UDC 004.03
DOI: 10.15587/2706-5448.2026.349943

Ihor Polataiko,
Leonid Zamikhovskyi

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

The object of research is the design processes of computer-software systems in the automation domain. Computer-software systems are considered as a combination of software and hardware.

This research focuses on addressing the lack of a formal methodology that differentiates the architectural design of computer-software systems for automation systems, including in hybrid Information Technology/Operational Technology environments.

Existing approaches, methods, and models for designing the architecture of computer-software systems do not account for the differentiation of approaches based on the fundamental differences in the nature of various system types used in the automation domain. This precludes the establishment of standardized and differentiated processes and methods for designing the architecture of such systems.

The main result is a new type-centric paradigm of architectural design, based on a taxonomy that classifies computer-software systems into fundamental types by their architectural nature. Furthermore, within the scope of the research, the levels and elements of computer-software systems are defined, which forms a standardized vision of the systems' constituent parts.

In contrast to existing models, approaches, and methodologies, the proposed type-centric paradigm for the architectural design of computer-software systems incorporates the fundamental aspects of the systems' nature that are important for architectural design, classifying systems based on this factor.

The proposed paradigm provides a foundation for systematized, specialized, and differentiated architectural design processes adapted to the specifics of the systems' nature. This ensures standardization, interoperability, and systematicity in the design of computer-software systems for automation systems, potentially forming a basis for automated architectural design systems.

Keywords: *software architecture, automation systems, system taxonomy, computer systems, architectural design.*

Received: 10.11.2025
Received in revised form: 28.12.2025
Accepted: 12.01.2026
Published: 28.02.2026

How to cite
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

1. Introduction

To solve the problems of modern scale, including the implementation of the concepts of Industry 4.0 and Industry 5.0, automation systems are increasingly aimed at a hybrid of classical industrial control systems and information systems. Such systems combine automated process control systems (APCS) based on feedback loops (operational technology), at lower levels, and information systems (information technology), at higher levels. Such a structure of automation systems is described in classical approaches, such as the reference model of computer-integrated manufacturing "automation pyramid" Purdue [1] or the ISA-95 standard [2]. The structure is also given in modern foundational documents, for example, the "Industrial Internet Reference Architecture" (IIRA) [3], describes functional domains that are largely based on information technologies. Among them: "information domain", "system management domain", "business domain" and "application domain". While the "control and monitoring domain" defines the systems to which control systems with Internet of Things capabilities belong.

In addition, the implementation of Internet of Things (IoT) technologies, including the Industrial Internet of Things (IIoT), is moving classic computer control systems closer to IT systems due to the expansion of communication and data processing capabilities at the edge [3]. At the same time, IT systems such as Kubernetes [4], which is widely used for orchestration of application containers, as well as simulations and computer games, internally use elements of control systems, in particular feedback-based control.

The process of architectural design of computer software systems (CSS) is a cornerstone of the software development life cycle of automation systems. However, this process affects not only the design phase; it is present at all stages of development. It lays the foundations of the system's ability to fulfill its goals (defined by the functional requirements for the system) in an effective and appropriate manner (defined by non-functional requirements and constraints).

Computer software system architecture (CSA) in automation requires both initial design for new systems (greenfield systems) and constant updating during the evolution of the system. Such evolution is due to the expansion of the list of system requirements due to the change of business needs and vision.

Many classic software architectural design models and architectural design templates are well-known and accepted in the field of software engineering. Among them are the classic "4+1" model [5], arc42 architectural templates [6, 7] and others. Architecture description standards [8]

ISSN-L 2664-9969; E-ISSN 2706-5448

and architectural process [9] are also defined in the discipline of software engineering. These standards serve as inspiration for the methods and approaches described in influential works that shape modern paradigms of software system architectural design. Among them, practical methods for designing and documenting system architecture based on architectural views [10], attribute-driven design (ADD) [11], and the "viewpoints and perspectives" approach [12].

In the field of automation, numerous standards and reference models are widely known and used to describe structures and hierarchies. These include reference models for computer-integrated manufacturing, such as the Purdue "automation pyramid" [1] and the ISA-95 standard [2]. Also widely known are reference models for the Internet of Things, such as IIRA [3] for the industrial case, and IoT-A [13] and ASSIST-IoT [14, 15] for the general case. Reference architecture models for Industry 4.0 (RAMI) [16] is a popular model for the architecture of industrial systems in Industry 4.0. There are also standards for control architectures for specific cases, such as batch production control [17], or specific industries, such as automotive [18]. Approaches and methodologies have been developed in the field of information systems and systems engineering to solve high-level design problems. For example, Domain-Driven Design (DDD) [19] is popular in IT systems for domain analysis and system design, and Functional Architecture for Systems (FAS) [20, 21] is a widely used system engineering method for cyber-physical systems [22].

Enterprise and system-level design standards and methods are described in many documents and publications on systems engineering and enterprise architecture. Among them, the NIST Cyber-Physical Systems Framework [23] and the NASA Systems Engineering Handbook [24]. Also, authoritative standards for the development and representation of enterprise-level architectures, such as DoDAF [25], MODAF [26], UAF [27], TRAK [28], NAF [29], and TOGAF [30], are well-defined. Although these approaches are excellent at defining system hierarchies, interoperability levels, and enterprise integration, they are mostly focused on the definition and interaction of systems. These standards do not define the architectural design processes for computer-software systems, taking into account the differences in approaches to designing systems of different nature.

In addition, architectural design frameworks have been proposed for general [31] and specialized cases, such as robotic systems [32]. There are also approaches based on Model-Based Systems Engineering (MBSE), such as [21], which propose a modeling-based system architecture process. However, these approaches are general in that they provide a universal design canvas. They do not contain specific recommendations for adapting to the specific architectural aspects of the fundamentally different types of systems present in today's heterogeneous large-scale automation systems.

Research and analytical reports on system design methodologies have also become publicly available over the past decades. For example, studies [33, 34] focus on methods for deriving software system architectures from system requirements. The study [35] focuses on the formal foundations and methods of designing cyber-physical systems. In [36], a model of the software architecture design process is proposed, introducing the concept of the software architecture development life cycle. In [37], a basic simplified process for designing automated systems is described. The adaptation of the "Rational Unified Process" methodology to the domain and tasks of systems engineering is given in [38].

In addition, an analysis of scientific works on taxonomies in the field of software systems was conducted. In particular, studies of anti-patterns in software classification taxonomies were analyzed [39] and examples of taxonomies in software engineering aimed at introducing additional dimensions of classification [40]. Also, systematic reviews focused on the landscape of existing taxonomies in software systems were analyzed [41].

The search for source materials was conducted in leading scientometric databases and digital libraries specializing in computer science and engineering, including IEEE Xplore, ACM Digital Library, Scopus, Web of Science, and Google Scholar.

The reviewed materials cover four key tangents, disciplines that are critical to this research:

1. *Industrial Automation and Reference Models in Industry (Operational Technology/Computer-Integrated Manufacturing):* Includes foundational models such as the Purdue Reference Model (PERA) [1] and its formalization in IEC 62264 (ISA-95) [2]. Also covers modern reference architectures for Industry 4.0 and IIoT, including Industrial Internet Reference Architecture (IIRA) [3], IoT-A [13], ASSIST-IoT [14], Reference architecture model for Industry 4.0 (RAMI) [16], and industry-specific standards such as ISA-88 [17] and AUTOSAR [18].
2. *Software Architecture (Information Technology):* Covers international standards for describing architecture (ISO/IEC/IEEE 42010 [8]) and architectural processes (ISO/IEC/IEEE 42020 [9]). Includes leading methodological works and models, such as the 4+1 View Model [5] and Domain-Driven Design [19], as well as foundational texts on software architecture practice [10–12] and standard templates for describing system architectures [6, 7].
3. *Systems Engineering and Enterprise Architecture* includes high-level frameworks used for designing complex systems and enterprise architectures. These include TOGAF [30], NAF [29], DoDAF [25], MODAF [26], UAF [27], SAF [31], as well as NASA's systems engineering guides [24] and the NIST CPS Framework [23].
4. *Academic research on methodologies and taxonomies:* includes research on the development of architectures based on requirements and use cases [20, 33, 34], the design of cyber-physical systems (CPS) [22, 35], and the development of taxonomies in software engineering [39–41].

Software engineering approaches (e. g., [8–12, 19]) have been analyzed for their focus on the design process. Such approaches have been found to be mostly domain-agnostic – they provide general methods [11, 12, 19] and means of describing [8] software architectures. At the same time, they do not provide specific recommendations and approaches to system design processes that take into account the specific nature of systems. For example, existing approaches do not take into account the conceptual differences between the design processes of embedded real-time systems for robotics, PLC programs, enterprise web-based systems such as Enterprise Resource Planning (ERP) systems, etc.

Industrial automation and systems engineering approaches (e. g., [2, 3, 16, 21, 23, 24]) have been analyzed for their focus on system structure. They were found to be domain-specific and define hierarchies, levels, and interaction zones (i. e., high-level system structure and interaction principles), but do not offer a prescriptive process for designing the computer software architecture of heterogeneous components within this structure.

Among the analyzed standards, models, and scientific studies, none were found that would describe the design processes of CSSs taking into account the fundamental differences and commonalities of the architectural aspects of different categories of systems. This analysis allowed to clearly articulate a central research gap: the lack of a single methodology that differentiates the process of architectural design of CSS in the field of automation based on fundamental differences in architectural aspects and CSS tasks. Taking them into account for different CSS categories in the field of automation would allow to define differentiated architectural design processes for each specific type of system in a standardized way. At the same time, this would allow to maintain sufficient generalization of processes and design methods to cover a wide range of systems of the corresponding type. The analysis conducted reveals a critical gap at the junction of the disciplines of automation, computer-integrated technologies and software engineering. Existing approaches and models of CSS architecture do not define and do not cover the differences between types of systems in accordance with the differences in their

design processes. Although there are general software design processes and industry-specific systems frameworks, none of the known methodologies offer a systematic architectural design process that explicitly considers the fundamental nature of a computer software system, especially in the context of automation.

Bridging this gap requires a new CSS taxonomy in automation, focused on the fundamental differences in the architectural aspects of such systems.

The object of research is the processes of CSS design in the field of automation. In the context of this research, computer-software systems are, in essence, software systems that run on programmable computer systems (which are considered as hardware infrastructure – the environment in which the software runs). Within the framework of this research, the term "computer-software system – CSS" covers software together with the environment in which it operates. Computer hardware is considered as an element of the infrastructure – the environment in which the software runs. The software part is software applications that operate within the infrastructure and, in the case of cyber-physical systems, interact with the hardware infrastructure.

The subject of research is fundamental architectural aspects and the taxonomy is focused on the differentiation of architectural design processes.

The aim of research is to develop a paradigm for type-centric architectural design of CSS in the field of automation, based on the CSS classification and the model of internal elements and levels of CSS. The paradigm forms the foundational platform for a unified methodology for architectural design of CSS, focused on the type of system, in the field of automation.

Within the framework of this research, the following objectives are set:

– to develop a model of elements and levels of computer software systems;
– to develop a CSS classification in the field of automation, based on architecturally important aspects of systems;
– to validate the defined taxonomy by mapping it to established industrial models [1, 3];
– to consider practical aspects of applying the proposed paradigm on the example of an industrial automation system.

The proposed paradigm defines a specific architecturally-oriented point of view on CSS, describing a model of the internal structure (model of CSS elements and levels) and a CSS classification based on the fundamental architectural nature. This is the foundation for various methods and methodologies for designing CSS, defining a basic system of concepts, terms and general approaches. This specific feature gives the approaches and models proposed in this study the characteristics of the CSS architecture design paradigm, which forms the theoretical basis for practical design methods and methodologies.

This article first presents a taxonomy that defines four base CSS types in automation. The main architectural aspects for each type are then described in detail, and the final stage is the mapping of these types onto established industrial models [1, 3] to demonstrate their applicability.

2. Materials and Methods

2.1. General research design

This research is conceptual and based on qualitative theory building methods [42]. The main objective is to use the taxonomy to address an identified gap in the architectural design of CSS in the field of automation. It is intended to serve as a foundation for the development of a type-centric paradigm for CSS design. Given this, a constructive approach to the research was chosen [43].

The methodology was structured into two consecutive phases, each of which is based on the results of the previous one:

– conceptual modeling and synthesis of a type-centric paradigm for CSS design. This is the core of the research, where an iterative synthesis process was applied to develop and formalize the proposed paradigm;
– qualitative validation and verification. In this phase, the developed paradigm was tested for coherence, completeness, and integrability with existing models. This was done by applying it to the classification of typical systems at different levels of existing reference models [1, 3], including establishing correspondences between the levels and the types of systems prevailing at them.

This research was conducted using standard hardware (workstation-class personal computers). Specialized hardware (e. g., PLCs, embedded systems, industrial computers) was not used, since the research is theoretical, not experimental. For visual modeling and iterative development of the taxonomy hierarchy, as well as for constructing conceptual diagrams, the graphical diagramming tool draw.io was used.

2.2. Paradigm development methodology

The goal of this phase was to create (construct) a new, practically-oriented CSS paradigm in the field of automation based on taxonomy. Taxonomy is an established method of scientific research used to structure a complex subject area by organizing its entities into categories [44]. This research used a systematic iterative inductive-deductive approach [45].

The research process was implemented in the form of the following four consecutive stages:

1. *Defining the goal and scope (defining meta-characteristics).* Based on the analysis of the scientific and practical problem, the initial step was to clearly define the goal of the paradigm. It was established that the paradigm should not only be descriptive, but also prescriptive and generative. That is, it should not only classify systems, but also serve as the basis for the formation of specialized architectural design processes focused on the fundamental features of the architectural aspects of systems of various types. The scope of application was defined as CSS at all levels of automation – from embedded systems and industrial PLCs to enterprise information systems.

2. *Inductive identification of categories.* At this stage, a qualitative analysis was conducted of a wide range of examples of real systems identified during the literature review, as well as the systems considered in Section 3: ERP, manufacturing execution system (MES), PLC-based systems, machine learning models, agent systems, computer games, operating systems, drivers, etc. These examples (empirical objects) were analyzed to identify fundamental common features and differences. This allowed the formation of initial, emergent clusters of systems that have common aspects from an architectural point of view.

3. *Deductive formation and justification of key criteria.* This was a key analytical step. Instead of using superficial criteria (e. g., programming language, scope of application), a search was conducted for a fundamental, essential discriminator. As a result of the analysis, it was found that the most significant criterion that directly affects architectural decisions during design (as will be shown in section 3.3) is the fundamental operating model of the system. Three main operating CSS models in the field of automation were identified (detailed in section 3.4):

1) event-driven (execution of logic in reaction to requests, events, triggers);

2) continuous, loop-based (functioning in a continuous cycle of interaction with the environment);

3) streaming, data-oriented (data processing in a pipeline format).

4. *Iterative synthesis and detailing of the paradigm.* At this stage, the initial categories (from stage 2) and criteria (from stage 3) were iteratively agreed:

1) formation of base types: application of operational criteria to empirical clusters allowed formalizing the four base types described in 3.1;

2) hierarchical detailing: each base type was analyzed for the presence of stable subgroups. This led to the definition of subtypes;

3) complex case analysis: systems that did not fit neatly into one category were analyzed (e. g., IoT devices, computer games). This

ISSN-L 2664-9969; E-ISSN 2706-5448

analysis led to the formalization of the concept of hybrid systems (section 3.5), ensuring the exhaustiveness of the paradigm;

4) orthogonal structural analysis: it was determined that in addition to the type of system, its structural composition is important. A decomposition method was carried out to isolate common elements and levels of the system, which included the formation of additional classifications for certain components of different levels of the system. This allowed to formulate an orthogonal model of the elements of the CSS (section 3.2), which complements the paradigm;

5) composition analysis: it was investigated how individual systems (corresponding to the types of classification) are combined into larger ones. This led to the definition of composite systems (systems-of-systems) and criteria for division into subsystems (section 3.6).

2.3. Qualitative validation strategy

The aim of this phase was to qualitatively assess and verify the developed paradigm for its logical consistency, completeness and practical utility, i.e. the applicability and integration of the CSS types defined in this research with standardized industrial systems architectures.

The chosen validation method consisted in deductive mapping of the proposed paradigm onto two widely recognized standardized reference models of industrial systems.

Validation objects:

– Purdue Enterprise Reference Architecture (PERA) [1];
– Industrial Internet Reference Architecture (IIRA) [3].

These models were chosen because they represent two different, albeit related, points of view on automation systems. Purdue focuses on the hierarchy of control functions and enterprise logic in industrial computer-integrated manufacturing systems, while IIRA focuses on functional domains and cross-cutting aspects in industrial Internet of Things systems.

Validation process. The validation process was as follows:

1. *Structural analysis of reference models:* The levels of the Purdue model [1] and the functional domains and levels of the three-tier architecture of IIRA [3] were analyzed in terms of the typical software systems that implement them.

2. *Classification:* Each of these typical software systems (e. g., PLC programs, Supervisory Control and Data Acquisition (SCADA), MES, ERP, analytical platforms, etc.) was classified using the criteria developed in Phase 1.

3. *Conformance analysis:* An analysis was conducted to see how well the proposed paradigm was consistent with existing models. The aim was to assess the applicability and integration of the CSS paradigm defined in this research for classifying typical CSS at different levels of existing leading reference models [1, 3].

3. Results and Discussion

3.1. Model of computer-software system elements and levels

The proposed CSS design paradigm for automation systems is based on two fundamental components:

– a component that considers the internal nature of the CSS – a model of CSS elements and levels;
– a component that considers the CSS as a whole and their decomposition into smaller subsystems – CSS classification based on architecturally important aspects.

Within the framework of the proposed paradigm, the internal CSS aspects are defined in the form of a model of CSS elements and levels.

Each CSS is divided into two levels:

1) *application (software) level:* software that implements the direct logic of control, automation, and business logic of the system;

2) *infrastructure level:* a set of hardware and software that create the environment (server computer, PLC, microcontroller, operating system, containerization environment, etc.) and general tools (databases, message brokers, etc.) leveraged by the application level.

A software component, both at the application and software infrastructure level, is defined in this research as an independent deployment unit – that is, a separate software application (which may have a modular structure internally).

Application software components are classified as follows:

– *Custom software or modified off-the-shelf software* (e. g. microcontroller firmware, PLC program, equipment health forecasting system, customized ERP, modified open source mobile application, etc.).
– *Self-hosted off-the-shelf software.* Deployed within the system infrastructure. Application level architectural design aspects are not considered, since the component is leveraged as a "black box" (e. g. SCADA, MES, ERP, CRM, CMS, etc.).
– *SaaS:* considered as third-party systems with which integrations are established. Such systems are outside the infrastructure and boundaries of the system under consideration (e. g. cloud ERP).

The CSS infrastructure is further divided into three layers:

1. *Hardware layer* – the level of physical or cloud hardware infrastructure (printed circuit boards, microcircuits on boards, sensors, actuators, physical servers, data storage systems, network elements, leased physical servers in the cloud, etc.). The following types of hardware infrastructure deployment are defined:
   – on-premise – local infrastructure in a local data center or in a production field (industrial computers, PLCs, etc.);
   – embedded – the system infrastructure is part of the device ("built into the device"). Typically, the infrastructure of such a system is a printed circuit board that contains one or more chips (integrated circuits). They run application software – usually called "firmware" (for example, a microcontroller, microprocessor, field-programmable logic integrated circuit (FPGA) or other types of programmable chips/systems-on-a-chip);
   – end-user device – the system infrastructure is represented by the user's device; examples: mobile, desktop, web applications SPA, etc.;
   – cloud – infrastructure rented from a cloud provider.

In many industrial scenarios, the type of placement of the system's hardware infrastructure can be "embedded" within the architectural design of the subsystem. But such a subsystem as a whole can be an element of a higher-level system with, for example, on-premise type of hardware infrastructure.

For example, a special control module built on a printed circuit board with a microcontroller and/or FPGA that executes control algorithms is a system with an embedded infrastructure type, but can be a subsystem of a complex industrial automation system with an on-premise infrastructure type.

In the case of cloud systems, a "serverless" approach is often used, in which the hardware layer is completely abstracted by the cloud provider within the framework of the higher-level services provided.

2. *Platform layer* – an infrastructure layer that describes the software that forms the deployment/execution environment for application software and software components of the infrastructure layer.

This layer includes:

– operating systems (e. g., Linux, Windows Server);
– virtualization environments (e. g., Proxmox, VMware ESXi);
– cloud virtual machines (AWS EC2, Azure VM, Google CE, etc. – which combine hardware and platform infrastructure);
– containerization systems (e. g. Docker) and container orchestration platforms (Kubernetes, OpenShift);
– cloud platforms as a service (PaaS) – AWS Elastic Beanstalk, AWS IoT Core, AWS ECS, Heroku, etc.;
– client runtime environments such as web browsers (which can be considered platforms for executing the web frontend).

3. *Software layer* – infrastructure software required for the functioning and integration of application components, as well as for

providing operational support and monitoring of the system (databases, message brokers, CI/CD systems, etc.).

It is important to note that within the framework of this methodology, third-party systems with which integrations are established do not belong to the system infrastructure.

Infrastructure software (both at the platform and software infrastructure levels) is divided into:

– Self-hosted – software deployed directly on hardware or platform resources.
– Managed-service – cloud-based managed services.
– SaaS – SaaS infrastructure services (used exclusively through the SaaS service API) provided by cloud providers.

Additionally, the CSS infrastructure is classified into two vertical categories by purpose – identifying whether the specific infrastructural setup serves the main functioning of the system or supports the cross-cutting operations:

– *Workload infrastructure*: infrastructure that ensures the functioning of the system.
– *Supportive infrastructure*: infrastructure that provides auxiliary functions, including monitoring (logging, metrics), continuous deployment (CI/CD), etc., necessary for the observability and ease of operation of the system.

Fig. 1 illustrates the elements of a computer software system according to the definition of this research.

Fig. 2 demonstrates a general case of a typical CSS in automation at higher levels of the hierarchy [1, 2], where information systems prevail.

Fig. 3 shows the category of cloud SaaS infrastructure services that are outside the hardware and platform infrastructure of the system, as well as SaaS applications that are considered third-party integrations.

For systems at lower levels of the automation hierarchy [1, 2], the layer structure is shown in Fig. 4. These levels are usually dominated by process control systems, embedded systems, and robotic systems. They are often deployed on chips that do not support full-featured operating systems or runtime environments (e. g., most microcontrollers).

Application software, often referred to as firmware (especially in the context of embedded systems), is deployed directly on the hardware infrastructure without any intermediate software infrastructure layers.

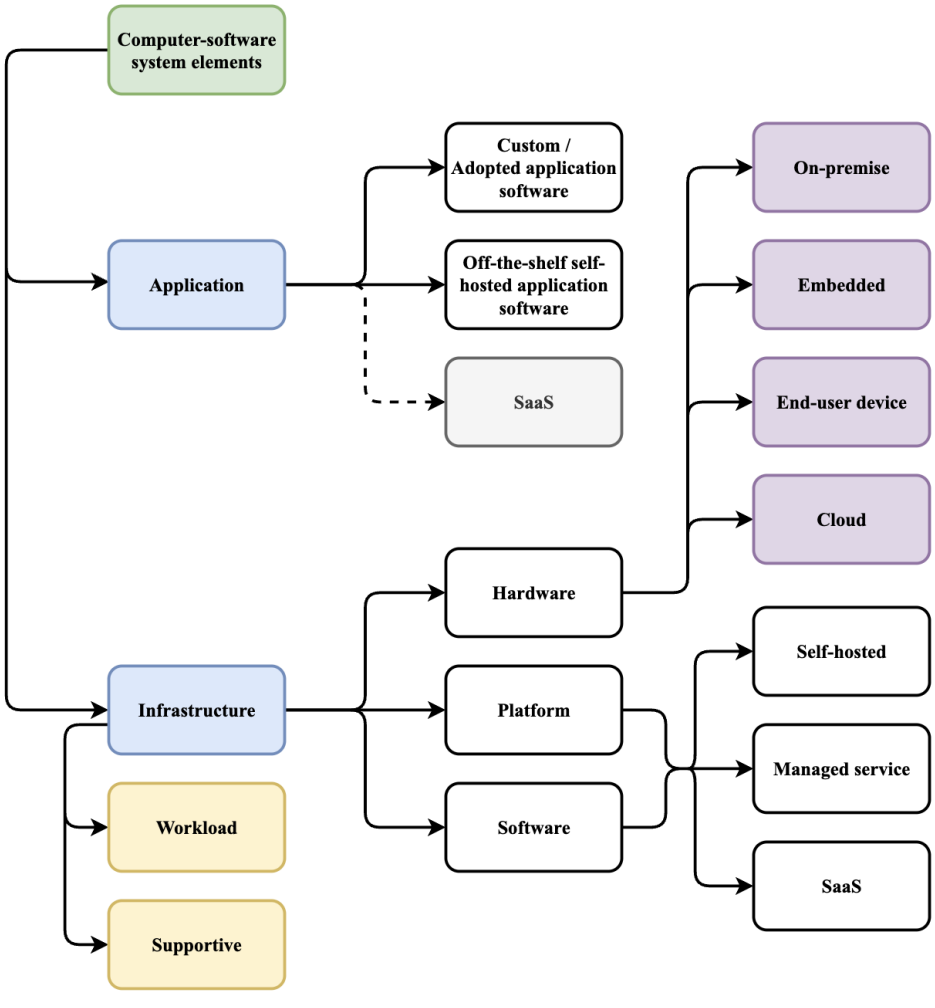

Fig. 1. Elements of a computer software system

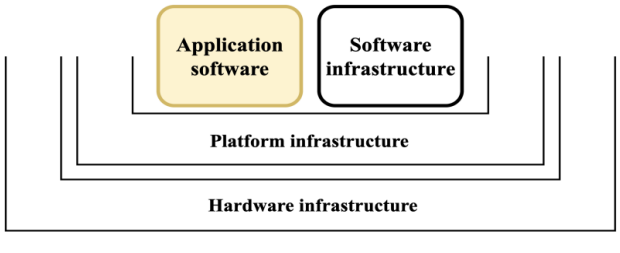

Fig. 2. General case of a higher-level computer-software system in automation

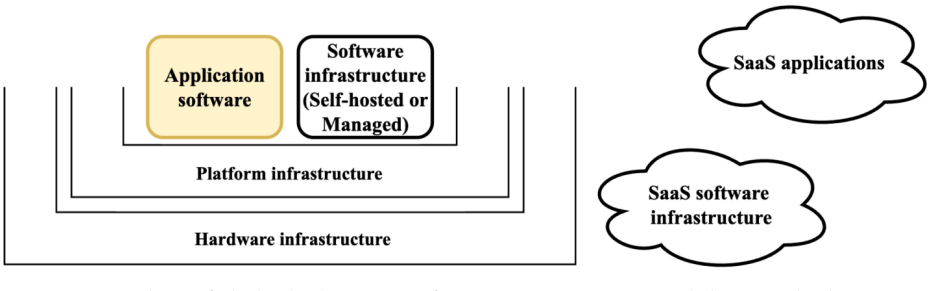

Fig. 3. General case of a higher-level computer software system in automation including SaaS cloud services

ISSN-L 2664-9969; E-ISSN 2706-5448

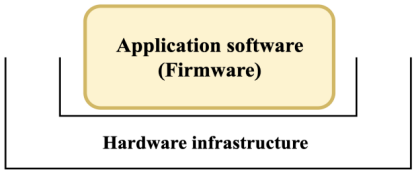

Fig. 4. The case of embedded firmware software systems

PLCs, which are most commonly used to implement process control in industrial automation, usually contain a thin layer of real-time operating system (RTOS). This layer is called "PLC firmware". It is considered a platform-level infrastructure within the scope of this research because it provides an environment for the PLC program to run.

In contrast, most RTOSs used by microcontrollers (e. g. FreeRTOS) are not considered platform infrastructure within the scope of this research. They are included directly in the application software as libraries, i.e. they are included in the same deployment unit as the application component (firmware).

3.2. Computer-software systems classification

3.2.1. Description of system types

The CSS classification distinguishes four base CSS types in the field of automation, based on the differences in their architectural nature and the corresponding architectural design aspects.

Within the framework of this classification, the following types of systems are defined:

1. *Behavior-oriented systems:*

1) domain-data-model-oriented systems (for example, enterprise information systems such as ERP, MES, Customer Relationship Management (CRM), etc.);

2) computation-oriented systems (for example, machine learning model inference serving system);

3) orchestration-oriented systems (for example, retrieval-augmented generation agent systems).

2. *Continuous systems:*

1) control-oriented systems:

– physical control systems (for example, embedded control systems, industrial automation systems based on PLC, etc.);

– virtual control systems (e. g., interactive simulations/computer games, Kubernetes controllers, etc.);

2) (continuous) agent systems (e. g., high-level robot trajectory planning system, self-driving car decision-making system, non-player character control logic in a computer game).

3. *Data-flow-oriented systems* (e. g., analytical processing systems, artificial intelligence model training systems, data-driven predictive diagnostics systems, etc.).

4. *Hardware resource management systems* (e. g., operating systems, drivers, hypervisors, etc.).

The types form a hierarchy in which four base types are at the core: behavior-oriented systems; data-flow-oriented systems; continuous systems; hardware resource management systems (Fig. 5).

Software for different types of automation systems at different levels of hierarchies, including industrial automation systems, robotics, information systems in automation, etc., can operate in different environments. For example, such software can operate in embedded systems, on programmable logic controllers (PLCs), industrial computers, in cloud environments, etc.

Systems of the same type can differ in their nature in terms of infrastructure. For example, embedded software deployed on a microcontroller placed on a printed circuit board (PCB) is different from PLC programs deployed on an industrial controller in an industrial installation. However, if to consider the infrastructure (the environment in which the code is executed and runs) as a separate aspect, the architectural aspects associated with the application software remain largely the same in both examples. Infrastructure can have different implementations in different systems. However, since it is isolated as a separate aspect, the nature of the software in both cases is similar, in terms of cyclic or interrupt-driven nature of operation. Both cases are characterized by an orientation towards "continuous" reading of input signals and formation of output control actions. In other words, the defined types of software systems reflect the fundamental similarities and differences in the architectural design of computer software systems. This forms the basis for the development of specialized (to system types) architectural design methodologies that can be applied to a wide range of systems of the same type.

Behavior-oriented systems are operational-oriented systems that automate business processes, manual tasks and operations by executing behaviors in response to certain triggers: API requests, user actions, events, etc. Examples of this type of systems are most web, desktop and mobile applications that perform certain behaviors in response to received actions, requests or events.

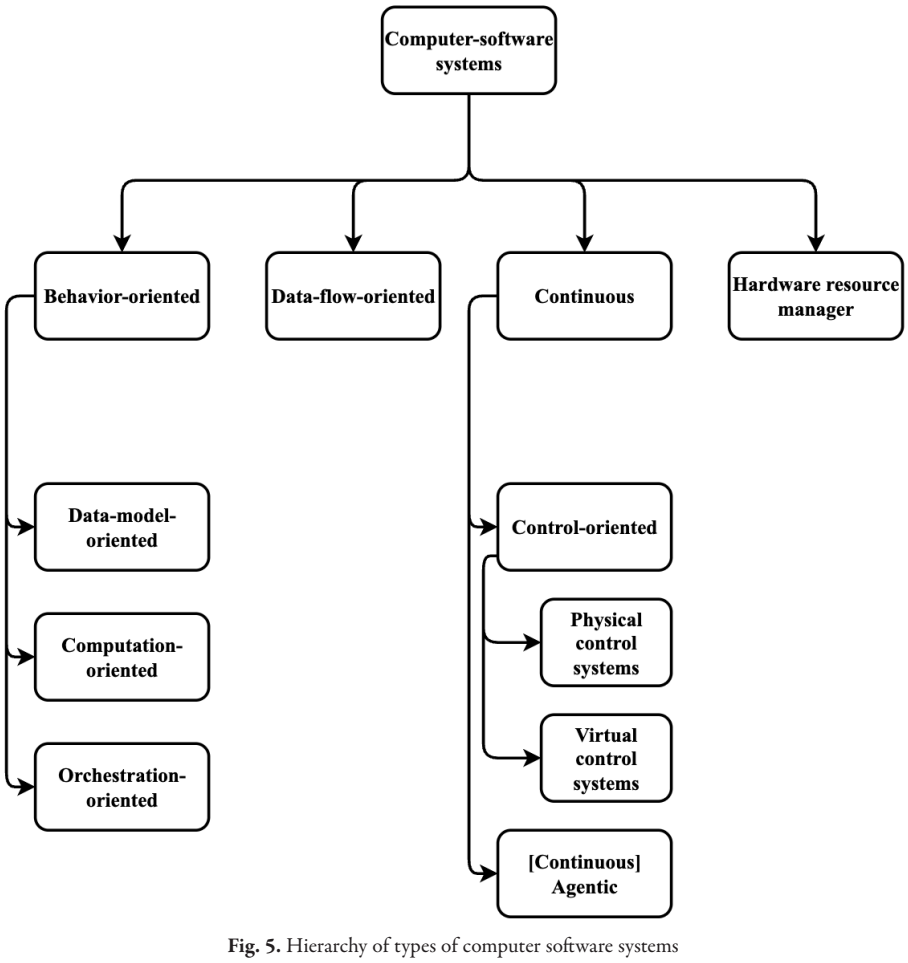

Fig. 5. Hierarchy of types of computer software systems

Within behavior-oriented systems, three subtypes have been defined that describe the different nature of such systems:

1. *Domain-data-model-oriented systems* are systems based on a domain model. Such systems automate business processes by implementing behaviors according to functional requirements (also complying with non-functional requirements and constraints). Behaviors manage data belonging to the system (defined in the domain data model) according to business rules and utilize integrations with other systems. Such behaviors are focused on operations related to data management, often forming steps or elements of business processes that are automated.
2. *Computation-oriented systems* are systems focused on performing computational operations and often do not have a domain model. Behaviors of such systems perform calculations or computations and return or publish the results.
3. *Orchestration-oriented systems* are systems that are oriented towards the execution of behaviors, the logic of which mainly consists of orchestrating the behaviors of integrated, bottom-up behavior-oriented systems. For example, an event-driven agent system that invokes the behaviors of lower-level integrated systems based on local agent decision-making. Such systems use the API of integrated systems to invoke their behaviors according to local decision-making logic.

Continuous systems are software systems that are characterized by continuous, real-time interaction with the physical or virtual environment. Unlike behavior-oriented systems that operate in response to specific triggers, events, or requests, continuous systems operate in a continuous cycle (or cycles).

They are characterized by an orientation towards:

– continuous reading of input signals from the environment (state of the control object, disturbances, external influences, etc.);
– processing of the read signals according to the internal decision-making or control logic;
– changing of internal states (e. g., transitions between operating modes or robot behaviors [46]);
– influencing the environment through output signals.

Such systems are fundamental to domains where time accuracy, feedback, and perception of the environment are critical, including robotics, industrial automation, autonomous vehicles, simulations, etc.

Within continuous systems, two subtypes have been identified that demonstrate the different nature of this type of system: control systems and continuous agent systems, both of which are continuous in nature but have different goals and solve different problems. Control systems, in turn, are divided into physical control systems and virtual control systems to emphasize the difference in the environments in which they operate.

Among all CSS types, only physical control systems are dedicated to interact with the physical world through sensors and actuators. All other types of systems operate with virtual concepts and do not directly control physical processes.

In general, two types of agent systems are distinguished: continuous, which are more common in control and robotics systems, and event-driven, which are more common in information technology and enterprise systems. In continuous agent systems, the agent process is continuous and directed towards achieving a goal, and any behavioral interaction with the system can only modify this process. This means that the agent operates autonomously in a continuous mode, rather than simply executing logic in response to an event, which highlights the main difference between behavior-driven and continuous agent systems. In event-driven agent systems, agents execute discrete workflows in response to specific events, i.e., such systems can be considered behavior-oriented (usually orchestration-oriented) systems and designed accordingly.

Data-flow-oriented systems are data processing and analytics systems in which the main aspects and architectural interests are focused on the ingestion, transformation, cleansing, processing, storage, and analytical retrieval of data. When designing such systems, architectural attention is usually paid to the data lineage, as well as the efficiency and manageability of the data flow or data pipeline in the system. In most cases, systems of this type work with large amounts of data. Examples are machine learning model training systems, analytical systems, business intelligence, ETL-type systems, batch or streaming data processing systems, etc.

Hardware resource management systems are software systems responsible for abstracting, allocating, monitoring, and coordinating access to physical computing resources such as the processor (CPU), memory, storage, input/output devices, etc. Such systems form a layer over physical computing resources, abstracting the details and complexity of their management and operation, creating a basis for higher-level software applications by providing a basic platform. Examples are operating systems, device drivers, hypervisors, etc.

Hardware resource management systems are not included in the scope of developing applied computer software systems in the field of automation. Their internal structure and architectural features are not the subject of this research. At the same time, this type of system is present at the infrastructure level of the CSS considered in the research, but performs the role of an infrastructure element that provides a platform for application software, and does not act as a system that is the object of architectural analysis.

3.2.2. Main architectural aspects of design for different types of systems

Architectural aspects and design tasks of the CSS differ depending on the type of system defined in the presented taxonomy. This difference is the main purpose of the proposed classification – it forms the foundational basis for architectural design methods focused on the type of system.

According to the elements of the system, the architectural design process covers the application and infrastructure levels, taking into account the infrastructure categories, levels and types of placement of the hardware component.

Architectural design of application software at the application level is performed only for the class of "custom or modified off-the-shelf software". For "off-the-shelf" application software components, architectural design of the application level is not considered, since such systems are treated as a "black box" with respect to their internal architecture.

From a technical point of view, the following aspects are important for different types of systems:

Behavior-oriented systems:

– system behaviors that are exposed through the APIs of application components, forming external system APIs, as well as internal APIs for inter-component interaction;
– system component model (and assigning behaviors to application software level components);
– system persistent data models (in the case of domain-data-model-oriented systems);
– behavioral logic aspects (business rules – mainly for domain-data-model-oriented and orchestration-oriented systems; computational models – mainly for computational-oriented systems);
– system infrastructure [software and hardware] (for some systems this is a critical aspect – for example, for cloud-native and embedded systems; for others, only dependencies at the infrastructure software level with which application components interact are important, since the systems themselves are platform and hardware independent);
– integration with other systems.

Data-flow-oriented systems:

– data lineage within the system (which allows to understand where data comes from, how it is transformed, and where it is going);
– component model (including interaction contracts between components and certain aspects of component logic);

ISSN-L 2664-9969; E-ISSN 2706-5448

– persistent data models (mostly analytical data models);
– system infrastructure;
– integrations with other systems.

Continuous systems:

– component model of the system;
– goals and principles of system operation (for example, principles of goal achievement for continuous agent systems or control/regulation models and rules for control systems);
– for physical control systems – signals and pins (which pins are responsible for interaction with other hardware and software components of the system);
– parallelism of execution/control cycles (since continuous logic, including control logic, is usually cyclic or built on timer interrupts);
– system infrastructure. For embedded systems – including the design of the printed circuit board, various microcircuits on it, integrated peripherals, sensors, actuators, etc. For on-premise systems – hardware and software of the platform and software levels of the infrastructure. For cloud systems – infrastructure based on cloud services).

3.2.3. System classification criteria

The system classification method is based on the fundamental properties of the system, its purpose and the tasks it performs, which can be reduced to a model of the system's functioning:

– if the system operates in a discrete, event-driven, request-driven or trigger-driven mode (often, but not necessarily, transactional), it is a behavior-oriented system;
– if the system's functioning is based on one or more continuous real-time loops usually implementing interaction with the physical or virtual environment, it is a continuous system;
– if the system performs data movement in the form of a pipeline, the main purpose of which is data transformation or final processing (for example, training a machine learning model), it is a data-flow-oriented system.

It is worth noting that the definition of the CSS types based on the operational model means that the type of system depends on the nature of its functioning in relation to the functions or tasks performed, and not on the technical means of their implementation. For example, an information technology system in automation can be a chatbot that integrates with a social messaging system (working as a higher-level system of the automation pyramid [1, 2]). The chatbot itself can be implemented according to the short-polling or long-polling pattern, i. e. the bot periodically calls a certain API of the messaging system in a cycle. When incoming messages appear for processing, the bot invokes the corresponding behaviors for their processing. Although the system has a cyclic (continuous) nature, this is only a detail of the technical implementation, and not its operational model. From the point of view of the operational model, the system remains behavior-oriented: it reacts to the arrival of a new message by calling a behavior. Similarly, pin-change interrupts in a continuous physical control system can be treated as behaviors: the interrupt service routine (ISR) is indeed a behavior of the system. However, this specific detail of many physical control systems within the scope of this research refers to continuous systems, since the operational nature of the system as a whole remains continuous for most physical control systems that use interrupts. In addition, interrupts can be timer-based, effectively acting as an alternative technical means of implementing a continuous loop in certain cases.

3.2.4. Hybrid system types

In the real world, automation systems often combine characteristics of more than one base type. Such systems are classified as hybrid systems.

The hybrid system design process combines the design processes of the corresponding base system types, applying methods from different base types to a single system, in order to fully reflect the details and features inherent in each base type in the architectural design.

For example, embedded Internet of Things (IoT) systems (classified as physical control systems) and computer games (classified as virtual control systems) include both behavior-oriented logic and control loop logic.

Behavior-oriented logic changes a certain state of the application, which is then used as a setpoint for the ongoing control logic.

API endpoint or listener of a user actions belongs to the behaviour-oriented part – it represents a behavior invocation.

Control loops, such as interaction with sensors/actuators or game-world control within game-loops, are a continuous part.

Interrupts occupy an intermediate position: calling an ISR is a behavior (as mentioned earlier), but this detail does not make the system hybrid in nature.

Typical combinations of hybrid systems that are often encountered in practice:

Different parent types:

– data-flow-oriented systems (base type) + behavior-oriented systems (base type);
– behavior-oriented systems (base type) + control-oriented systems (subtype of continuous systems);
– behavior-oriented systems (base type) + continuous agent systems (subtype of continuous systems);
– hardware resource management systems (base type) + control-oriented systems (subtype of continuous systems).

Same parent type:

– control-oriented systems (subtype of continuous systems) + continuous agent systems (subtype of continuous systems);
– domain-data-model-oriented systems (subtype of behavior-oriented systems) + computationally oriented systems (subtype of behavior-oriented systems);
– domain-data-model-oriented systems (subtype of behavior-oriented systems) + orchestration-oriented systems (subtype of behavior-oriented systems).

For hybrid systems, the property of associativity applies, i.e. a hybrid system can be a composition of more than two base types of computer software systems.

3.2.5. Composite systems/systems-of-systems and system decomposition

To simplify the design process, it is recommended to decompose a system into subsystems of different types, where possible, instead of forming hybrid systems.

Subsystem decomposition means setting boundaries around one or more application-level components (along with associated infrastructure-level elements) and considering them together as a separate (smaller) system. In the context of this research, the key factor that allows subsystem decomposition is the ability to physically partition the system at the application level (one application-level component cannot be partitioned into multiple subsystems).

A subsystem is a system itself, which, in relation to the higher-level system, functions as its component. Subsystems have physical boundaries – they do not share an application runtime. If a system cannot be physically partitioned, it is considered hybrid and is designed using a combination of system design processes of different types.

Decomposition of a system into subsystems of base types (sometimes hybrid types) forms a composite system.

Within this paradigm, subsystems are defined by the following factors:

– *Independence and integrity.* A subsystem can exist and achieve its goals autonomously.
– *Uniform system type.* The elements (including application-level components) of a subsystem are described by a single system type in the context of considering the subsystem (even if this type is hybrid).
– *Atomicity factor – unit of deployment.* A system that is a unit of deployment at the application level (a single application-level component) cannot be divided into subsystems.

A higher-level system can exist solely to orchestrate its subsystems – in the literature this is sometimes called a system-of-systems orchestrator [3]. Or it can be a complete system with its own goals, different from a simple orchestration, the components of which are separate subsystems (for example, an embedded robotic system that is a subsystem of a larger industrial system).

It is worth noting that components of non-hybrid systems, in some cases, can also be represented from the points of view of several types of systems (from different perspectives), if the component serves as an integration point between systems of different types.

For example, a microservice can be part of a behavior-oriented system as a custom application-level component, and at the same time be part of a data-flow-oriented system model as a data source that generates events for analytical calculations. The infrastructure of the subsystems can be partially (only integration elements) or completely shared.

It is also worth noting that a component of the infrastructure level of the system or a self-deployed off-the-shelf application-level component can be a system on its own, but it makes no sense to consider its internal structure. Such systems are treated as "black box" components of the corresponding levels and types of components. A classic example is a database management system (DBMS), which is a system in itself if considered separately. But within the scope of the architectural design of computer software systems using a DBMS, it is considered as a component of the software infrastructure layer, without considering its internal structure.

To represent the structure of a higher-level system, a high-level component diagram can be used, which describes how subsystems are combined into an overall system. The components in such a diagram usually represent subsystems or higher-level system's own application-level or infrastructure-level components.

3.3. Mapping of identified system types to established models in industrial automation

3.3.1. Types of software systems at different levels of the automation pyramid (Purdue model)

The automation pyramid [1] is a generally accepted standard that reflects the hierarchy of systems in large-scale computer-integrated manufacturing and enterprise automation systems. Therefore, it is important to consider how the paradigm of a typocentric approach to CSS proposed in this work can be applied to classify and design systems at different levels of this hierarchy.

It was determined which CSS types, according to the proposed classification, prevail at each level of the pyramid (Fig. 6).

As can be seen from Fig. 6, the predominant CSS types at different levels of the pyramid are as follows:

1. *Control level* – automated control systems (mainly based on PLCs, as well as embedded control systems based on microcontrollers or microprocessors). At this level, Continuous systems (in particular, the subtype – Physical control systems) dominate. Possible variations: hybrids of Continuous and Behavior-oriented systems (especially in the case of IoT-enabled "smart devices"). A continuous system can be deployed on hardware without an operating system (bare-metal) or based on an operating system (including RTOS).
2. *Supervisory level* – at the supervisory control level, the following types of systems are found:
   – behavior-oriented systems (including SCADA);
   – continuous agent systems (for automated targeted supervisory control);
   – data-flow-oriented systems (for near-real-time data analytics).
3. *Planning level* – behavior-oriented systems (including MES) and data-flow-oriented systems for analytics and forecasting prevail.
4. *Management level* – behavior-oriented systems (including ERP) and data-flow-oriented systems prevail, especially for analytics and forecasting.

3.3.2. Types of software systems in different functional domains and levels of the three-tier architecture of IoT systems (according to IIRA v1.10)

Another important foundational document shaping the understanding of industrial distributed network systems is IIRA [3] (Industrial Internet Reference Architecture), which describes approaches to the design of Industrial Internet of Things (IIoT) systems.

Fig. 7 shows how different types of software systems can be represented within the functional domains of IIRA at the tiers of the three-tier architecture proposed by IIRA. It is worth noting that the belonging of functional domains to certain tiers of architecture, according to IIRA, reflects the predominant correspondence, but does not exclude the appearance of systems of a certain domain at other tiers. The predominant types of software systems in the tiers and functional domains of IIRA:

1. *Edge tier:*
   – *Domain Control & Monitoring.* Continuous systems (physical control systems) or hybrids of continuous and behavior-oriented systems predominate.
2. *Platform tier:*
   – *Domain Information.* Main type: Data-flow-oriented systems (ETL, ELT, analytics), mainly batch and streaming data processing.
   – *Domain System Management.* Main type: behavior-oriented systems combined (often as hybrid types) with continuous systems (e. g., virtual cluster state management system) and data-flow-oriented systems (e. g., for telemetry processing).
3. *Enterprise tier:*
   – *Domain Business.* Mainly behavior-oriented systems.
   – *Domain Application.* Mainly behavior-oriented systems.

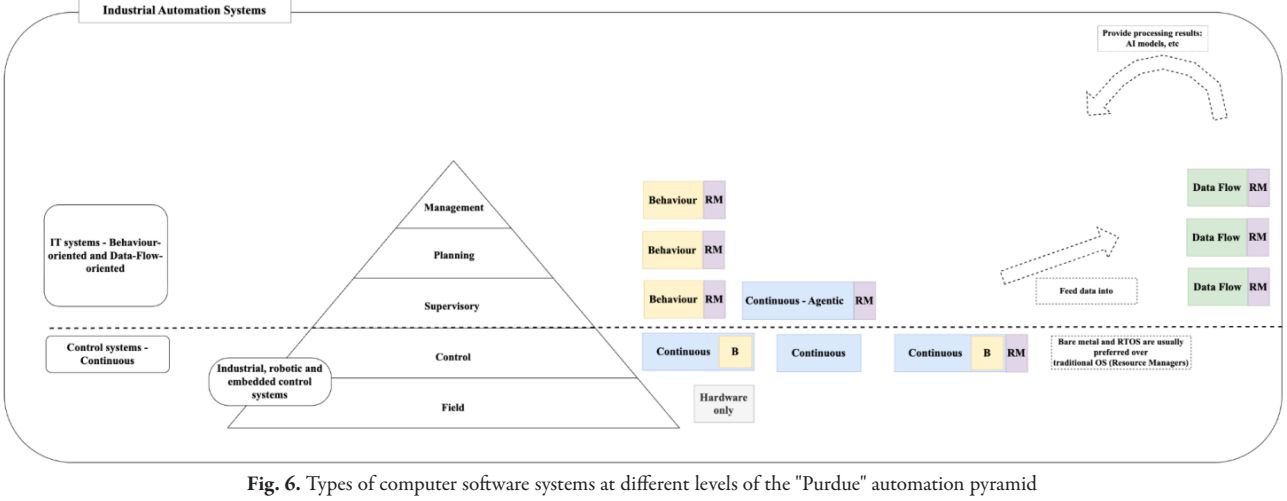

Fig. 6. Types of computer software systems at different levels of the "Purdue" automation pyramid

ISSN-L 2664-9969; E-ISSN 2706-5448

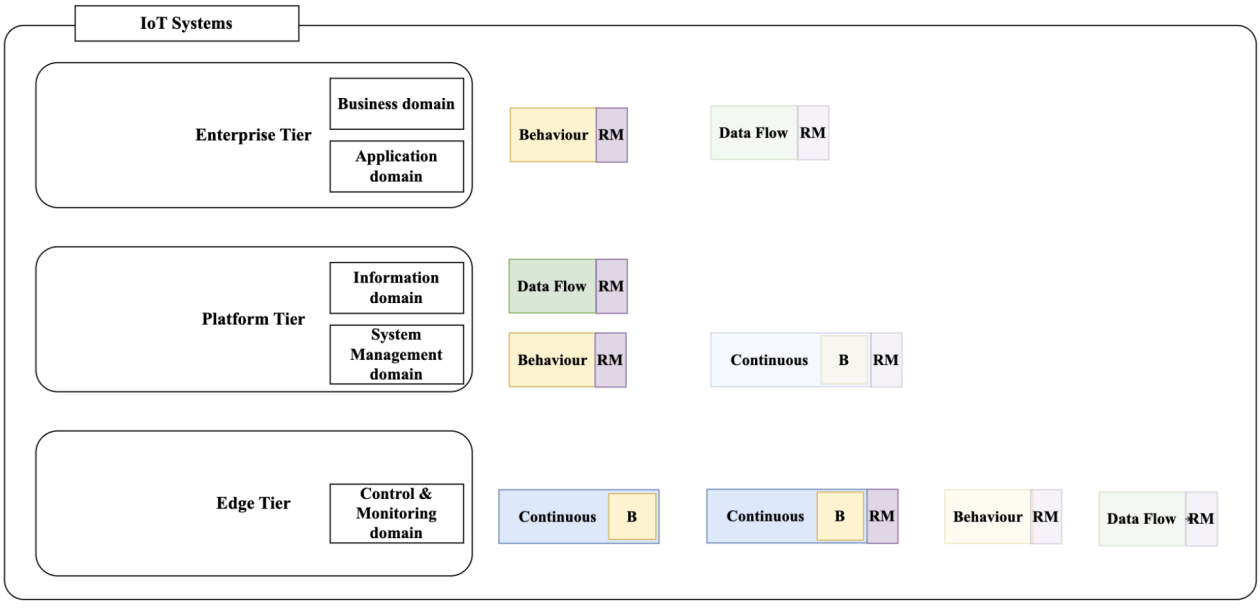

Fig. 7. Types of computer software systems at tiers and functional domains of IIRA

Given that the belonging of domains to architecture tiers does not exclude the presence of systems of these domains at other tiers, theoretically systems of all types can exist at each tier.

In most cases, the enterprise tier has the least connection with continuous systems.

3.3.3. Infrastructure of computer software systems at different levels of the automation pyramid and the three-tier IoT IIRA architecture

As part of the research, the prevailing infrastructure CSS solutions in the field of automation were determined according to the levels of the "Purdue" pyramid and the tiers and functional IIRA domains.

At the L0-L1 levels of the automation pyramid (Field and Control), as well as at the Edge tier of the three-layer IoT architecture proposed by IIRA, the hardware infrastructure consists of local hardware elements, i.e. those located in close proximity to the controlled process or object. Such equipment belongs to the on-premise (and often also embedded) type of hardware infrastructure placement, according to the terminology of this research, and includes:

– computing elements (PLCs, microcontrollers, single-board computers, computer stations, etc.);
– network elements (routers, cables);
– local connection elements (printed circuit boards);
– elements of interaction with the environment or control object (sensors and actuators).

The platform-level infrastructure, at the L1 level of the automation pyramid, may consist of RTOS device firmware (e. g., PLC firmware), operating systems, virtualization environments, or may be absent (in the case of most embedded systems based on microcontrollers and FPGAs).

At the L2–L4 levels of the automation pyramid (Supervisory, Planning, Management), as well as at the Platform and Enterprise tiers of the three-tier IIRA architecture, the infrastructure may consist of both local (on-premise) hardware infrastructure and cloud infrastructure services:

– *IaaS* – Infrastructure as a Service (hardware or platform-level infrastructure, e. g., virtual machines, private virtual clouds).
– *PaaS* – Platform as a Service (platform infrastructure with an abstracted hardware layer, e. g., cloud IoT platforms such as AWS IoT Core or Azure IoT Hub).
– *Managed Services* (software or platform-level infrastructure with, usually, a configured hardware base, for example, AWS RDS, AWS EKS).
– *SaaS* – Software as a Service (software-level infrastructure with a fully abstracted hardware layer, accessible through the API of the infrastructural SaaS service, for example, AWS Dynamo DB, AWS S3).

The use of cloud infrastructure at these levels is a common trend today. Cloud infrastructure provides flexibility and reliability. The need for them is growing due to the needs for processing big data, advanced analytics, machine learning (ML) models, elastic load scaling, etc. – tasks that are difficult to implement on local capacities.

3.4. Practical application of the proposed paradigm

For demonstration purposes, within the scope of this research, some practical aspects of the application of the proposed paradigm are considered on the example of a gas transportation process automation system at the compressor station level.

Typically, such systems consist of:

– programmable logic controllers (PLC) at lower levels of the hierarchy. They form the basis of the automatic control system (ACS) for gas pumping units (GPU) of the compressor shop;
– SCADA systems, both at the level of each compressor shop and at the compressor station level.

In addition, such systems may include other subsystems, such as:

– subsystems for diagnosing the state of GPU units, their components and GPU as a whole;
– subsystems for autonomous high-level control and intelligent decision-making;
– subsystems for monitoring the ecological and technical condition of the GPU combustion chamber, which, among other things, allow monitoring the state of the environment (air).

An example of the subsystems of such a CSS is shown in Fig. 8.

The system consists of several subsystems integrated with each other. The allocation of subsystems was carried out in accordance with the proposed, in this work, factors for determining subsystems. Each of the subsystems is independent and integral and consists of at least one independent deployment unit at the application level.

For the subsystem of "autonomous high-level control and intelligent decision-making", an example of integration with the external system – PagerDuty – is demonstrated to notify technical personnel in case of anomalies in the system operation.

Each of the allocated subsystems can be classified in accordance with the proposed paradigm, based on the architectural nature of the

system. This allows the use of specialized models and methods for designing the architecture of each of them. So:

– *PLC control program as part of the gas transport automation system.* An example of a "Continuous system", namely a "Physical control system" (the type is marked in orange in Fig. 8);

– *SCADA system.* An example of a subsystem that is implemented entirely as "Self-hosted off-the-shelf" software at the application level. The internal structure and design details of such a subsystem do not need to be considered in the architectural design of the CSS (marked in gray in the figure);

– *Analytical data processing subsystem.* Example of a "Data Flow Oriented System" (such systems are marked in green in Fig. 8);

– *Autonomous high-level control subsystem.* An example of a system that is a hybrid of two subtypes of a "Behavior-oriented system" (the type is marked in orange in Fig. 8): "Computation-oriented systems" and "Domain-data-model-oriented system";

– *Subsystem for monitoring the environmental and technical condition of a GPU.* An example of a "Physical Control System" (the type is marked in orange in Fig. 8).

It is important that the proposed paradigm provides approaches to classifying systems, thereby providing a basis for detailed methods and processes for designing systems of each type. The classification ensures the general applicability of design methods within each type, since it is based on the fundamental architectural nature of the systems.

According to the model of levels and elements of the CSS, the components of the application and infrastructure levels for this system are defined. The methods themselves for designing the CSS architecture, at the levels described in accordance with models of elements and levels of the CSS, and for the proposed types of systems are beyond the scope of this research and are illustrative. An example of a system representation from the point of view of the hardware and platform infrastructure of the system is shown in Fig. 9.

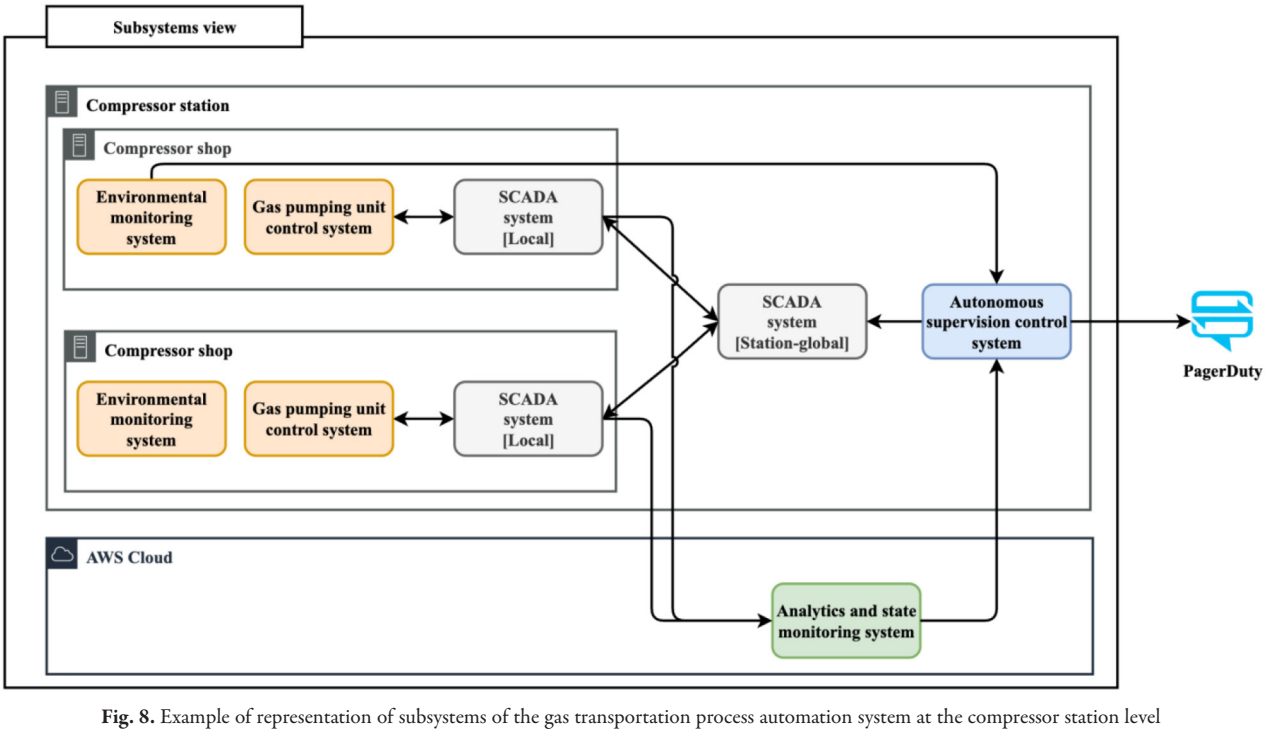

Fig. 8. Example of representation of subsystems of the gas transportation process automation system at the compressor station level

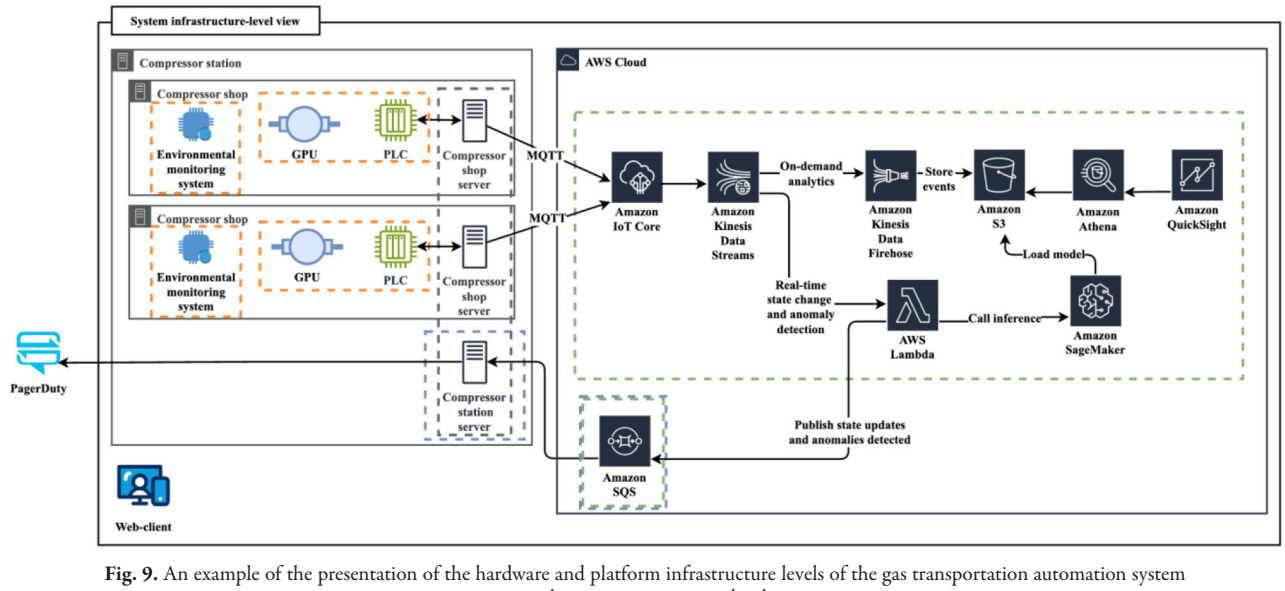

Fig. 9. An example of the presentation of the hardware and platform infrastructure levels of the gas transportation automation system at the compressor station level

ISSN-L 2664-9969; E-ISSN 2706-5448

The infrastructure of this system, at the hardware level, is an example of an "on-premise" infrastructure for all subsystems, except for the "analytical data processing subsystem", which is based on the "cloud" infrastructure. The "GPU environmental and technical condition monitoring subsystem", from the point of view of the higher-level system, is an element of the "on-premise" infrastructure, but, at the level of the subsystem itself, is an example of an "embedded" infrastructure.

Web-oriented user interfaces of subsystems belong to the "end-user-device" type of hardware infrastructure placement.

At the platform infrastructure level:

– for PLC, the firmware of the PLC itself belongs to the platform infrastructure level;

– for SCADA systems and the subsystem of "autonomous high-level control and intelligent decision-making", the platform-level infrastructure is the operating systems (e. g., Linux) installed on industrial or server computers. In the case of using virtualization, the virtualization environment (e. g., Proxmox, ESXi, etc.) or containerized environment (Docker Swarm, Kubernetes, OpenShift, AWS ECS, etc.) also belongs to the platform infrastructure level.

Similarly, for this example, the entire subsequent chain of application of the proposed paradigm can be presented, which goes beyond the scope of this research.

3.5. Discussion

The proposed type-centric paradigm for designing CSS for automation systems is based on aspects of the fundamental nature of the system architecture. This allows interpreting the research results as a proposed formalization of the system-wide and intrasystem models that form the CSS design paradigm, as a theoretical basis for CSS architecture design methodologies.

The main distinguishing features of the results are:

– differentiation of design processes for certain CSS types, which ensures methodicality and interoperability when designing CSS for IT/OT environments in the field of automation;

– ensuring standardization and unification of the CSS vision within the framework of the proposed paradigm, based on the model of CSS elements and levels.

The originality of the results lies in the development of a type-centric paradigm for computer software systems in automation. Unlike existing approaches, classifications and functional hierarchies, the paradigm proposes a classification of systems based on fundamental differences in their architectural aspects. This forms a conceptual basis for the creation of specialized models and methods for designing CSS, adapted to the nature of each type of system, which allows formalizing and standardizing the transition from requirements to artifacts of architectural design of CSS. In addition, the paradigm defines a standardized model of internal aspects of CSS design, which is especially relevant for the design of complex systems in hybrid IT/OT environments. The proposed paradigm defines a unified set of aspects and terms in the form of CSS elements and levels, which provides the basis for the systematicity and standardization of architectural design processes.

The scope of practical application of the obtained results is the standardization and differentiation of CSS architectural design processes for automation systems in hybrid IT/OT environments. The proposed paradigm forms a theoretical basis, providing the opportunity to form practical systematized, specialized and differentiated processes of architectural design of CSS for automation systems, adapted to the specifics of the nature of the systems. The proposed CSS classification provides a tool for differentiation at the system level. The model of CSS elements and levels provides a standard set of intrasystem aspects and a structural CSS metamodel. This ensures standardization and methodicality in the CSS design for automation systems, as well as interoperability of artifacts of architectural design processes.

The processes that can be formed on the basis of this paradigm will determine the stages of a sequential transition from the formation of system requirements to its architecture, defined in the format of standardized artifacts. This is ensured by the described standardized set of CSS elements, as well as by the defined common architectural aspects of systems of different types.

Also, this paradigm forms a potential basis for improving automated architectural design systems [47] by providing a basis for defining standard architectural design processes adapted to the relevant types of systems.

Limitations of research: this research only forms a basis for the potential construction of processes and methods for designing CSS, specialized for each type. The processes and design methods themselves, based on this paradigm, can be defined and standardized both globally and separately by each corporation or company that will use this paradigm.

The CSS paradigm establishes a prism for seeing such systems, important for differentiating aspects, methods and processes of architectural design based on the fundamental nature of the systems. The proposed paradigm is complete, but the design of complex large-scale multi-domain automation systems cannot be limited to it alone. This reveals the need to integrate these results with existing in-house and well-known approaches and methods to the extent necessary in each individual case.

This research focuses on CSS in the field of automation; however, this paradigm can potentially be applied to CSS outside this field.

Prospects for further research: the next stage of the research may be to define the processes and methods of architectural design of CSS for the general case of each type of system. Such processes should be compatible with [8] and focus on architecturally significant aspects and main architectural tasks of the corresponding types.

Research [47], devoted to the automated design of software architecture, as well as works similar to [48]; aimed at automating the design of control logic, constitute a promising direction in connection with the rapid development of artificial intelligence (AI) applications in the field of general automation. However, such studies focus mainly on details related to AI-specific aspects of systems (e. g., multi-agent frameworks), but lack an established underlying system design methodology that an AI system could follow.

The presence of such a methodology could significantly improve the results and their interoperability and integration, since the system would operate within standardized architectural processes with expected and standardized results, for example, in the form of corresponding architectural views [8, 10, 12].

4. Conclusions

1. A model of elements and levels of the CSS in automation has been developed, which form the aspects that should be covered by architectural design processes. This ensures standardization and unification of the CSS vision within the proposed paradigm.

2. A CSS classification in automation has been developed, based on fundamental aspects of the nature of systems. Its main difference from existing taxonomies is the focus on the differentiation of approaches to the design of the CSS architecture within different types of systems. The classification addresses the critical need for a more structured and differentiated approach to the design of the CSS architecture in automation. For each defined type of system, the main architectural aspects are described. The proposed classification lays the formal basis for adapting design processes to the CSS specifics in automation. It fills the gap in system-level architectural design, contributing to the establishment of standardized and differentiated methods of transition from requirements to the CSS architecture. This allows the formation of differentiated practical methods and processes for designing the CSS architecture, taking into account the fundamental aspects of the nature of the systems.

3. The developed taxonomy was validated by deductive mapping onto established industrial reference models – the Purdue automation pyramid and the IIRA reference architecture. The mapping results confirmed the full compatibility of the proposed CSS types with the functional levels and components of generally recognized industrial architecture models, which proves their relevance and compliance with established industry practice.

4. Some practical aspects of applying the proposed paradigm are considered using the example of a gas transportation process automation system at the compressor station level. The considered example demonstrated the application of the paradigm to solve practical problems of CSS designing for an automation system.

Conflict of interest

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

Financing

The research 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 and editing; ***Leonid Zamikhovskyi***: Supervision, Project administration, Writing – review and editing.

References

1. Williams, T. J. (1990). A Reference Model for Computer Integrated Manufacturing from the Viewpoint of Industrial Automation. *IFAC Proceedings Volumes, 23 (8),* 281–291. https://doi.org/10.1016/s1474-6670(17)51748-6
2. *IEC 62264-1:2013 Enterprise-control system integration – Part 1: Models and terminology* (2013). International Electrotechnical Commission. Available at: https://webstore.iec.ch/en/publication/6675
3. *The Industrial Internet Reference Architecture (IIRA), Version 1.10: An Industry IoT Consortium foundational document* (2022). Boston: Industry IoT Consortium. Available at: https://www.iiconsortium.org/wp-content/uploads/sites/2/2022/11/IIRA-v1.10.pdf Last accessed: 16.10.2025
4. Controllers. *Kubernetes.* Available at: https://kubernetes.io/docs/concepts/architecture/controller Last accessed: 16.10.2025
5. Kruchten, P. B. (1995). The 4+1 View Model of architecture. *IEEE Software, 12 (6),* 42–50. https://doi.org/10.1109/52.469759
6. Starke, G., Simons, M., Zörner, S., Müller, R. D., Losch, H. (2019). *arc42 by Example: Software architecture documentation in practice.* Birmingham: Packt Publishing.
7. *arc42.* arc42. Available at: https://arc42.org Last accessed: 16.10.2025
8. ISO/IEC/IEEE 42010:2022. Software, systems and enterprise – Architecture description (2022). *ISO/IEC/IEEE.* Available at: https://www.iso.org/standard/74393.html Last accessed: 16.10.2025
9. ISO/IEC/IEEE 42020:2019. Systems and software engineering – Architecture processes (2019). *ISO/IEC/IEEE.* Available at: https://www.iso.org/standard/68982.html Last accessed: 16.10.2025
10. Bass, L., Clements, P., Kazman, R. (2021). *Software Architecture in Practice.* Boston: Addison-Wesley Professional.
11. Cervantes, H., Kazman, R. (2016). *Designing Software Architectures: A Practical Approach.* Boston: Addison-Wesley Professional.
12. Rozanski, N., Woods, E. (2012). *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives.* Boston: Addison-Wesley.
13. Bauer, M., Boussard, M., Bui, N., Carrez, F., Siemens, C., Alube, J. et al. (2013). *Internet of Things – Architecture IoT-A Deliverable D1.5 – Final architectural reference model for the IoT v3.0.* Available at: https://www.researchgate.net/publication/272814818_Internet_of_Things_-_Architecture_IoT-A_Deliverable_D15_-_Final_architectural_reference_model_for_the_IoT_v30
14. Fornés-Leal, A., Lacalle, I., Palau, C. E., Szmeja, P., Ganzha, M., Paprzycki, M. et al. (2022). ASSIST-IoT: A Reference Architecture for Next Generation Internet of Things. *New Trends in Intelligent Software Methodologies, Tools and Techniques.* https://doi.org/10.3233/faia220243
15. Szmeja, P., Fornés-Leal, A., Lacalle, I., Palau, C. E., Ganzha, M., Pawłowski, W. et al. (2023). ASSIST-IoT: A Modular Implementation of a Reference Architecture for the Next Generation Internet of Things. *Electronics, 12 (4),* 854. https://doi.org/10.3390/electronics12040854
16. Megow, J. (2020). *Reference architecture models for Industry 4.0, smart manufacturing and IoT: An introduction.* Berlin: Begleitforschung PAiCE; iit – Institut für Innovation und Technik in der VDI / VDE Innovation + Technik GmbH.
17. ANSI/ISA-88.00.01-2010 Batch Control Part 1: Models and Terminology (2010). *International Society of Automation.* Available at: https://www.isa.org/products/isa-88-00-01-2010-batch-control-part-1-models Last accessed: 16.10.2025
18. AUTOSAR (AUTomotive Open System ARchitecture). *AUTOSAR Consortium.* Available at: https://www.autosar.org Last accessed: 16.10.2025
19. Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Boston: Addison-Wesley Professional.
20. 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
21. Forlingieri, M., Lamm, J. G., Roth, S., Walker, M. (2022). *Model-based system architecture.* Hoboken: Wiley. https://doi.org/10.1002/9781119746683
22. Eichmann, O. C., Lamm, J. G., Melzer, S., Weilkiens, T., God, R. (2024). Development of functional architectures for cyber-physical systems using interconnectable models. *Systems Engineering, 27 (6),* 993–1011. https://doi.org/10.1002/sys.21761
23. Griffor, E. R., Greer, C., Wollman, D. A., Burns, M. J. (2017). *Framework for Cyber-Physical Systems: Volume 1, Overview (NIST Special Publication 1500-201).* National Institute of Standards and Technology. Available at: https://doi.org/10.6028/NIST.SP.1500-201
24. NASA Systems Engineering Handbook (NASA/SP-2016-6105 Rev2) (2017). *National Aeronautics and Space Administration.* Available at: https://ntrs.nasa.gov/citations/20170001761 Last accessed: 16.10.2025
25. The DoDAF Architecture Framework Version 2.02 (2010). *The U.S. Department of Defense.* Available at: https://dodcio.defense.gov/Library/DoD-Architecture-Framework Last accessed: 16.10.2025
26. MOD Architecture Framework (2012). *Ministry of Defence.* Available at: https://www.gov.uk/guidance/mod-architecture-framework Last accessed: 16.10.2025
27. About the Unified Architecture Framework Specification Version 1.2 (2022). *Object Management Group.* Available at: https://www.omg.org/spec/UAF/1.2/About-UAF Last accessed: 16.10.2025
28. Plum, N. (2025). *TRAK Enterprise architecture framework.* Available at: https://sourceforge.net/projects/trak Last accessed: 16.10.2025
29. NATO Architecture Framework Version 4 (NAFv4) (2020). *NATO.* Available at: https://www.nato.int/content/dam/nato/webready/documents/publications-and-reports/NAFv4_2020.09.pdf Last accessed: 16.10.2025
30. *The TOGAF® Standard* (2022). The Open Group. Zaltbommel: Van Haren Publishing.
31. *Gesellschaft für Systems Engineering e.V.* (GfSE). System Architecture Framework. Available at: https://saf.gfse.org/ Last accessed: 16.10.2025
32. Dömel, A. (2025). *The Robot Architecture Framework: A systematic approach for describing the architecture of complex, autonomous robotic systems.* [Doctoral dissertation; University of Bremen].
33. Hatebur, D., Heisel, M. (2009). Deriving Software Architectures from Problem Descriptions. *Software Engineering,* 383–392.
34. Alebrahim, A., Hatebur, D., Heisel, M. (2011). A Method to Derive Software Architectures from Quality Requirements. *2011 18th Asia-Pacific Software Engineering Conference.* IEEE, 322–330. https://doi.org/10.1109/apsec.2011.29
35. Fitzgerald, J., Gamble, C., Larsen, P. G., Pierce, K., Woodcock, J. (2015). Cyber-Physical Systems Design: Formal Foundations, Methods and Integrated Tool Chains. *2015 IEEE/ACM 3rd FME Workshop on Formal Methods in Software Engineering.* Florence 40–46. https://doi.org/10.1109/formalise.2015.14
36. Rama, A., Reddy, M., Govindarajulu, P., Naidu, M. M. (2007). A Process Model for Software Architecture. *International Journal of Computer Science and Network Security, 7 (4),* 272–280.
37. Liu, L. (2018). The Process to Design an Automation System. *Journal of Physics: Conference Series, 1087,* 042001. https://doi.org/10.1088/1742-6596/1087/4/042001
38. Cantor, M. (2003). Rational Unified Process for Systems Engineering: Part 1. *Journal of Systems Architecture – JSA.*

ISSN-L 2664-9969; E-ISSN 2706-5448

39. Sas, C., Capiluppi, A. (2022). Antipatterns in software classification taxonomies. *Journal of Systems and Software, 190,* 111343. https://doi.org/10.1016/j.jss.2022.111343
40. Britto, R., Wohlin, C., Mendes, E. (2016). An extended global software engineering taxonomy. *Journal of Software Engineering Research and Development, 4 (1).* https://doi.org/10.1186/s40411-016-0029-2
41. Usman, M., Britto, R., Börstler, J., Mendes, E. (2017). Taxonomies in software engineering: A Systematic mapping study and a revised taxonomy development method. *Information and Software Technology, 85,* 43–59. https://doi.org/10.1016/j.infsof.2017.01.006
42. Guenther, J., Falk, I., Cole, M. J. (2023). Theory building from qualitative evaluation. *Evaluation, 29 (4),* 410–427. https://doi.org/10.1177/13563890231196603
43. Lukka, K. (2003). *Case study research in logistics.* Turku: Turku School of Economics and Business Administration, 83–101.
44. Kwasnik, B. (1999). The role of classification in knowledge representation and discovery. *Library Trends, 48 (1).*
45. Mohammed, S. S. (2024). Deductive and inductive research. *Introduction to Research Methods.* Japan Bilingual Publishing Co., 203.
46. Iovino, M., Förster, J., Falco, P., Chung, J., Siegwart, R., Smith, C. (2024). *Comparison between Behavior Trees and Finite State Machines.* arXiv:2405.16137. https://doi.org/10.48550/arXiv.2405.16137
47. 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
48. Guo, X., Keivan, D., Syed, U., Qin, L., Zhang, H., Dullerud, G. et al. (2024). *ControlAgent: Automating Control System Design via Novel Integration of LLM Agents and Domain Expertise.* arXiv:2410.19811. https://doi.org/10.48550/arXiv.2410.19811

Ihor Polataiko, *PhD Student, Department of Information and Telecommunication Technology and Systems, Ivano-Frankivsk National Technical University of Oil and Gas, Ivano-Frankivsk, Ukraine, ORCID: https://orcid.org/0000-0002-9111-0162*

✉***Leonid Zamikhovskyi,*** *Doctor of Technical Sciences, Professor, Head of Department of Information and Telecommunication Technology and Systems, Ivano-Frankivsk National Technical University of Oil and Gas, Ivano-Frankivsk, Ukraine, e-mail: leozam@ukr.net, ORCID: https://orcid.org/0000-0002-6374-8580*

✉*Corresponding author*