capabilities" – might arise during assignment and analysis of the capabilities defined in Capabilities view to the subsystems or external systems. In some situations it's possible to make decisions on custom vs adopted concept at the stage of the subsystems view design. If it's possible to make a decision about the presence of the off-the-shelf self-hosted subsystem, integrated 3rd party and SaaS systems to cover capabilities, subsystems in this view are classified into: – custom; – non-custom: off-the-shelf self-hosted; – non-custom: external system integration. A system type, according to [23], should be assigned to each custom subsystem. Integrated SaaS systems and 3rd party systems are treated out of the scope of the design system (external integrations) and not treated as subsystems. But these systems may have the capabilities assigned to them. Any self-hosted subsystems are represented within the global system's boundary, while external integrations like SaaS and 3rd-party systems are represented outside of the global system's boundary. Recommended formats: system landscape diagram [6], capability-subsystem mapping table. The model might, optionally, include Actors on the diagram to represent which of the subsystems are user-facing and which are internal and available only to other subsystems. Basic UX-related notes on actor's specifics of interactions with the system are also acceptable. Notations and indications of integrations with external systems as well as integrations with physical/virtual concepts are acceptable. Notations of «Gateways» represented as a circle are acceptable to show that all downstream subsystems, connected to the gateway, integrate with all upstream subsystems. Recommended to leverage color coding to visually differentiate the system types [23] of custom subsystems, for e.g.: orange – continuous control-oriented, yellow – continuous agentic, green – data-flow-oriented, blue – behavior-oriented; grey – off-the-shelf self-hosted	System	I-2, SA-1

Subsystem-scale abstract viewpoints (Table 5) are dedicated to describe the concerns of subsystem-wide aspects like control within the subsystem as well as logical decomposition of the system on abstract components.

Table 5

Subsystem – Abstract

ID	Viewpoint	Description	Scale	Based upon viewpoints
1	2	3	4	5
SSA-1	Control viewpoint	Control-related design and math models. Recommended formats: control block diagrams, control mathematical models' descriptions	Subsystem	I-1, I-2, SC-1
SSA-2	Agent Mission viewpoint	Represent the goal-achievement strategy (strategies) that agent follows. Consists of Stages, which represent the steps of the goal achievement process (usually, include iterations – loops; as agents often progress iteratively). Recommended formats: activity diagram (UML – Unified Modeling Language) [26]	Subsystem	I-1, I-2, SC-1
SSA-3	[Logical / Abstract] High-level components viewpoint / [Logical / Abstract] High-level components – Data Flow viewpoint	Describes abstract logical components. Logical components should represent capabilities-implementing units. Primary concern of the viewpoint is to outline the logical groupings of functionality within each system: each of them forming a potential of becoming a physical component of application-level (with corresponding infrastructural software components) or a module of physical application-level component, or being covered by infrastructural software, or being covered by SaaS integration. Should include mapping of capabilities to logical components. Viewpoint is primarily based on subsystem capabilities (SC-1) and might require decomposition of capabilities into more fine-grained capabilities with some degree of influence by NFR and constraints (I-2). For continuous agentic systems additionally based upon SSA-2 and for continuous control systems additionally based on SSA-2. SSA-3 is a transitional design viewpoint (design-time reasoning artifact), which is not included in the final set of architectural deliverables (suppressed by SSA-3-ext) and should no longer be maintained after the initial architectural design. Recommended formats: logical component interaction diagrams, data flow diagrams. Might, optionally, show actors on the diagram to represent which of the subsystems are user-facing and which are internal and available only to other subsystems. Notations and indications of integrations with other systems as well as integrations with physical/virtual concepts are acceptable. Notation of "Gateways" represented as a circle is acceptable to show that all downstream subsystems integrate with all upstream subsystems. Acceptable to represent a couple of subsystems on one diagram for smaller systems; Diagram per subsystem is recommended for systems with many components and subsystems. Connector notations on SSA-3 should represent the following: – behavior-oriented – show interactions; – data-flow-oriented – show data flow; – continuous-agentic – show interactions; – continuous-control – show interactions via control mapping to logical components	Subsystem	I-2, SC-1, SSA-1, SSA-2
SSA-3-ext	[Logical / Abstract] Components viewpoint with components classification into Custom / Off-the-shelf / SaaS	This view takes SSA-3 as an input and classifies each logical component as: – custom; – off-the-shelf; – external system integration. Infrastructure-level SaaS (usually, cloud provider's services) are treated as part of the designed system, not as external integration. Application level SaaS are treated as external integrations and moved outside of the system's boundary. Decision on whether SaaS services belong to application-level or infrastructure-level, in some cases, can be vague: e.g., decision whether identity federation provider is application-level or infrastructure-level. In this case it's up to the architect's perspective to decide – recommended to assign to application-level and treat as external SaaS if the system infrastructure will be planned on different cloud or on-premises, and assign to infrastructure-level if other infrastructure would be based on the same cloud provider. If, upon view construction, it's evident that Custom logical components within the subsystem exhibit fundamental nature of system types [23] – the split into subsystems and reiteration is necessary. Recommended formats: logical component interaction diagrams (with classification), data flow diagrams (with classification). Recommended to leverage color coding to outline components classification on the diagram, e.g.: white – custom application-level with corresponding infra-level; grey – off-the-shelf self-hosted application-level with corresponding infra-level; cyan – off-the-shelf self-hosted infra-level; purple – infra-level SaaS; grey – external system integration (should be relocated outside of system boundaries if the logical component was previously within)	Subsystem	I-2, SSA-3, SSA-1, SSA-2

Continuation of Table 5

1	2	3	4	5
XA-1	UX viewpoint	Outlines UX (user experience) aspects. UX aspects can be represented cross-subsystems – on the system-level in case the UX flow encompasses several subsystems. Recommended formats: – [UX] User journey; – [UX] Use-case diagram; – [UI] Web UI screens; – [UI] HMI screens	Subsys-tem / System	I-1, I-2, SA-1, SC-1, SSA-3-ext

"XA-" and "XC-" viewpoints represent architectural concerns that may be instantiated at system or subsystem scope, depending on the architectural context.

Table 6 describes a component-scale abstract viewpoint, outlining the logical operations required to be performed by the component and available as part of its contract, which define the reason for the component's existence for components of certain system types. It's relevant only for custom components or for modified off-the-shelf components (which might require some reverse engineering to describe architecture of the existing system). Also might be applicable for leveraged platform-like adopted application-level components (or platform-like subsystems), when components allow defining new behaviors (exposed through APIs or triggered internally).

Behavior is defined as an abstract operation that a system is capable of performing, which is externally exposable: as API / action of the behavior-oriented system's component.

Processing is defined as an abstract data processing task (data stream event processing or batch data job processing), which will be externally exposable or externally triggerable by a data-flow-oriented system's component.

Action, in scope of this study, is a user-exposed ability (via human-machine interface / user interface), e.g., button on the user interface, triggering behavior.

Subsystem scope concrete level viewpoints are described in Table 7. These viewpoints are dedicated to outline implementable architecture with concrete details on the subsystem level, covering the aspects of both application-level and infrastructure-level as well as some cross-cutting aspects like deployment strategies.

Table 6

Component — Abstract

ID	Viewpoint	Description	Scale	Based upon viewpoints
CA-1	Behavioral viewpoint	Catalog of behaviors (for behavior-oriented system's components) or processings (for data-flow oriented system's components). CA-1 is a transitional design viewpoint (design-time reasoning artifact), which is not included in the final set of architectural deliverables (suppressed by CA-1-ext) and should no longer be maintained after the initial architectural design. Recommended formats: behavior catalogs, processing catalogs	Component	I-1, I-2, SSA-3-ext
CA-2	Behavioral Dynamic viewpoint	Supportive viewpoint representing how certain functionality should be achieved across sequence of behaviors or processings. Required for the cases where functional requirement or technical means of achieving certain functionality require detailing on the cross-behaviors flow level. Recommended formats: sequence diagram (UML), activity diagram (UML)	Subsystem / System. Usually, certain component serves as an entry point to a certain dynamic cross-behavioral flow, requiring detailing	I-1, I-2, SSA-3-ext, CA-1

Table 7

Subsystem — Concrete

ID	Viewpoint	Description	Scale	Based upon viewpoints
1	2	3	4	5
SSC-1	[Physical] Software components viewpoint (application & infrastructure software & infrastructure platform)	The primary concern of this viewpoint is the software components (units of deployment) to be deployed: including application and infrastructure levels (software & platform). The view includes infrastructure platform, where applicable, to showcase the environment of the runtime of application software and infrastructure software. The view might include hardware components – sensors and actuators for Control systems. The view may show nested items in infrastructure software like queues on the broker or buckets in storage service. The view is based primarily on the SSA-3-ext viewpoint. Usually, one custom logical component is mapped to one physical application-level component (with corresponding infrastructure-level components). Multiple logical components can be mapped to one physical application-level component and become modules in case of custom application-level software. One logical component, as well, might be decomposed into a couple of physical software components of application-level or infrastructure-level. Recommended formats: «C4 container diagram» [6], etc.	Subsystem	I-2, SSA-3-ext, XA-1, CA-1 (for hardware-first systems XC-2)

Continuation of Table 7

1	2	3	4	5
XC-2	Infrastructure viewpoint (hardware & platform; cloud services infra)	The primary concern of this viewpoint is hardware components, platform infrastructure software, and cloud services to form an infrastructure for the system. The view creation might be skipped for some systems that are truly hardware-agnostic. For some systems, like embedded computer-software systems or computer-software systems with cloud-native infrastructure, this view is mandatory. Recommended formats: deployment diagram (UML), vendor-notations-specific cloud diagrams, hardware components diagrams, networking diagrams, printed circuit boards designs, etc.	Subsystem / System	I-2, SC-1, SSA-3-ext (for software-first systems SSC-1)
XC-3	Deployment viewpoint	The viewpoint is dedicated for specifying how the system is deployed onto infrastructure. Is especially relevant for systems with non-deterministic deployment strategies, for example when one software system can deploy another software system on demand within the specified hardware/platform/cloud infrastructure and there is a need in outlining the architecture of the deployment. Recommended formats: deployment flow diagrams, etc.	Subsystem / System	I-2, SSC-1, XC-2

Some subsystem-scale viewpoints are also applicable to non-custom (“off-the-shelf self-hosted” / SaaS) subsystems. Those include:

– XA-1 for detailing general UX flow;

– XC-2 for infrastructure architecture representation (only for “off-the-shelf self-hosted” systems);

– XC-3 for deployment architecture representation (only for “off-the-shelf self-hosted” systems);

– SSC-1 for physical components representation specifying which software components form a subsystem (only for “off-the-shelf self-hosted” systems).

These representations are necessary for detailing the general user experience (XA-1) and for detailing the infrastructure component (XC-2, XC-3, SSC-1).

Table 8 defines a design viewpoint, which is an intermediary between component’s behaviors / processings, defined in CA-1, and detailed API / contract / actions specifications (which will be defined by CC-4 viewpoint). At this stage, the inputs from CA-1 as well as infrastructural and application-level decisions (SSC-1, XC-1) are taken into consideration. This is necessary to define the transport and API mechanism for exposure of the behaviors / processings. Low-level specification details remain out of context.

For custom or modified self-hosted application-level components, concrete component-scale designs, described based on the viewpoints defined in Table 9, are necessary to cover the technical implementation concerns and details required at this scale level.

An application-level component (defined as an independent deployment unit / software application [23]) might consist of multiple modules. Module, in scope of the given study, is defined as code organization / compilation unit within a component. Table 10 outlines the low-level viewpoints related to specific behaviors / processings / stages, which require design-time detailing description on their logic or mathematical aspects.

Table 8

Component – Abstract-to-Concrete

ID	Viewpoint	Description	Scale	Based upon viewpoints
CA-1-ext	Generic APIs viewpoint [Reiteration on CA-1]	Represents updated behaviors / processing catalogs (defined by CA-1), with added protocols / API standards / access ways (for actions) details for each Behavior/Processing. E.g.: Kafka event, REST API (representational state transfer application programming interface), GraphQL API, WebForm, WebPageRequest, OPC UA (open platform communications unified architecture), scheduled, UserInterfaceAction, etc. Recommended formats: behaviors / processings catalogs (with protocol / API / Access-way details included)	Component	I-2, CA-1, SSC-1, XC-2

Table 9

Component – Concrete

ID	Viewpoint	Description	Scale	Based upon viewpoints
1	2	3	4	5
CC-1	Execution viewpoint	The primary goal of this viewpoint is to describe the concurrent tasks and their interactions during runtime (within the same component scope): RTOS (real-time operating system) tasks, PLC (programmable logic controller) organization blocks, cross-instance interactions (across instances of the same application-level component scaled horizontally) and tasks distribution in clustered processing. Recommended formats: execution diagrams, component diagrams (UML), activity diagrams (UML), sequence diagrams (UML)	Component	I-2, CA-1, SSA-1, SSA-2, SSC-1, XC-2, XC-3

Continuation of Table 9

1	2	3	4	5
CC-2	States / Stages viewpoint	Represents the internal stateful nature of the components: discrete states that component can be in internally or stages that component can follow. Might include descriptions of finite-state-machines (FSMs), behavior trees, PLC sequential function chart steps (one of the ways to implement FSM in PLC program), etc., with descriptions of logic at each state (traced to «agentic mission viewpoint» or «control viewpoint», if applicable). Each state or stage might have specific activities or control rules. For control systems, while control models (SSA-1) describe system dynamics over time, the CC-2 view would describe discrete modes governing which control rules apply. Recommended formats: state machine diagrams (UML), behavior tree diagrams, etc.	Component	I-2, SSA-1, SSA-2
CC-3	Component structural viewpoint	Represents modules of the custom application-level component. Recommended formats: «C4 component diagram» [6], etc.	Component	I-2, CA-1, SSC-1, CC-1, CC-2
CC-4	Contracts / APIs / Actions viewpoint	Describes detailed contracts / APIs / actions specifications, serving as an input to development based on API-first approach. For complex distributed systems, events should be centrally documented in the event catalog. Service-owned specifications should only reference the events from the events catalog. Recommended formats: OpenAPI, AsyncAPI, custom contract spec format, actions catalog, interface methods docs, etc.	Component / Module (if reiterated, in cases of monolithic components with many modules)	CA-1-ext
CC-5	Integrations APIs viewpoint	Represents integrated systems. Integrated systems can be: – internal; – external. Cloud services belong to the infrastructure level of the designed system and are not considered integrations. Recommended formats: for external / not-owned APIs, the documentation in OpenAPI / AsyncAPI / Postman / Bruno formats is required. For owned APIs (which are part of another custom application-level component within the global system), the API itself would be documented by the API-owning component: in this case, only tracing to the leveraged endpoints is necessary. A simple catalog of references to OpenAPI spec endpoints of API-owning services might be enough; otherwise, API-hub might be leveraged for centralized management of internal integrations as part of a supportive infrastructure	Component / Module	CA-2
CC-6	Domain data model viewpoint	Represents in-application domain data model – usually based on classes or structs; also includes structures like PLC tags. Recommended formats: entity relationship diagram (ERD), class diagram (UML), PLC tags catalog, etc. «State machine diagram (UML) can be additionally included to represent the rules of state transitions of an entity state»	Component / Module	I-2, CA-1, SSA-1, SSA-2, SSC-1, CC-1, CC-2
CC-7	Persistent data model viewpoint	Represents a data model stored in persistent data storage – usually, database schema. This view is important for application-level components that operate with persistent data models – primarily, in the context of behavior-oriented systems. Alternatively, such a view can be important from the system/subsystem perspective, in the context of preserving certain data within the system – primarily for data-flow-oriented systems. In the general case, such a view is associated with the application-level component that owns the persistent data model. If the model is defined at the system/subsystem level, then references to the same data model are added for different application-level components that operate with the model. In some cases, the persistent data model can be defined at the system/subsystem level without binding to application-level components. The persistent data model itself is implemented by an infrastructure-level component – in most cases, a database. Recommended formats: ERD	Component / Module. In some cases – Subsystem / System (in such case CC-6 might be dependent on CC-7 and partially or fully based on CC-7)	CC-6, I-2, CA-1, SSC-1, XC-2
CC-8	I/O signal interfaces and pins viewpoint	Represents software-to-hardware link: describes the pins (or I/O ports for PLC) and their purpose from application-level software point of view. Recommended formats: pins catalog with descriptions of purpose / expected voltage or current / pin types (digital, analog, digital interrupt, etc.), hardware schematics diagram	Component	I-2, SSA-1, SSC-1, XC-2
CC-9	Behaviors orchestration flow viewpoint	Dedicated for behavior-oriented orchestration-oriented automation systems. Used to describe the stateful orchestration systems, for example processes in Camunda BPMN (business process model and notation). The processes in such a system might have long running stateful process-instances outlining business processes being automated and invoking multiple behaviors over process-instance lifecycle (cross top-level behaviors). Recommended formats: BPMN 2.0, etc.	Component	I-1, I-2, SA-1, SC-1, CA-1

The description of CSS architecture from the viewpoints of behavior/processing/stage details is necessary only in certain cases and only for certain parts of the system. These include those parts where the specifics of the business logic, calculations, or aspects of work at specific stages have an architecturally significant component and require detailing and elaboration. In most cases, the information necessary for the implementation of business logic is a component of the requirements and is implemented based on them, within the context of the designed architecture.

Table 10

Behavior / Processing / Stage – Details

ID	Viewpoint	Description	Scale	Based upon viewpoints
D-1	Procedure viewpoint	Describes the conceptual logic flow behind one specific behavior. Recommended formats: sequence diagram (UML), activity diagram (UML), etc.	Behavior	I-1, I-2, CA-1, CA-2 (D-1 might detail logic of integration flow execution from the perspective of certain behavior by zooming in)
D-2	Calculation viewpoint	Describes the math behind one specific behavior / processing / stage. Recommended formats: mathematical notation, MATLAB scripts (The MathWorks, Inc., USA), etc.	Behavior / Processing / Stage	I-1, I-2, CA-1
D-3	Task / Transformation viewpoint	Describes the rules of logic or data transformation rules behind one specific processing / stage. Recommended formats: activity diagram (UML)	Processing / Stage	I-1, I-2, CA-1

It is possible that non-custom application-level components (off-the-shelf self-hosted / SaaS) are platform-like or form a platform-like system (Off-the-shelf or externally integrated system). So, even though they are adopted and are not developed from scratch, some level of on-top development, configuration, and customization might be needed. Such application-level components are not modifications of implementation of adopted application-level components (which, according to [23], would make them treated the same way as custom application-level components); they are extensions made usually by in-system configurations and scripting (especially, in the case of SaaS).

For non-custom platform-like subsystems, the design at the component scale is performed for the level of the entire subsystem. This is because, within the framework of the method, decomposition of such systems into components is not performed.

The architectural design at the component scale for platform-like non-custom components (subsystems / integrated external systems) is usually expressed through CA-1, CA-1-ext, CC-2, CC-3, CC-4, CC-5, CC-6, CC-9 (also CC-7 – not from the perspective of the data model design but from the perspective of data model leveraging, especially for the case of integrating existing external-to-the-component persistent data model). Also, potentially, application of D-1, D-2, D-3 is possible. Examples include the following platform-like non-custom subsystems, in the design of which it is necessary to involve elements of design at the component scale:

– SCADA (supervisory control and data acquisition) servers – e.g., tags definitions belonging to CC-6 and possible API extensions, defined via CA-1-ext, CC-4 and CC-5. For example, custom behaviors (implemented as custom scripts) that can be invoked through MQTT (message queuing telemetry transport) API for WinCC Unified SCADA (Siemens AG, Germany) [27];

– ERP platforms (e.g., SAP) – e.g., entities defined via CC-6, or API extensions defined via CA-1-ext and CC-4, as well as integrations defined in CC-5;

– CRM platforms (e.g., Salesforce) – e.g., entities defined via CC-6, or API extensions defined via CA-1-ext and CC-4, as well as integrations defined in CC-5;

– CMS platforms (e.g., Wordpress) – e.g., entities defined via CC-6, or API extensions defined via CA-1-ext and CC-4, as well as integrations defined in CC-5;

– etc.

5. 3. Architectural design process and viewpoints application to different system types

Table 11 presents the applicability of defined viewpoints for different system types according to [23].

Table 11

Matrix of architectural viewpoints applicability to computer-software system types

#	Behavior-oriented	Data-flow-oriented	Continuous agentic	Continuous control
1	2	3	4	5
System – Input				
1.1	I-1 and I-2			
System – Abstract				
2.1	SA-1			
System – Concrete				
3.1	SC-1			
Subsystem – Abstract [General]				
4.1	–	–	SSA-2	SSA-1 (optional)
4.2	XA-1 (optional)			
Subsystem – Abstract [Logical structure]				
5.1	SSA-3	SSA-3	SSA-3	SSA-3
5.2	SSA-3-ext	SSA-3-ext	SSA-3-ext	SSA-3-ext
Component – Abstract				
6.1	CA-1	CA-1	–	–
6.2	CA-2 (optional)	CA-2 (optional)	–	–
Subsystem – Concrete				

Continuation of Table 11

1	2	3	4	5
7.1	SSC-1	SSC-1	SSC-1	SSC-1
7.2	XC-2 – subsystem / system scope			
7.3	XC-3 – system / subsystem scope (optional)			
Component – Concrete				
8.1	CC-1 (optional)	CC-1 (optional)	CC-1 (optional)	CC-1 (optional)
8.2	–	–	CC-2 (optional)	CC-2 (optional)
8.3	CC-3 (optional)	CC-3 (optional)	CC-3 (optional)	CC-3 (optional)
pre 8.4	CA-1-ext (optional)	CA-1-ext (optional)	–	–
8.4	CC-4	CC-4	–	–
8.5	CC-5 (optional)	CC-5 (optional)	CC-5 (optional)	CC-5 (optional) [mostly, might be leveraged by virtual control systems]
8.6	CC-6 (optional)	CC-6 (optional)	CC-6 (optional)	CC-6 (optional)
8.7	CC-7 (optional)	CC-7 (optional)	CC-7 (optional)	–
8.8	–	–	–	CC-8 [for physical control systems]
8.9	CC-9 (optional)	–	–	–
Behavior / Processing / Stage – Details				
9.1	D-1 (optional)	–	–	–
9.2	D-2 (optional)	D-2 (optional)	D-2 (optional)	–
9.3	–	D-3 (optional)	D-3 (optional)	–

The sequence of rows outlines a progression from viewpoints of higher-level scopes to lower-level scopes and from abstract to concrete. If followed top to bottom, Table 11 defines the architectural design process where each subsequent step (creation of view of specific viewpoint) relies on the artifacts of the previous steps. Reiteration on previous steps due to the new architectural details being unveiled is acceptable and common. The template represents the architectural design of the computer-software system.

Note: the row number in Table 11 outlines recommended sequence of stages and steps of architectural design and documentation creation: number before the dot symbol represents the stage and number after represents the step (e.g., 2.3 – stage 2, step 3).

5. 4. Architectural design process rules

Rules of the architectural design process as part of the method:

1. The process is iterative. After each stage and step, it is possible to return to the previous stages and steps to reiterate and introduce adjustments.

2. Designing and documenting an architectural view includes the mandatory creation or review and, if necessary, adjustment of the corresponding governance document artifact.

3. A list of "Assumptions and Questions" is maintained, with answers obtained through:

– discussions with project stakeholders;

– discussions with domain experts;

– discussions with external technical experts;

– validation of hypotheses using proof of concept (PoC) and prototypes.

These questions should be related to specific architecturally significant [12] aspects of the system (technical and requirements-based). The answers will affect the architectural design and record the evolution of the system concept, in addition to architectural decision records (ADR). "Assumptions and Questions" records should be linked to the corresponding ADRs. This will allow recording the context of decisions made and ensure full traceability of the architecture. This approach reflects the evolution of the system concept based on obtained answers and verified assumptions.

4. At each architectural design stage, PoCs and prototypes may be created to validate technical assumptions and compare architectural options for particular aspects.

5. During initial design of the system, significant decisions, preferably to be outlined as ADRs, which brings additional context details to the reasons behind the solution represented by the architectural views and governance documents.

6. During ongoing maintenance the architecture of the system on the scales of system and subsystem should be reviewed upon changes to the architecturally significant requirements [12]. The architectural design on component-scale can be refined often by engineering teams receiving the new requirements related to specific architectural artifacts.

7. During ongoing maintenance of the system architecture documentation, any changes to system-level or subsystem-level views require the creation of corresponding ADRs.

8. Validation of the architecture against system requirements, especially non-functional requirements (NFRs), must be performed at the end of each architectural design stage.

9. Verification of the system implementation's compliance with the designed architecture is performed using standard practices, such as implementation reviews and the application of static analysis tools where applicable (e.g., ArchUnit).

10. Verification of the completed system's architectural compliance with requirements is carried out through testing against functional and non-functional requirements. In addition, by monitoring the system in the operational environment (continuous NFR compliance) [28] using various monitoring tools and systems (these systems belong to the supporting infrastructure category [23]).

5. 5. Quantitative assessment of the integral effectiveness of the proposed method

To ensure objectivity in evaluating the effectiveness of the proposed system-type-centric method, its qualitative characteristics were transformed into measurable quantitative indicators. The assessment is based on metrics that reflect the key requirements for the design of heterogeneous computer-software systems (CSS) in automation: process prescriptiveness, completeness of architectural decomposition, adaptability and algorithmic readiness for integration with artificial intelligence (AI-Readiness).

The comparative analysis is conducted exclusively within the subject area of this study – the design of heterogeneous CSS for automation systems. Basic standards (e.g., ISO/IEC/IEEE 42010) are flexible general-purpose meta-models. Their assessments according to the given criteria reflect not the shortcomings of these standards, but their deliberate methodological abstractness compared to the specialized prescriptive method being proposed.

For comparative analysis, generally accepted industry standards, frameworks, and architectural models are involved: the ISO/IEC/IEEE 42010 standard, the SEI views and beyond [7] and ADD [12] methodologies, the arc42 design template [5], the C4 model [6], the functional architecture for systems (FAS) approach [13], the system architecture framework (SAF) [14], as well as the IoT-A architectural reference model [18].

The evaluation is performed using four formalized indices, normalized on a single discrete scale from 1 to 4. The use of a discrete scale ensures the reproducibility of the evaluation and allows for correct relative comparison of architectural approaches.

The process prescriptiveness index (PPI) evaluates the presence and determinism of the design process:

– 1 – absence of an embedded process (purely descriptive meta-standard, notation, or documentation template);

– 2 – presence of general abstract guidelines for the design process at the meta-level;

– 3 – partially formalized or phase-based iterative approach;

– 4 – strictly deterministic process with clearly defined rules, transition matrix, and dependencies.

The completeness of architectural decomposition (CAD) evaluates the level (from abstraction to specifics or from high-level aspects to implementation details) and hierarchy of the system model:

– 1 – absence of formalized levels and hierarchy (the method operates with a flat set of concepts or artifacts without rigidly defined rules of structural division);

– 2 – basic one-dimensional decomposition (structuring occurs only along one vector: e.g., exclusively splitting into functional blocks or only logical decomposition, without binding to physical or infrastructural levels);

– 3 – extended structural decomposition (e.g., the presence of defined “zooming” scales, such as “system – container – component” (as in the C4 model [6]), but this division has a purely spatial/structural character);

– 4 – cross-cutting multi-vector hierarchical model (e.g., simultaneous application of the spatial vector of “system – subsystem – component” scales and the conceptual vector “from abstract to concrete”, which are integrated into the stages of the system design method at different levels of the model of CSS elements and levels [23]).

Adaptability to the heterogeneous nature of systems (AHS) determines the ability of the method to take into account fundamental differences in CSS types:

– 1 – rigid binding to one class of systems (purely IT systems or purely PLC programs);

– 2 – universality (agnostic), requiring manual unformalized adaptation;

– 3 – specialized adaptation for a separate domain;

– 4 – embedded fundamental adaptability (differentiation of architectural viewpoints depending on the system type).

Interoperability of artifacts and readiness for automation (IRA) evaluates the structuredness of artifacts for their machine processing (LLM or AI agents [29]):

– 1 – implicit links and non-standardized artifacts (purely a model of artifact description);

– 2 – standardized artifacts but implicit links;

– 3 – standardized artifacts and links between them, but absent component of design process staging;

– 4 – defined stages of the design process and inputs/outputs of the sequence of design stages, forming a graph of dependencies between standardized artifacts.

The evaluation results are presented in Table 12.

Table 12

Quantitative comparison of architectural methods by criteria of effectiveness and AI-readiness

Method / framework / approach	PPI	CAD	AHS	IRA	Total score
System-type-centric architectural design method	4	4	4	4	16
IoT-A reference model [18]	3	3	3	3	12
SAF (System Arch. Framework) [14]	3	4	2	3	12
SEI (ADD, views and beyond) [7, 12]	3	3	2	3	11
FAS (Functional Arch. for Systems) [13]	3	2	2	3	10
C4 model [6]	1	3	2	3	9
arc42 template [5]	1	3	2	2	8
ISO/IEC/IEEE 42010	1	2	2	2	7

As shown by the results of the quantitative assessment (Table 12), existing standards and approaches serve as a reliable basis for describing architecture. However, in the context of the design of heterogeneous CSS in automation, they require significant efforts for manual adaptation. This is reflected in the AHS index values, which do not exceed 2–3 points.

In particular, the C4 model [6] and arc42 template [5] provide structured representation of architecture (CAD = 3) but do not contain a built-in design algorithm (PPI = 1), which limits their use as full-fledged methods of architectural synthesis.

In turn, the SEI (ADD, views and beyond) [7, 12] methodologies, the SAF [14] framework, and the IoT-A reference model [18] provide a higher level of process support (PPI = 3). However, despite extended decomposition (in particular, CAD = 4 for SAF) or partial domain specialization (AHS = 3 for IoT-A), they remain only partially formalized. None of these approaches has built-in mechanisms of fundamental type-oriented adaptation to different architectural natures of systems (AHS < 4), which reduces the degree of their interoperability and readiness for full automation (IRA <= 3).

The proposed system-type-centric method demonstrates maximum values for all criteria (16 points) due to the combination of several key properties. First, the method formalizes a reproducible and deterministic architectural design process (PPI = 4). Second, it defines a complete hierarchical

model of architectural decomposition that covers both software and physical aspects of systems (CAD = 4). Third, the built-in system-type-oriented adaptation (AHS = 4) ensures the selection of relevant architectural viewpoints depending on the fundamental nature of the system. The combination of these characteristics, supplemented by strict rules for defining inputs and outputs for each architectural viewpoint, ensures a high level of artifact interoperability (IRA = 4). This allows considering the method not only as a design tool but also as a formalized basis for building automated architecture generation systems based on artificial intelligence.

Thus, the proposed method forms a formalized basis for architectural design with an increased level of reproducibility, completeness of decomposition, and structuredness of artifacts, which creates the prerequisites for its application in tasks of automated architectural design. The proposed method ensures an increase in the integral effectiveness of architectural design compared to existing approaches, due to a higher level of formalization, adaptability, and completeness of decomposition. The obtained results follow directly from the structure of the method and the adopted system of criteria, and are not based on empirical assumptions, which ensures their reproducibility.

5. 6. Example of application of the design method

An example of application of the design method is demonstrated on an industrial gas pumping automation system at the level of a compressor station, described in [23]. The system consists of the following subsystems:

– gas pumping unit (GPU) control subsystem based on programmable logic controllers (PLCs), located at the lower levels of the automation hierarchy; the PLC program directly implements the control logic of the compressor shop unit. An example of a "Continuous System", specifically a "Physical Control System" (the type is indicated in orange in Fig. 4);

– SCADA subsystem of the compressor shop (local SCADA). An example of a platform-like subsystem, implemented on the basis of "Self-hosted off-the-shelf" software (SCADA WinCC) at the application level (the type is indicated in grey in Fig. 4);

– SCADA subsystem of the compressor station (global SCADA), which consolidates data from local SCADAs and provides a dispatch interface at the station level. It is also a platform-like subsystem based on "Self-hosted off-the-shelf" software (the type is indicated in grey in Fig. 4).

The internal structure of an off-the-shelf product is not considered in the architectural design of the CSS. However, the platform-like nature of both SCADA subsystems requires the definition of a limited set of representations for project-specific extensions (tags, message processing scripts, integration APIs);

– Data analytics processing and condition monitoring subsystem. An example of a "Data-flow-oriented System" (the type is indicated in green in Fig. 4);

– Autonomous high-level control subsystem. An example of a system that is a hybrid of two subtypes of "Behavior-oriented System" (the type is indicated in blue in Fig. 4): "Computation-oriented System" and "Domain-data-model-oriented System";

– Air condition monitoring subsystem in the shop. An example of a "Physical Control System" (the type is indicated in orange in Fig. 4).

At the System scale, the following architectural representations are applied (Table 11):

– functional requirements viewpoint (I-1) and NFR and constraints viewpoint (I-2) – for specifying the problem space, in the form of a catalog of functional and non-functional requirements and constraints, including real-time requirements for GPU control loops, telemetry latency limits, and SCADA availability requirements;

– capabilities viewpoint (SA-1) – for defining the hierarchy of business-oriented and derived technical system capabilities (GPU control, supervisory control at the shop level, supervisory control at the station level, analytics and equipment condition monitoring, autonomous decision-making, air monitoring) based on the input specifications I-1 and I-2;

– subsystems viewpoint (SC-1) – for decomposing the system into subsystems, classifying each of them (custom / off-the-shelf / external system integration), and assigning a system type [23] to each custom subsystem. The same representation also displays an external integration with the SaaS service PagerDuty for notifying technical personnel of anomalies, which is outside the system. Color coding is used for visual differentiation of system types according to SC-1 recommendations. Fig. 4 shows the representation of subsystems of the industrial automation system [23];

– UX viewpoint (XA-1), applied within the system, for describing end-to-end user scenarios of the shop operator and station dispatcher, covering the UX of local and global SCADA, as well as the consoles of the autonomous control subsystem.

For each of the subsystems, relevant architectural viewpoints have been determined based on the type of subsystem. The architectural viewpoints and selected representation formats for each of the subsystems at the Subsystem scale are described below.

For the GPU control subsystem based on PLC (continuous control), the following set of representations has been formed:

– control viewpoint (SSA-1) – in the format of a control block diagram with a mathematical model of GPU control;

– high-level components viewpoint (SSA-3) and its extension with component classification (SSA-3-ext) – for logical decomposition of the subsystem and designation of the PLC program as a custom application-level component;

– software components viewpoint (SSC-1) – representation of software components at the application and infrastructure levels – in this case, the PLC program as a software component at the application level;

– infrastructure viewpoint (XC-2) – for describing the physical controller, industrial network, etc.;

– deployment viewpoint (XC-3) – for describing the strategy of deployment and updating of the PLC program on hardware resources in the production environment.

For the air condition monitoring subsystem in the shop (Continuous Control, embedded):

– control viewpoint (SSA-1) – mathematical models of processing sensor signals and calculating integral air quality indicators;

– SSA-3 and SSA-3-ext – for logical decomposition of the embedded system;

– XC-2 – for describing the embedded hardware infrastructure, in particular the design of a printed circuit board with an ESP32 microcontroller;

– SSC-1 – for representing physical software components (firmware);

– XC-3 – for describing the firmware version update process in the production environment (e.g., the OTA (Over-the-air) update process).

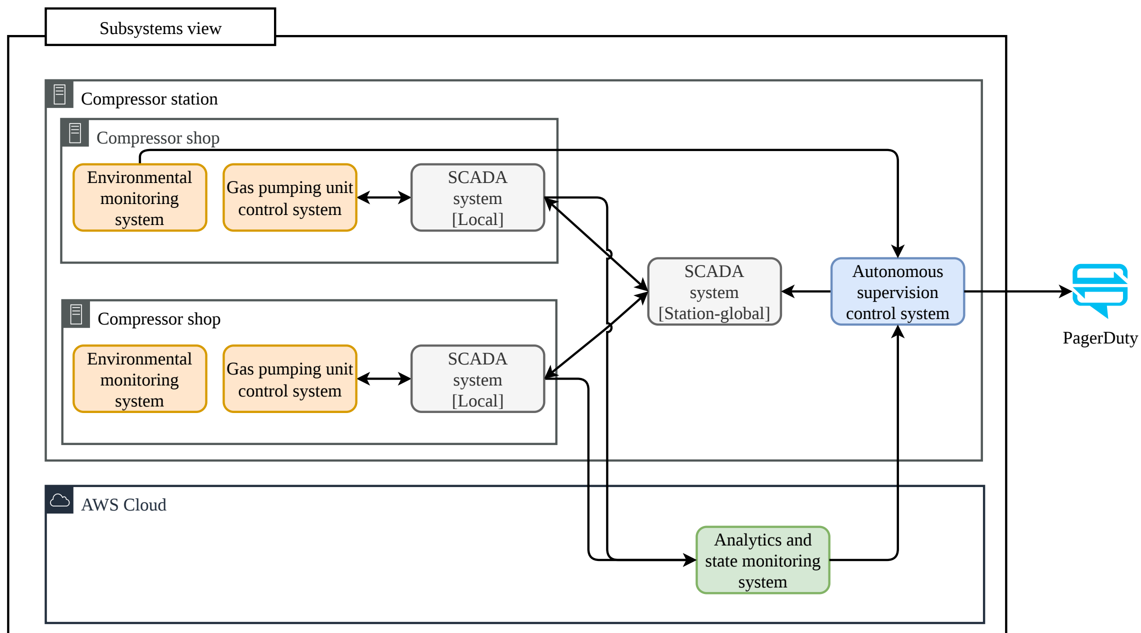

Fig. 4. Example of representation of subsystems of an industrial automation system

For the data analytics processing and condition monitoring subsystem (Data-flow-oriented, cloud-based):

– SSA-3 and SSA-3-ext in the format of a data flow diagram – for logical representation of the telemetry processing flow from the point of reception via MQTT to storing aggregated data and invoking the ML model;

– SSC-1 and XC-2 – for representing cloud components based on AWS (IoT Core, Kinesis Data Streams, Kinesis Data Firehose, Lambda, S3, SageMaker, Athena, QuickSight).

For the autonomous high-level control subsystem (behavior-oriented, hybrid of computation-oriented and domain-data-model-oriented):

– SSA-3 and SSA-3-ext – for logical decomposition of the subsystem into abstract components at the application and infrastructure levels;

– SSC-1 – represents software components, including a server software component (Java; application level) and a self-hosted Postgres server (infrastructure level), including software-relevant aspects of the runtime environment;

– XC-2 – for describing the on-premises hardware infrastructure of the subsystem and the virtualization environment.

For the local SCADA subsystem of the compressor shop (off-the-shelf self-hosted, platform-like), full internal architectural design is not performed. According to the architectural method, a limited set of representations is applied for off-the-shelf self-hosted subsystems:

– XA-1 – for describing UX flows of the shop operator based on the local SCADA;

– SSC-1 – for describing which off-the-shelf components form the subsystem (SCADA WinCC server and MS SQL server DBMS);

– XC-2 – for describing the hardware-platform environment of the SCADA server (on-premises server of the compressor shop without virtualization);

– XC-3 – for describing the strategy of deploying SCADA servers on the shop server.

For the global SCADA subsystem of the compressor station (off-the-shelf self-hosted, platform-like), a similar set of representations is applied with emphasis on the station scale:

– XA-1 – for describing UX flows of the station dispatcher in the dispatch screens of the global SCADA;

– SSC-1 – for displaying off-the-shelf components that form the subsystem (SCADA WinCC server and MS SQL server);

– XC-2 – for describing the hardware-platform environment of the SCADA server of the station, including the organization of virtualization on the compressor station server;

– XC-3 – for describing the strategy of delivery and deployment of SCADA server software in the virtualization environment, e.g., the deployment and update process via an engineering station based on Siemens TIA Portal (Siemens AG, Germany) for generation and updating of the SCADA runtime configuration.

Fig. 5 shows an example of a consolidated representation for the SSC-1 viewpoint – the representation of physical software components.

For each of the custom application-level software components of each of the subsystems, as well as for platform-like off-the-shelf components, relevant architectural representations of the component scale have been determined.

For the custom component "GPU control PLC program" (PLC program; FBD) within the GPU control subsystem (continuous control):

– execution viewpoint (CC-1) – for describing the cyclic execution of PLC organization blocks and task distribution in the PLC runtime environment;

– states / stages viewpoint (CC-2) – in sequential function chart (SFC) format for describing discrete operating modes of the GPU (stopped, startup, normal operation, emergency shutdown) and rules of transitions between them; each mode corresponds to its own set of control rules, traced to SSA-1;

– domain data model viewpoint (CC-6) – in PLC tags catalog format;

– I/O signal interfaces and pins viewpoint (CC-8) – for displaying the mapping of PLC inputs/outputs to physical signals of the GPU (pressure measurements, temperatures, control of actuators).

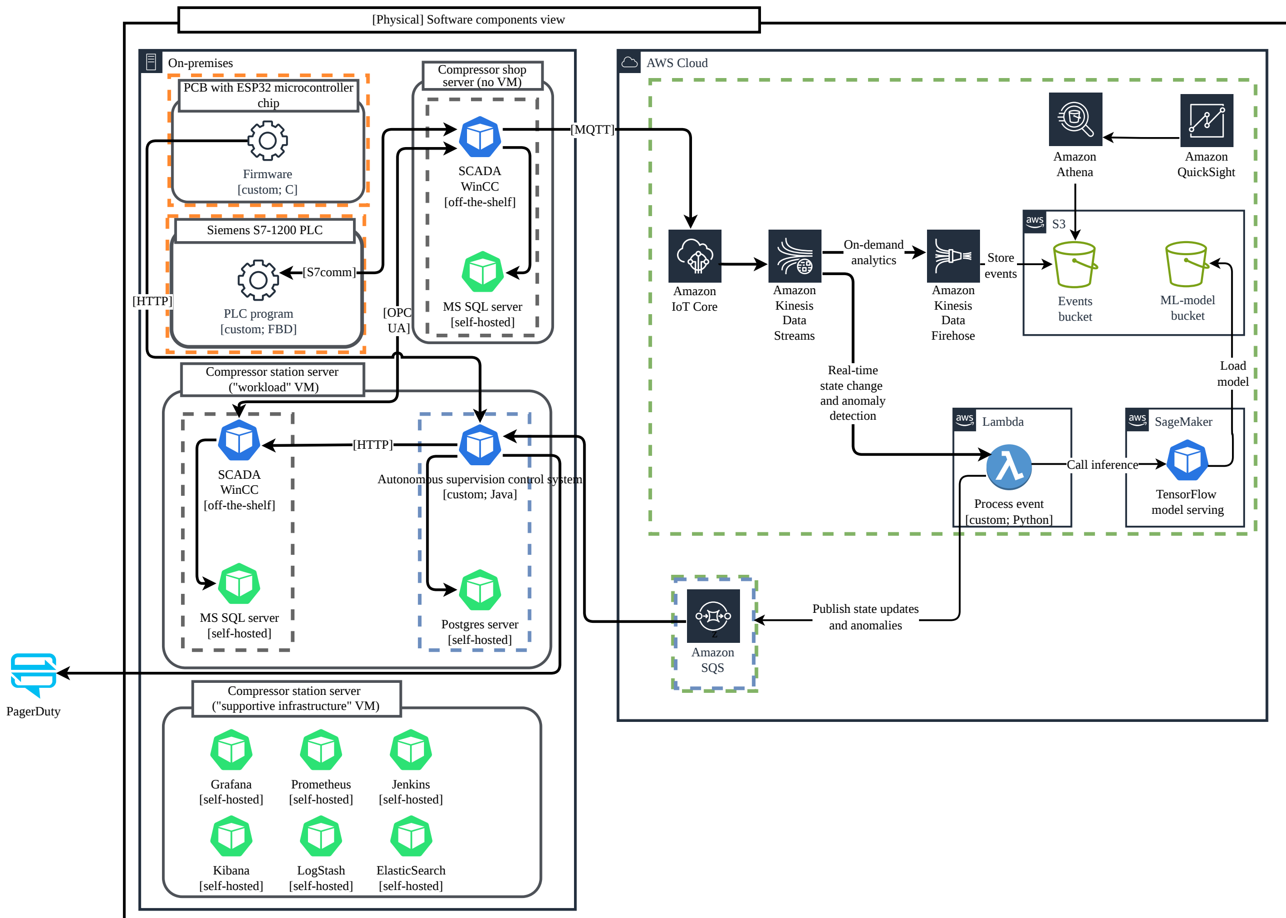

Fig. 5. Example of representation of physical software components

For the custom component “Air monitoring system firmware” (Firmware; C) within the air condition monitoring subsystem in the shop (continuous control, embedded):

– CC-1 – for describing RTOS tasks (sensor reading, processing, transmission);

– CC-2 – for the finite-state machine of device operating modes;

– CC-3 – for representing the modular structure of the program in the format of a C4 component diagram;

– CC-5 – for describing integration with the autonomous high-level control subsystem;

– CC-6 – for the internal domain model of air telemetry;

– CC-8 – for describing the pins of the ESP32 microcontroller and their purpose (ADC, digital inputs, UART/I2C interfaces for sensors).

For the custom component “Analytics event handler” (Process event; Python on AWS Lambda) within the data analytics processing and condition monitoring subsystem (Data-flow-oriented):

– CA-1 and CA-1-ext – for the catalog of processings indicating MQTT/Kinesis as the access method;

– contracts / APIs / actions viewpoint (CC-4) – for the specification of the structure of input and output events (event schema) in AsyncAPI format;

– integrations APIs viewpoint (CC-5) – for describing integrations with SageMaker (calling inference of the TensorFlow model);

– persistent data model viewpoint (CC-7) – for describing the structure of aggregated data in S3 buckets and Athena tables;

– calculation viewpoint (D-2) – for the mathematical model of anomaly detection;

– task / transformation viewpoint (D-3) – for the rules of normalization, filtering, and enrichment of input telemetry.

For the custom component “Autonomous supervision control system” (Java) within the autonomous high-level control subsystem (Behavior-oriented, hybrid of computation-oriented and domain-data-model-oriented):

– CA-1 and CA-1-ext – for the catalog of behaviors indicating API types (REST API, MQTT-based API, etc.) and scheduled triggers;

– behavioral dynamic viewpoint (CA-2) – in sequence diagram format for cross-behavioral scenarios (e.g., “anomaly detected – check GPU status via SCADA – decision making – control command – escalation to PagerDuty”);

– component structural viewpoint (CC-3) – in C4 component diagram format for the modular structure of the Java application;

– CC-1 – for describing concurrent tasks (MQTT consumers, request handlers, background tasks);

– CC-4 – for the specification of its own REST API in OpenAPI format;

– CC-5 – for describing integrations with local and global SCADA WinCC (via HTTP), with the analytics subsystem (via AWS SQS), and with external PagerDuty (via HTTP);

– domain data model viewpoint (CC-6) – in class diagram format for the domain model;

– persistent data model viewpoint (CC-7) – in ERD format for the Postgres schema;

– procedure viewpoint (D-1) – for detailing the logic of key decision-making behaviors;

– calculation viewpoint (D-2) – for describing mathematical models of decision-making.

For the platform-like off-the-shelf components "SCADA WinCC" of the local and global SCADA subsystems:

– CA-1-ext and CC-4 – for documenting API extensions (e.g., custom WinCC Unified MQTT message processing scripts) on each of the SCADA subsystems;

– CC-5 – for describing integrations of local SCADA with the GPU control subsystem (via S7comm), as well as global SCADA with local SCADAs (via OPC UA);

– CC-6 – for the specification of the project-specific tag hierarchy of each of the SCADA subsystems (tags of local SCADA cover shop tags; tags of global SCADA – consolidated station tag model);

– CC-7 – not applicable to SCADA systems, as the schema of their databases (MS SQL) is not subject to extensions or changes.

For each of the applied architectural viewpoints, a governance document is additionally defined that describes the project-specific standardization of representations: mandatory attributes, naming conventions, mandatory information in models, recommended modeling tools, and rules for tracing between representations of different viewpoints. For example, the governance document for CC-6 in the context of the PLC program reflects the tag naming convention and mandatory tracing of signals in CC-8 to tags, while the governance document for CC-4 reflects project-general conventions for building APIs of software components.

The given set of architectural representations, defined within the architectural viewpoints proposed in this method for the architectural design of CSS of various types [23], demonstrates the application of the method to a typical industrial automation system. Similarly, the method can be applied to complex heterogeneous CSS in automation. Including for the architectural design of IT systems in automation, the design of robotic systems, and the design of industrial automation systems.

6. Discussion of the results of the development of the system-type-centric method for architectural design of computer-software automation systems

This study defines a formalized system-type-centric method for the architectural design and documentation of computer-software automation systems. By synthesizing the theoretical concepts of the system-type-centric paradigm [23] with widely accepted standards (ISO/IEC/IEEE 42010) and viewpoint-oriented approaches [7, 8] in the industry, the study addresses the fragmentation and over-abstraction prevalent in existing design methodologies.

The obtained result is explained by the definition of sets of architectural viewpoints corresponding to the specifics of the fundamental architectural nature of different CSS types [23] (Table 11). The set of architectural viewpoints corresponds to the interests of stakeholders (Table 1) defined at different levels (Fig. 1, 2) and is formed in the form of an architectural documentation model (Tables 2–10). The architectural viewpoints in the applicability matrix (Table 11) are organized according to the level structure, as well as in the transition from abstract to concrete design (Fig. 1). The structure of the applicability matrix defines a clear order for constructing architectural artifacts in accordance with the defined viewpoints. The construction of artifacts at lower levels is directly based on artifacts constructed at higher levels. This forms the architectural design process of CSS. Thus, the method provides prescriptive and sufficiently detailed tools for architectural design. At the same time, the method is applicable to a wide range of CSS precisely due to adaptation to the specifics of the architectural nature of CSS [23].

The proposed method combines prescriptiveness and detail with applicability to a wide range of CSS and completeness of the model, which allows building the architecture of the system at all levels. This is its main difference compared to existing standards (ISO/IEC/IEEE 42010) and approaches [7, 8], which do not propose a prescriptive process. It also distinguishes it from previously proposed architectural design methods [19, 20], which have a siloed nature and focus on simplified transitions.

The main contributions of this study include:

– the establishment of a comprehensive architectural design model defined as a set of architectural viewpoints structurally categorized across system, subsystem, and component scales. This model outlines a systematic progression from abstract capabilities and logical structures to concrete, physical aspects of the system at the software level and the infrastructure level;

– the mapping of specific viewpoint applicabilities to fundamentally distinct system types [23] (behavior-oriented, data-flow-oriented, continuous agentic, and continuous control). This ensures that the unique architectural aspects of systems with different fundamental architectural nature are addressed;

– the definition of a reproducible architectural design process that prescribes a uniform approach to computer-software systems architectural design. The process starts from inputs (functional requirements, NFRs, constraints) and results in the designed and documented architecture expressed via architectural documentation with interoperable artifacts. The inclusion of system-specific governance documents alongside architectural views ensures system-specific tailoring capabilities built into the method, resulting in improvements to long-term maintainability and architectural compliance.

Ultimately, the proposed method provides engineering teams with a versatile yet highly prescriptive toolset. It minimizes structural ambiguity during the design of complex automation systems and fosters interoperability between architectural design artifacts of disparate system components.

The limitations inherent in this study relate to the boundaries and conditions of application of the proposed solutions: the method is intended for use in ranges of input data where the initial requirements (in particular NFRs and constraints) are sufficiently formalized and stable. That is, the boundaries of application of the method relate only to direct architectural design. Application of the method in enterprise architectures, where domain knowledge discovery, capability definition, and requirements formation are separate and fundamental processes, requires integration of this method with appropriate methodologies and approaches.

Among the disadvantages of this study, the high labor intensity of manual creation and maintenance of the complete architectural documentation package according to the proposed formalized method should be noted. In the future, this disadvantage can be eliminated by integrating this method into automated design software systems.

The formalization of the system-type-centric architectural design method opens several highly promising directions

for future study and practical application within computer-software systems engineering:

– integration with computer-aided design (CAD) and computer-aided software engineering (CASE) tools: one of the main areas of the study includes embedding the proposed method into CAD and CASE tools. The development of specialized modeling profiles based on this method would provide a semi-automated workflow for architects, directly ensuring the production of interoperable and compliant artifacts. This will require additional efforts to standardize model documentation formats for each of the architectural viewpoints, in accordance with the internal formats and standards of CAD or CASE systems;

– method extensibility and evolution: another critical area of the study is the formalization of rules for introducing extension points (additional, domain-specific viewpoints). Future studies may define mathematical or logical constraints that allow the method to evolve alongside emerging technologies without compromising its core properties of artifact interoperability and consistency. Related challenges will include the proper integration of introduced architectural viewpoints and extensions into the structure of existing viewpoints, preserving dependencies and traceability;

– AI-driven architecture generation: the structured nature of the proposed method may serve as a direct instruction set for AI-driven automated architecture generation systems. Future studies may explore how large language models (LLMs) and intelligent agents can leverage these standardized viewpoints to act as architectural co-pilots. This could enable fully automated pipelines capable of translating high-level business requirements into interoperable architectural descriptions, and subsequently, into executable system implementations and automated verification suites. This direction is highly relevant at the present time, however the proposed approaches [29] focus mainly on the aspects of artificial intelligence development and the principles of building agentic systems. They do not pay attention to a standardized set of instructions, which is fundamentally important, and the basis for which the proposed method could form.

7. Conclusions

1. The conceptual foundations and structural principles of a system-type-centric architectural design method have been formalized, establishing a coherent methodological basis for architectural design that integrates viewpoint-oriented documentation, hierarchical architectural scales, and a structured design progression.

2. A comprehensive and standardized set of architectural viewpoints has been defined, spanning multiple architectural scales and specifying their intended scope, inputs, dependencies, and representation forms, thereby forming a complete and consistent architectural documentation model.

3. A prescriptive architectural design process has been established by defining the ordered application of architectural viewpoints and mapping their applicability across fundamentally different system types, enabling systematic and reproducible architectural design for heterogeneous automation systems.

4. Formal rules governing the architectural design process have been defined at the lifecycle level, ensuring architectural consistency, decision traceability, and controlled evolution of architectural solutions over time.

5. A quantitative assessment of the integral effectiveness of the proposed method has been carried out by formalizing qualitative characteristics into measurable indices and conducting a comparative analysis against generally accepted industry standards, methodologies, and frameworks of architectural design. This demonstrated the effectiveness of the method according to the criteria of process prescriptiveness, completeness of architectural decomposition, adaptability to the heterogeneous nature of systems, and interoperability of artifacts and readiness for automated application.

6. The applicability of the proposed method has been demonstrated on the example of architectural design of a heterogeneous computer-software industrial gas pumping automation system at the level of a compressor station, which confirmed its practical suitability for designing complex automation systems combining subsystems of different architectural types.

Conflict of interest

The authors declare that they have no conflict of interest in relation to this study, whether financial, personal, authorship or otherwise, that could affect the study and its results presented in this paper.

Financing

The study was performed without financial support.

Data availability

Manuscript has no associated data.

Use of artificial intelligence

The authors confirm that they did not use artificial intelligence technologies in creating the submitted paper.

Authors' contributions

Ihor Polataiko: Conceptualization; Investigation; Methodology; Writing – Original Draft; Writing – review & editing; **Leonid Zamikhovskyi:** Supervision; Project administration; Writing – review & editing.

References

1. Zamikhovskyi, L., Nykolaychuk, M., Levytskyi, I. (2025). Extending the functionality of topologies of Web-oriented control systems for technological objects based on "Open User Communication". Eastern-European Journal of Enterprise Technologies, 6 (2 (138)), 94–115. https://doi.org/10.15587/1729-4061.2025.348728
2. Zamikhovskyi, L., Zamikhovska, O., Ivanyuk, N., Mirzoieva, O., Nykolaychuk, M. (2025). Development of an anti-surge protection system for gas pumping units based on hardware and software vibration monitoring tools. Eastern-European Journal of Enterprise Technologies, 4 (2 (136)), 117–132. https://doi.org/10.15587/1729-4061.2025.337736

3. Zamikhovskyi, L., Nykolaychuk, M., Levytskyi, I. (2025). Development of a simulation model of a WEB-oriented servo drive frequency control system based on "Digital Twins" technology. Technology Audit and Production Reserves, 6 (2 (86)), 76–90. https://doi.org/10.15587/2706-5448.2025.345825
4. Zamikhovskyi, L., Nykolaychuk, M., Levytskyi, I. (2024). Organizing the automated system of dispatch control over pump units at water pumping stations. Eastern-European Journal of Enterprise Technologies, 5 (2 (131)), 61–75. https://doi.org/10.15587/1729-4061.2024.313531
5. Starke, G., Simons, M., Zörner, S., Müller, R. D. (2019). arc42 by Example: Software architecture documentation in practice. Birmingham: Packt Publishing. Available at: https://www.packtpub.com/en-us/product/arc42-by-example-9781839219269
6. Brown, S. (2015). The C4 model for visualising software architecture. Leanpub. Available at: https://leanpub.com/visualising-software-architecture
7. Bass, L., Clements, P., Kazman, R. (2021). Software Architecture in Practice. Addison-Wesley Professional. Available at: https://www.oreilly.com/library/view/software-architecture-in/9780136885979
8. Rozanski, N., Woods, E. (2011). Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives. Boston: Addison-Wesley. Available at: https://www.oreilly.com/library/view/software-systems-architecture/9780132906135
9. Kruchten, P. B. (1995). The 4+1 View Model of architecture. IEEE Software, 12 (6), 42–50. https://doi.org/10.1109/52.469759
10. Kruchten, P. (2003). Rational Unified Process, The: An Introduction. Addison-Wesley Professional. Available at: https://www.oreilly.com/library/view/rational-unified-process/0321197704/ch17.html
11. Rational Unified Process for Systems Engineering. Available at: https://public.dhe.ibm.com/software/rational/web/whitepapers/2003/TP165.pdf
12. Cervantes, H., Kazman, R. (2016). Designing Software Architectures: A Practical Approach. Boston: Addison-Wesley Professional. Available at: https://www.oreilly.com/library/view/designing-software-architectures/9780134390857
13. Lamm, J. G., Weilkiens, T. (2013). Method for Deriving Functional Architectures from Use Cases. Systems Engineering, 17 (2), 225–236. https://doi.org/10.1002/sys.21265
14. System Architecture Framework. Available at: https://saf.gfse.org/
15. Dömel, A. (2025). The Robot Architecture Framework - A systematic approach for describing the architecture of complex, autonomous robotic systems. University of Bremen. https://doi.org/10.26092/elib/3825
16. Systems Engineering Handbook (NASA/SP-2016-6105 Rev2) (2017). National Aeronautics and Space Administration. Available at: https://ntrs.nasa.gov/citations/20170001761
17. Weilkiens, T., Lamm, J. G., Roth, S., Walker, M. (2022). Model-Based System Architecture. Wiley. https://doi.org/10.1002/9781119746683
18. Internet of Things Architecture. Available at: https://cordis.europa.eu/project/id/257521/results
19. Hatebur, D., Heisel, M. (2009). Deriving Software Architectures from Problem Descriptions. Software Engineering 2009 - Workshopband. Available at: https://dl.gi.de/items/c3d3e64d-c704-4891-b7ad-9a751caad44e
20. Alebrahim, A., Hatebur, D., Heisel, M. (2011). A Method to Derive Software Architectures from Quality Requirements. 2011 18th Asia-Pacific Software Engineering Conference, 322–330. https://doi.org/10.1109/apsec.2011.29
21. Reddy, A. R. M., Govindarajulu, P., Naidu, M. M. (2007). A Process Model for Software Architecture. IJCSNS International Journal of Computer Science and Network Security, 7 (4), 272–280. Available at: http://paper.ijcsns.org/07_book/200704/20070439.pdf
22. Hofmeister, C., Kruchten, P., Nord, R. L., Obbink, H., Ran, A., America, P. (2007). A general model of software architecture design derived from five industrial approaches. Journal of Systems and Software, 80 (1), 106–126. https://doi.org/10.1016/j.jss.2006.05.024
23. Polataiko, I., Zamikhovskyi, L. (2026). Development of system-type-centric paradigm of computer-software systems architectural design for automation systems. Technology Audit and Production Reserves, 1 (2 (87)), 43–56. https://doi.org/10.15587/2706-5448.2026.349943
24. Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley Professional. Available at: https://www.oreilly.com/library/view/domain-driven-design-tackling/0321125215
25. Vernon, V. (2013). Implementing Domain-Driven Design. Addison-Wesley Professional. Oreilly. Available at: https://www.oreilly.com/library/view/implementing-domain-driven-design/9780133039900
26. da Silva, V. T., Noya, R. C., de Lucena, C. J. P. (2005). Using the UML 2.0 activity diagram to model agent plans and actions. Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, 594–600. https://doi.org/10.1145/1082473.1082563
27. MQTT Communication with WinCC Unified (2026). Siemens. Available at: https://support.industry.siemens.com/cs/attachments/110000706/110000706_MQTT_WinCC_Unified_DOC_V1_0_en.pdf
28. Ford, N., Parsons, R., Kua, P. (2022). Building Evolutionary Architectures. O'Reilly Media. Available at: https://www.oreilly.com/library/view/building-evolutionary-architectures/9781492097532
29. Zhang, Y., Li, R., Liang, P., Sun, W., Liu, Y. (2025). Knowledge-Based Multi-Agent Framework for Automated Software Architecture Design. Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, 530–534. https://doi.org/10.1145/3696630.3728493