Computer Science and Information Systems 00(0):0000–0000 https://doi.org/10.2298/CSIS123456789X

Applying a Hybrid Deployment Strategy for Software Updates to the Manufacturing Execution System Layer*

Petar **Rajković** [1], Dejan **Aleksić** [2], Dragan **Janković** [1], Aleksandar **Milenković** [1], and Anđelija **Đorđević** [1]

[1] University of Niš, Faculty of Electronic Engineering,
Aleksandra Medvedeva 4, 18104 Niš, Serbia
petar.rajkovic@elfak.ni.ac.rs
dragan.jankovic@elfak.ni.ac.rs
aleksandar.milenkovic@elfak.ni.ac.rs
andjelija.djordjevic@elfak.ni.ac.rs

[2] University of Niš, Faculty of Science and Mathematics, Department of Physics,
Višegradska 33, PO BOX 224, 18106 Niš Serbia
alexa@pmf.ni.ac.rs

Abstract: Complex industrial systems consist of many heterogeneous devices running different hardware and software in a connected, layer-organized environment. Since all these software instances must be updated occasionally, and since they could affect the layers under and above, the definition of deployment strategies that will reduce downtime is necessary. In previous work, we focused on identifying common problems in software update processes and concentrated on the most effective update strategies running at the lowest (Internet of Things – IoT) and highest (Enterprise Resource Planning – ERP) levels. The result was a set of recommendations and strategies that should help minimize network utilization and processing resources and make the process as energy-efficient as possible. After that, the core effort of the research is shifted toward the Manufacturing Execution System (MES) layer – the layer that brings the higher complexity, both in terms of connectivity and software complexity. Following the actual Industry 4.0 paradigm, the software in the MES layer becomes even more critical since it is expected to integrate a whole new set of responsibilities previously belonging to various levels or external solutions. To facilitate further requests, deployment strategies are reevaluated and enriched with innovative approaches such as A/B testing and the separate update service. This paper shows the possible further development of the hybrid software deployment system when applied to the multiconnected levels, such as the MES. The adaptation shows positive results regarding the network load distribution and significant effort reduction in cases when a rollback is needed.

Keywords: Industrial software, Manufacturing Execution Systems, Software deployment strategy, Resource Awareness, Industry 4.0

1. Introduction and Background

Complex industrial systems represent an exciting conglomerate of various technical solutions. Knowledge from different engineering sciences is needed to solve the challenges from process modeling, signal collecting, and processing through plant layout

* This manuscript is an extended version of the paper published in the proceedings of the CERCIRAS 2021 workshop

design to raw materials and finished goods transportation, distribution, and storage. Nowadays, all these aspects are supported by adequate software. Due to significant differences between diverse aspects of the organization in the industrial facility, the complete structure is divided into standardized layers. The standard ISA95 [1] defines in detail how to split the industrial system organization and what the responsibility of each layer is. Following the standard, the information technology (IT) subsystem in the industrial environment consists of many heterogeneous devices running different pieces of software in a connected and layer-organized environment [2] (Figure 1). Starting from the sensor and actuator layer connected with microcontrollers (in our work, we will reference it as the IoT layer) [3], through the Edge layer [4][5], via SCADA [6] and manufacturing execution systems (MES) [7] to enterprise resource planning (ERP) [8], all pieces of equipment run the software that needs to be updated occasionally.

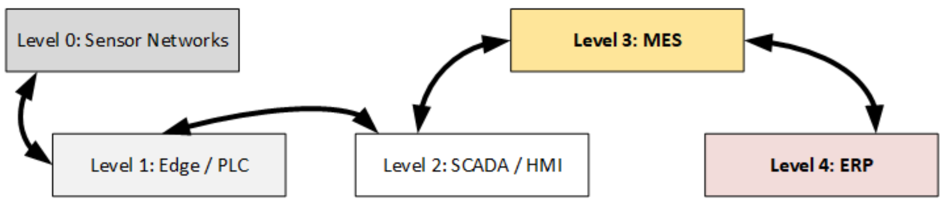

Figure 1 ISA95 levels according to the industrial standard

Software update, as a process, is an activity that is considered highly problematic in the industrial environment. From the point of view of the process engineer, it should either happen never or only in predefined maintenance slots. It came from the experience with the previous deployment methods, where intensive planning must be done, and some areas of the industrial facility will be disconnected for a more extended period. If deployment needs to be reverted or reconfigured, the problem will be even more significant.

In our previous work, we have been focused on the deployment of the software in the lowest (the IoT level [9]) and the highest (the ERP level [10]) levels of the system. From the connectivity point of view, these two layers have been of minor operation complexity since they maintain connectivity only to nearby levels. IoT nodes are usually connected only to Edge computers, while the ERP communicates with MES. The main difference between them is the requirements regarding the volume of the needed resources. In the IoT area, resource shortages are faced in every aspect of work.

This paper represents the direct extension of the work published in the CERCIRAS 2021 workshop. The definition of the testing environment and the default deployment strategies used for IoT nodes were the starting point and thus included in this work. This paper describes the usage of the concepts of the software update approach for a single node with limited storage space and expands then further on the application at the MES level.

As mentioned, any deployment strategy must consider energy consumption, storage space, and processing power. Such an environment requires carefully defined deployment methods and, even more importantly, backup and restore strategies in case of unsuccessful deployments. The next step was to generalize the approach described for IoT nodes and apply it to the ERP layer [10]. In that sense, this paper could be seen as the further continuation of the work we described in [10]. Since ERP layer software was built with more advanced software tools, it offers more possibilities for defining the update strategy. In that sense, the different software deployment methods were analyzed, and a set of routines that should improve deployment scenarios was proposed and evaluated. Deployment strategies, defined in [10], were the next step in our deployment and were thus used as another starting point in our work. The explained use of advanced strategies was another building block to define routines for the MES software. The software at the ERP level shares the complexity, technology stack, and implementation approaches with MES, which was of significant value for this work.

The next goal is to apply the proposed deployment routines to the MES layer. MES is the layer that brings the higher complexity into the design, both in terms of connectivity and software complexity. Following the actual Industry 4.0 paradigm, the software in the MES layer becomes even more important since it is expected to integrate a whole new set of responsibilities previously belonging to various levels or external solutions. Deployment strategies are reevaluated and enriched with innovative approaches to facilitate further requests. For example, the MES server could be connected to SCADA on one side and to the ERP on another. In contrast, the clients could be connected directly to measuring devices in Edge or IoT to register and visualize different measurements.

In this situation, downtime during the update needs to be evaluated through multiple sides to ensure proper reconnection and operation continuation from various sides. Also, one must remember that with new requirements under Industry 4.0, the MES software should offer new functionalities that often come without full specification and where multiple versions must be simultaneously evaluated. This paper shows the results of the research that had the following research tasks:

- Test and adapt the deployment strategies suggested in [9] and [10] and try to use them both for server and client components of the MES level.
- Focus to reduce network load on the MES server side.
- Organize deployment to stop the erroneous deployment as soon as possible.
- Integrate the process of the practical test of new functionalities when the customer must choose between multiple solutions.

This research relies on our previous work, primarily described in [9] and [10] and represents its continuation and improvement.

2. Background – Industry 4.0 Paradigm and Existing MES Systems

MES and Industry 4.0 are critical components of the modern manufacturing landscape. They aim to integrate technology and data to optimize production processes, improve efficiency, and drive innovation in many new ways. Industry 4.0, the Fourth Industrial Revolution, represents a radical shift in manufacturing practices.

It involves the digitization and the use of advanced scheduling and execution algorithms in manufacturing processes, moving away from mass production towards customized production that caters to individual customer requirements (Figure 2). This means that a portion of the planning and scheduling will be moved from ERP to the MES level. Next, MES plays a crucial role in Industry 4.0 by providing real-time visibility, control, and intelligence across the entire product life cycle value chain. It should allow for seamless communication, analysis, and data utilization to drive intelligent actions in the physical world. This means that the connection from MES will not only go to the SCADA layer but also directly to Edge, IoT, and sensor networks in some cases.

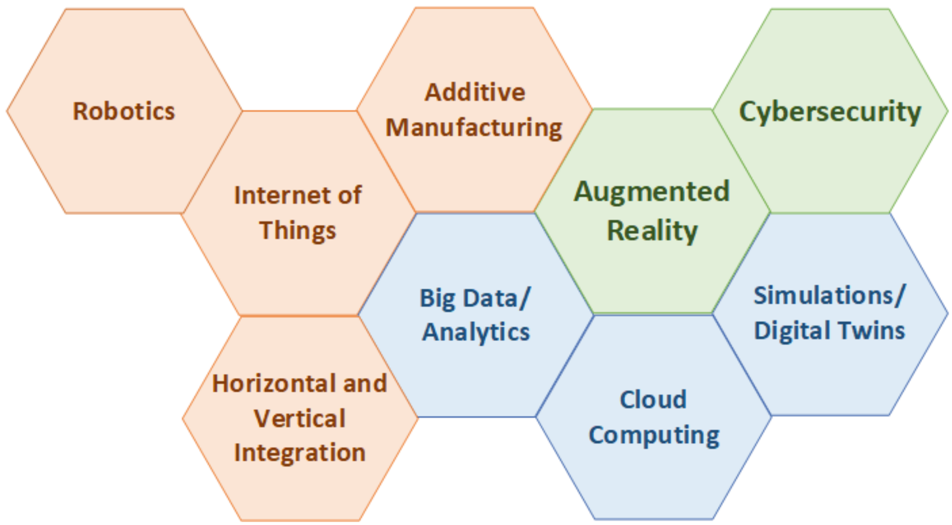

Figure 2 Main elements of Industry 4.0

With full rights, the new generation of MES and Industry 4.0 is expected to enable organizations to harness the power of digital technologies and intelligent, connected systems to revolutionize their manufacturing processes. They would allow organizations to optimize production processes, improve efficiency, and drive innovation by leveraging robotics, analytics, artificial intelligence, nanotechnology, the Internet of Things, and cloud computing. These technologies enable organizations to automate tasks, analyze data for actionable insights, and connect various parts of the production process for seamless coordination and optimization. At the same time, it is, more than ever, expected that software runs with the lowest possible downtime and that all activities run as smoothly as possible.

While previous work focused on single-connection levels, like IoT and ERP, this paper will evaluate the application and extension of the existing set of recommendations for software at the MES level. MES-level software is significantly different from those running in IoT nodes but closer to ERP systems. First, MES systems usually follow service-oriented architecture (SOA) with various clients.

These software instances run on servers or in the workstation, with significant processing power and memory storage compared to IoT nodes. It looks like the MES systems run in an environment where resources are not the problem, but it is not quite like that. Depending on the configuration and the set of required operations, MES clients could weigh up to a few hundred megabytes. It depends, of course, on the implementation technology and other dependencies. Still, if they are implemented as the thick client, the usual user requirement, their update process could employ significant network traffic.

Compared to ERP software, MES runs fewer complex algorithms, but it connects more extended software and services and runs in significantly more numbers and variants of clients. It is also essential to state that with the current technology demands fueled by the Industry 4.0 initiative, the importance of the MES system rose. Nowadays, MES is often required to provide many functionalities native to other systems. The MES should now support continuing different reporting, overall equipment effectiveness tracking (OEE), Andon boards, deeper integration with ERP systems and SCADAs, and ending various synchronizations with warehouse, packaging, and other systems.

3. Related Work

The existing literature offers various deployment strategies, evaluations, and recommendations. In most cases, the existing research covers software that runs in layers such as MES and ERP. Besides, it has been constructive for our current scope of research, but it was a bit misleading when one tends to define the close-to-universal strategies and approaches. These higher layers deal with clients transferring significant data and executing numerous transactions. When defining development strategies for lower levels, the standard approaches from the literature are not directly implementable due to their unique limitations.

The most critical points for resource management at lower levels are storage capacity and data traffic through connecting networks. The overall effect is not the same on all layers [15]. MES runs in a shop floor environment on devices with processing power similar to standard computers.

The storage space is not a critical requirement for devices running MES or ERP software, but they are usually connected to their server using the wireless network. The wireless networks in the industrial environment could experience different disruptions because of operating nearby machines generating high-frequency harmonics as well as other security threats [16]. Data package verification and consistency are critical for MES and ERP client nodes. When deploying a new version of the software to some device, an update package, which is of significantly higher volume than usual data traffic, needs to be distributed via a network, verified, and stored on the destination device, and the old version needs to be backup in case of rollback [17] [18]Next, the Edge layer's primary mission is to collect all the data from sensor networks and pass it to the MES. In this case, the proper buffer implementation ensures smooth software upgrades.

All the mentioned layers are highly heterogeneous, with different pieces of hardware running the software instances with diverse categories of software. Overall, in the complete industrial system, the type of used devices, their number, and the amount of transferred data (per device) could be between 1kB and 1GB. To make the complete process more demanding, the devices sometimes do not have enough memory to store two software versions; thus, they would require backup in a different location. This leads to the situation that sometimes it is nearly impossible to have an upgrade with no, or at least with very low, downtime [19].

As with every process, a software update could fail for numerous reasons. In that case, a complete deployment approach or deployment system needs to provide the possibility to roll back to the previous version [20]. The rollback will then take more resources and make the situation even worse, so we need to ensure that system governance successfully goes through the process [21].

To reduce the impact of the mentioned problems and potential system downtime, we aimed to define a more general approach that could be configured to use the combination of blue-green [22] and canary deployment [23] styles in combination with both shared and local backups [24]. This approach looks promising at the IoT level. The approach was tested in a production environment, and the results were published in [9] and [10].

Working on a general set of recommendations [9] [10], we conclude that regardless of the type of software and the operating level, the blue/green approach could be effectively used at any node (Table 1). New components used to build IoT nodes increased memory and processing power, so keeping two versions simultaneously would probably not be a problem. The blue/green approach, per se, could be improved with additional techniques such as buffers and backup nodes [9] [10]. For example, at all levels, a blue/green approach supported by the dark mode with feature flags could be used for server node deployment. This will give flexibility and security; newly developed features could be gradually turned on until the complete server update is reached. For clients, blue/green is the primary choice, which could be enriched with buffers and feature flags if the resource pool and used implementation technology allow.

Table 1 Elements of the deployment strategy used in various levels (BG – blue/green, DF – dark mode with feature flags, CS – canary with sentinel node, CB – canary with backup node, IB – intermediate buffer, (XX) - optionally) (as suggested in [10])

Level	Server	Client network	Single Client
Levels 0 and 1 (sensor network)	BG + (DF)	CB + IB	(BG) + IB
Level 2 (IoT nodes and Edge computers)	BG + DF	CB / CS + IB	BG + IB
Level 3 (MES)	BG + DF	CS / CB	BG + DF + IB
Level 4 (ERP)	BG + DF	CS	BG + DF + IB

The level of downtime reduction is significantly reduced in this scenario, compared to standard approaches such as recreate deployment and rolling deployment [27]. In the recreate deployment, the previous version of the software is shut down, and the new one starts after the old one has been stopped. Rolling deployment is applicable for complex systems with multiple servers. It is based on the recreate deployment but applies to various services. The downtime is exceptionally low, but the length of an upgrade process depends on the number of servers/nodes in an array, and it could take considerable time. The proposed deployment strategy will improve overall deployment time even more in the case of the rolling strategy since the blue/green switch could be done in the close period; there is no need to wait until all the servers are updated in the sequence.

The new request that does not fit into the proposed framework is to have the possibility to support simultaneous evaluation of different versions of functionality. Besides, it could be done through the feature flags, but it will eventually require more consolidation and stabilization work. The A/B testing deployment approach is included to address such requests. This approach is used on the client side to improve the development and test phase and provide the possibility for limited testing in the production environment. This approach aims to offer different functionalities to some clients and then evaluate the user's reaction and acceptance. The update is usually done in a few groups of varying sentinel nodes.

Recreating and rolling deployment are crucial concepts in software development and operations at the MES level [28]. Recreating refers to rebuilding a software system or environment from scratch, often to resolve issues or update components. On the other hand, rolling deployment involves deploying new software versions in a gradual and controlled manner, allowing for continuous delivery and minimizing downtime. Integration with other systems and services traditionally occurs at the end of a development life cycle, but rapidly developed applications are integrated almost immediately. Testing occurs during every iteration, enabling stakeholders to quickly identify and discuss errors, code vulnerabilities, or complications and immediately resolve them without impacting the development progress. As stated in [29], "integration with other systems and services traditionally occurs at the end of a development life cycle, but rapidly developed applications are integrated almost immediately". This iterative approach to development and testing is a crucial aspect of recreating and rolling deployment methodologies.

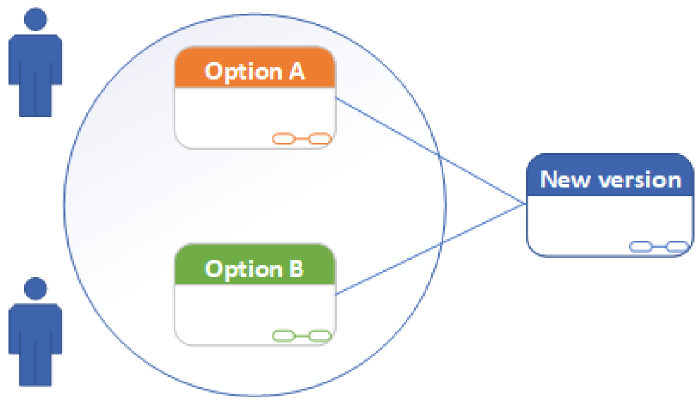

Figure 3: A/B testing deployment

The use of A/B deployment (**Figure 3**) strategy has become increasingly popular in various fields, including technology, marketing, and product development. This strategy involves testing two different versions, A and B, of a product or service to see which performs better [25]. The first source highlights the importance of product or service innovations in engaging customers and improving performance. It suggests that the market development strategy, which focuses on pursuing additional market segments or geographical regions, can increase sales but also comes with more risk. The second source discusses different methods for gaining market share, including product development and market development [26]. In the Industry 4.0 era, the use of A/B testing deployment is a comparable advantage within the installations of MES. The installation supports A/B testing and easy transition to the new version, considered more advanced and customizable. The A/B testing is widely popular with deployment based on container technologies, such as Kubernetes [30], since they involve end-users in decision-making over the new version of the software.

4. Testing Environment

As it has been known, the update process comes with the risk of diverse potential failures that could leave parts of the system unresponsive, running with unpredictable behavior, or emitting erroneous data. For this reason, the update process must be executed in a highly controllable environment that allows easy and efficient rollbacks in case a flawed deployment is detected. As stated before, all software components in the industrial system are usually organized in layers. Layers exchange data with each other using different software protocols. The mentioned facts make the overall software update process a bit more complex than within a standard information system environment, and every error could lead to serious domino effects [11] [12]. Updating software in one layer could impact the targeted device and other devices in the same and different layers. For example, the update performed on the device running at the MES level could affect software instances running in other layers.

The additional limitation point is the expectation for the highest possible performance and the requirement that software run using as few resources as possible. The complete system must have a high degree of resource awareness, and both storage space and network bandwidth usage must be carefully planned during the update process in order not to reduce the execution of the running components significantly [13][14]. For this reason, the resolute digital twin is used for testing.

The digital twin (**Figure 4**) is created partly in the laboratory environment and partly in the cloud to simulate different connectivity scenarios and have an overview of worse-case scenarios regarding latency and execution. The emulated hardware in a digital twin is set to the lowest acceptable resource level, which should simulate worse execution conditions than those in the production environment. The testing digital twin is introduced during the implementation of the one-of-the-kind production system [32]. As the demo factory, the plant producing doors and windows is set.

Such a production facility is used for demonstrating since it combines all diverse kinds of production and needs multiple sensors and precise mechanical units to be integrated. On the MES, the level needs several diverse types of clients and services. The digital twin environment used for testing was described in [10] and improved to support more complex environments. Previous research focused either on IoT nodes, which were entirely configured in the local network, or on ERP clients, which were all the same and ran only in the cloud.

The IoT level in the digital twin consists of 100 nodes connected to simulated instances of sensors and actuators. Each IoT contains a different number of sensors and actuators, which count within the node and could be anything between a few and 1,000. The count of 100 gives enough flexibility and complexity to perform testing in the development phase. The digital twin, an exact mirror replica of the industrial facility environment, could be created for the production phase. In the default model, following the ISA95 model, sensors are connected to IoT nodes. Especially after the Industry 4.0 concept brought new requirements for MES systems, a direct connection between MES clients and measuring sensors could be established, too.

Sensors within one IoT node could be different, and all could run various software. Sensors could be active either constantly or just for predefined periods. They could collect very heterogeneous data with varying sample rates during their operation time. All these facts make the IoT level very dynamic from the operational point of view. They could increase the probability that the complete node went out of a stable state in case of problematic deployments.

The available memory space is usually between 1 and 5 MB per device, which is enough for the necessary software. The nodes in the IoT layer are connected using various methods, ranging from cable network connectors to LoRaWAN, which creates an inconsistent environment in terms of connection speed and quality. The most complex situation is with LoRa-connected devices since their bandwidth could be only 10-20 kbps.

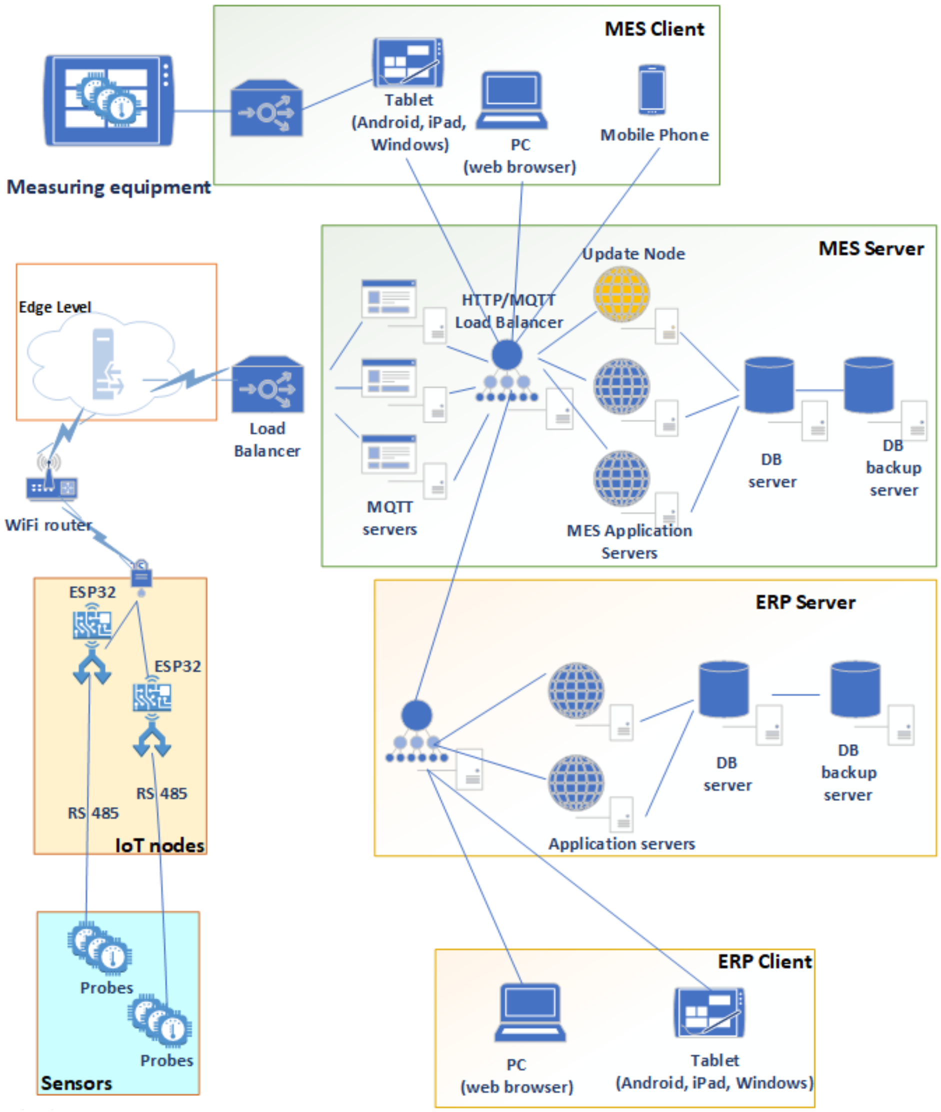

Figure 4: The composition of the examined system containing all levels of the ISA95 model

IoT node layers are further connected to Edge computers or Edge nodes. Edge nodes communicate between the shop floor and hazardous areas on one side and higher levels, such as MES and enterprise resource planning (ERP), on the other. Edge nodes are devices based on Raspberry Pi or similar base sets and are usually connected by a Wireless network with an effective network speed of around 20 Mbps. Their space requirements are around 30 MB per node. There were 10 of these nodes in our test environment. To support testing, the mesh of 10 Edge computers is modeled in digital twin. Each of them is set to collect data from 10 IoT nodes.

From the resource awareness point of view, software components on MES and ERP levels are easier to manage. They run on desktop/laptop computers with enough processing power, disk space, and bandwidth, but resource planning is inevitable even with them. In our test environment, we used 200 MES clients connected to 4 MES servers (two load-balancing and two redundant, with the possibility to change the configuration) and 30 ERP clients connected to the Microsoft Dynamics server. All the clients at this level are a few hundred megabytes in volume and are located under a gigabyte network.

Table 2 Different MES clients and their functionalities

MES Client Type	Connection within MES Level	Connection to other levels/services
Administrative	Server	ERP
Operation	Buffer, Server	Edge
Configuration	Server	ERP, External cloud services
Management	Server, Operation clients	Reporting
Measurement	Server	IoT, Edge

Regarding MES clients, a few diverse types are supported by performing different connectivity and execution actions (Table 2). Administrative clients perform operations related to the ERP level. They are responsible for synchronizing operations definitions, catalog data, material definitions, and other master data needed to exchange data between MES and ERP properly.

The operation client has a connection to the execution buffer on the MES side and to the Edge level. The execution buffer is an optional implementation that allows clients to continue to run when the server is offline. It contains a buffer filled with tasks that must be executed in the workstation and that collects data generated during the production process. Once the connection is reestablished, the data flow will resume, and the server-side upgrade will have the lowest possible impact on the clients.

Configuration client is described in detail in [32]. It is used to define new products and eventually upload these data to cloud services and ERP. Management client acts as a synchronization node between ERP and operation clients. It is responsible for downloading production orders from ERP and uploading collected status change data measurements, etc. Ultimately, the measurement client will provide the interface for material registration and integration with IoT nodes such as sensors and other measurement devices.

5. Transition of Deployment Strategy from IoT to MES node

The software update process for IoT nodes and sensor/actuator devices running in a production environment is considered particularly sensitive. In industrial automation, sensors and actuators emerge as fundamental components that underpin efficient, safe, and precise operations. These unassuming devices are pivotal in monitoring, controlling, and optimizing various processes across diverse industries. The update of such small components requires detailed planning before an update. Thorough planning is needed because they are, on the one hand, tiny both in size and capacity, and on the other hand, they are running in a hazardous environment where the only possible connection is relatively slow LoRa networks with no wiring possible and limited physical access, (**Figure 5**). If some physical intervention is needed, the stoppage of the complete industrial process is often a requirement.

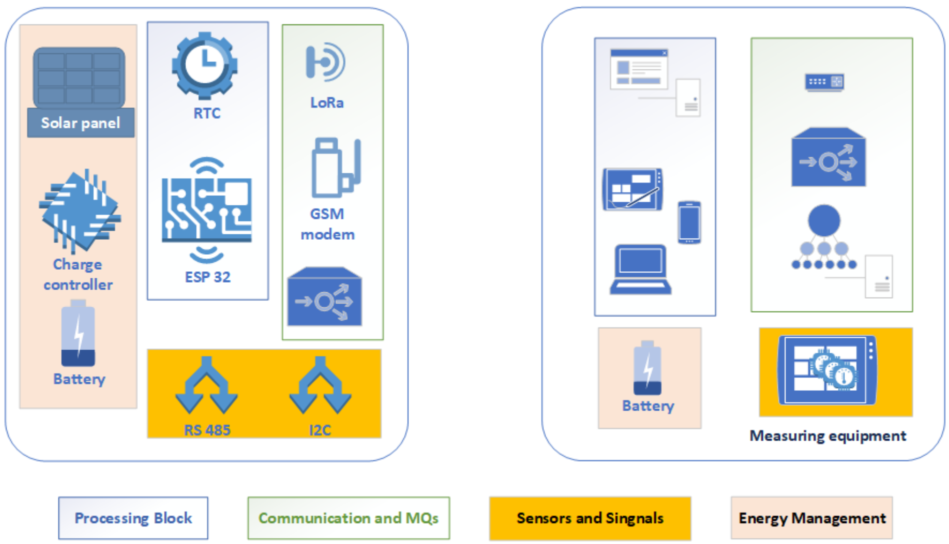

Figure 5: Comparison of building blocks of IOT (left) and MES (right) client nodes

Besides the slow network, the low-performance hardware is one additional potential problem. This fact could result in an unacceptable long update process, which could move the targeted device off the system for an extended period. The last, but not the least important, is the energy consumption problem. Software updates are an activity that requires significantly more energy than regular data collection and data transmission processes. Thus, this process must be planned for when the battery is charged to the highest possible level and when the eventual rollback will not drain the battery.

At first sight, it looks like there are no common issues or problems between IoT and MES clients. MES clients have fewer limitations, especially in processing power and storage capacity. Stating that one can assume that any kind of deployment strategy is convenient for MES clients. It could be said this from a strictly technical perspective, but when including different business requirements, it turned out that deployment at the MES level must be carefully designed, too. Furthermore, the main building blocks for both clients are similar (**Figure 5**). In both client types, regardless of different implementation technologies, Communication, data collection, and the processing block could suffer from the same problem. The problems with the low energy level are related to IoT, while the MES clients could suffer from synchronization and compatibility problems.

Noticing this, we realize that the deployment strategy defined for IoT nodes could apply to MES clients and be enriched with the experience through the project of ERP client deployment. but the concepts used for IoT nodes could be applied to the MES nodes. As it has been presented in Table 1. Blue/green deployment could be used if the destination node has enough storage space. The difference would be in the specific implementation technology, but the concept will remain the same. Additionally, an intermediate buffer, defined at the IoT node level, could be safely applied to the MES level. The MES nodes implementation is based on the concept from the IoT level and then enriched with additional features that will bring even further benefits to the MES level.

Traditionally, the MES nodes usually used some of the classic deployment methods—recreate or rolling deployments. Such an approach has been acceptable in recent years. Still, due to the manufacturing shift towards Industry 4.0, users started looking at the re-installation process connected with downtime as a problem. In the case of rolling-like deployment, the issue relates to a long waiting period until the new version becomes fully available.

Furthermore, such an approach would require an IT assistant in the facility, ready to help, run an installer, or perform some similar support activity. Since this was not acceptable anymore, we aimed for an approach already applied in IoT nodes and for its transition to MES-level software.

5.1. Software Update Approach for IoT and MES Nodes

Looking at the single IoT node, our choice for a software update is a semaphore-based green/blue approach (**Figure 6**). This approach is possible with devices storing at least two software versions simultaneously. In this case, the critical points are typically low bandwidth and possibly low battery levels. The approaches to solving these two problems are elaborated further in [34].

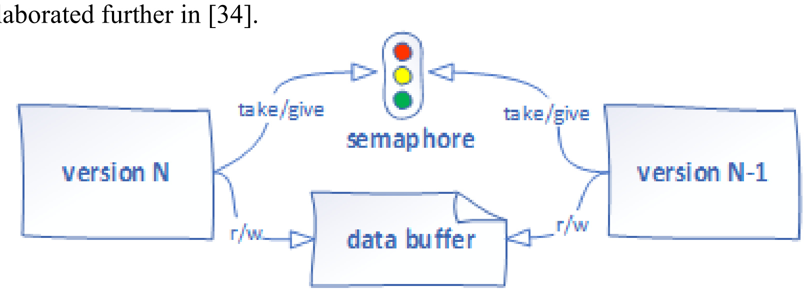

Figure 6: Semaphore-based blue-green deployment strategy used for IoT nodes [10]

The problems with applying such an approach at the MES level resemble the IoT level. First, data storage limitation is not, per se, the main issue, but the device could run into such a problem when the access rights for the installer are not managed correctly. The issues with access rights are not present in the IoT node since the vendor is responsible for hardware and software. At the MES level, the software is installed, in most cases, on the customer's equipment, for which the IT security and management team is responsible for maintenance.

As mentioned, the problem with low space could appear at the MES level if the installer has no delete rights for older versions. Since the MES clients could come with a few hundred megabytes of installed software and generate large log files, the issue with the space could arise if the delete and backup processes are not managed correctly.

Next, the installation could also create bandwidth problems if not appropriately managed. For example, in a factory with 200 workstations, each would require an MES client installed. In some cases, more MES clients could be launched on the same machine. At least 200 clients will require an update when an updated version is detected. If distributed from a single spot, as often chosen, the update process could easily make a bottleneck in the network. Furthermore, the MES client will maintain a connection to more layers in the ISA95 structure, which could cause further synchronization problems. For comparison, nodes at the ERP level, closely elaborated in [10], does not have connections to another system, which makes them much easier to handle.

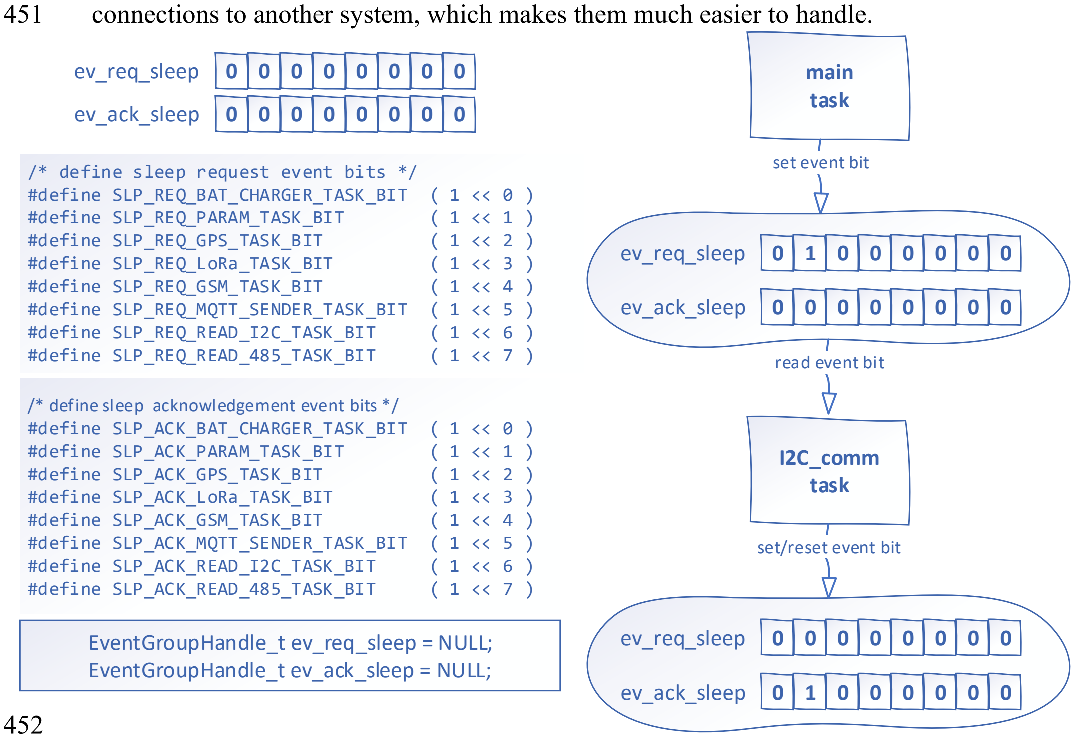

Figure 7: Software update sequence with the sleeping sequence

Coming back from IoT nodes, the base for the deployment approach is a blue/green strategy. This is the backbone of our update system. It is easy to be implemented in any technology. The main idea behind the blue/green strategy is to ensure that the target device always keeps at least two software versions – actual running (version N-1) and previously verified (version N-2). To reduce the data loss during the switchover, the node setup is completed by a message queue. Message queue collects data from sensors, and data are removed from the queue after being processed. The queue could be implemented as an independent entity to continue collecting data during the switchover.

The update process starts by replacing version N-2 with the new version N. At that moment, version N-1 is still active, and the device runs uninterrupted. During that period, the device experiences higher-than-average network traffic and battery use. Once version N – 2 is deleted and version N is uploaded and verified, the switchover could start. The device begins operating version N, but its communication points remain inactive. When version N is fully up and running, the semaphore opens communication to version N and stops version N-1.

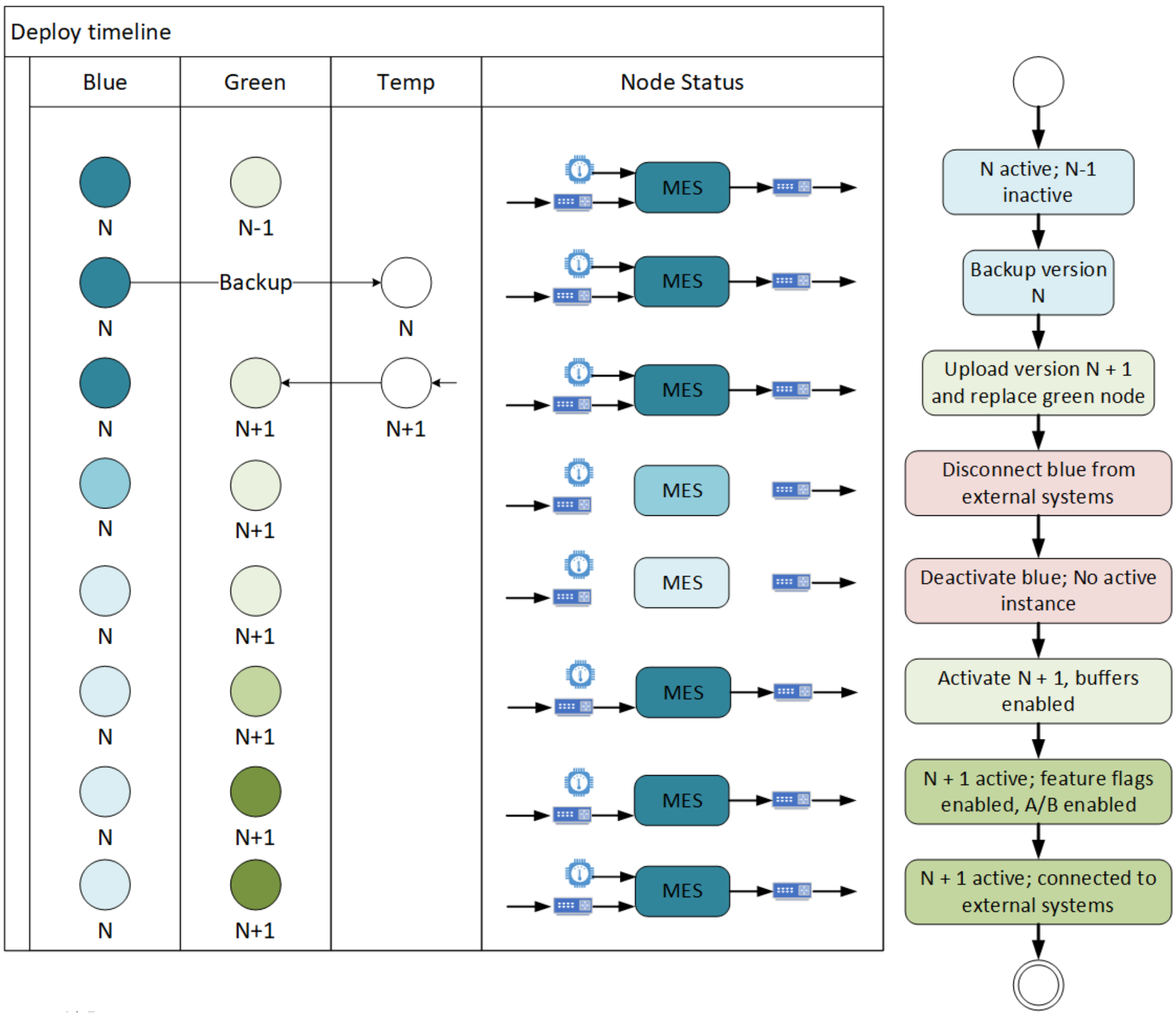

Figure 8 Software update sequence for MES client (expanded from [10])

In that case, there is almost no operation downtime, and the complete update process is seamless for the customer (**Figure 7**). In a well-orchestrated process, data loss during the switchover can be effectively mitigated. In the worst-case scenario, only signals received during the switchover—typically lasting several seconds—may be lost and left unprocessed. The switchover is seamlessly executed for IoT nodes by transitioning to sleep mode. Since sleep modes are an integral part of processing, facilitated by a dedicated core, transitioning to and from sleep mode is considered a native operation for IoT nodes.

In many cases, this approach will also be fully applicable to MES nodes. Unfortunately, not always. Two central problems appeared here with MES clients. First, as mentioned before, the older version (N – 1) will not be deleted in case of a lack of privilege. If not managed properly, this will cause a problem with the space on the destination node. The next problem is the switchover phase. MES clients are much larger pieces of software with a powerful GUI that maintains integration with different services on the MES level and even to different Edge, SCADA, and IoT devices. The proper switchover would require not only the replacement of the client version but also the reestablishing of connection to other connected instances (Figure 8). This makes the buffering system even more important here than at other levels.

Blue/green is not a favorable solution; it is only for successful updates. It proves its value when the update fails. In that case, blue/green offers an effortless way to switch back to the previous (valid and proven) version N—1. Furthermore, such a rollback will not require additional data traffic, which is desirable in any scenario and level. Once the error is solved, version N could be replaced with the next update.

The blue/green setup supports both full and partial version updates. In case of a partial version update, the new version will be generated when the copy of N-1 gets merged with new libraries and configuration files. The partial approach is faster and brings a lower network load. It is helpful for MES-level clients, but it is even more suitable for devices with more processing power on the IoT level. The easiest way to spot them at the IoT level is to check if they use GSM modems and LoRa adapters. In brief, partial deployment is more efficient for more complex software components.

This approach will not solve every deployment problem. In some cases, it could be inefficient or even useless. In case of a partial update, it could happen that the deployment package did not come with all necessary dependencies. Then, the update will fail, leading to additional data transfer and new version creation.

Next, the new version might be larger than the available space, even after deleting version N-1. In this situation, the blue/green approach cannot give positive results, and the update will fail. This would lead to the request for additional intervention and, in the best case, reducing the deployment to recreation mode.

Since the software is connected to services and other running instances on various levels, the interface between them might change from time to time. Or even buffer service needs to be updated. If this happens, blue/green will not help or solve the problem. Such updates then need to be implemented during planned downtime and meticulously organized to follow all necessary steps in the required order.

The last but not the least essential problem is when the device runs out of power during the update process. It could happen to any device, but those running on battery are more prone to this problem. The mentioned problem is not typical for MES nodes. They are connected to standard LAN/WLAN or Profibus network and are usually connected to the continuous power supply. If they lose the power during the update, they will continue to run the N version after the restart. Also, if the MES client is installed in a battery-running device, such as a tablet or laptop, their operation system will be configured to run updates only if the device is connected to the power grid.

As the clients run in more powerful nodes and more complex environments, their update process could be enriched with more proficient methods. The methods are feature flags, dark mode, or A/B testing, which will offer an easy transition to new functionality. The new version will be the same as the previous one upon the switchover, and then new functionalities could be gradually enabled. The end user would increasingly receive new features in this way. In case of a problem, the features could be quickly disabled remotely. Also, new versions of features could be assigned to specific clients to evaluate, following the A/B testing strategy.

5.2. Software Update Approach for Devices with Limited Storage Space

To address this challenge, an additional device of the same type, preferably with a larger storage capacity, is introduced. This backup node is a repository for storing backup versions of the currently running software. In scenarios where the Internet of Things (IoT) layer comprises multiple similar or identical nodes, adding an extra device is not perceived as a drawback but as a justifiable minimal cost.

The same approach applies to Manufacturing Execution System (MES) clients. However, the key distinction lies in the role assigned to the chosen node. In the MES environment, the selected node assumes the mantle of a leading or sentinel client responsible for distributing update packages within its designated group. Utilizing backup nodes at the MES client level is also feasible, especially in cases where stringent IT security protocols prohibit the retention of old software versions due to company policies.

The deployment process commences by transferring the new version (version N) to the backup or sentinel node. Once this operation is completed, the backup node disseminates version N to all devices running the same software. Notably, this approach slightly extends overall downtime, as the target node must first halt the previous version (N – 1), acquire the new version, and subsequently initiate version N. Conversely, no discernible difference in overall downtime occurs when the backup node acts as a sentinel.

An inherent drawback of this approach pertains to increased data traffic requirements. However, this traffic is confined solely to communication between the sentinel or backup node and the clients within its designated group. An additional advantage emerges during potential rollback scenarios. After uploading version N to the backup node, deployment to sensor nodes occurs sequentially. The process begins with the sentinel device (borrowing from the canary deployment concept), where comprehensive validation under production conditions occurs. If the new version proves valid, subsequent nodes receive the update. Conversely, the rollback sequence is limited to the sentinel device if issues arise.

In the second scenario, continuous uptime on the device is not feasible during the update process. Specifically, the currently running version (N-1) must transition to sleep mode and then be removed from the destination device. Subsequently, the new version (version N) is uploaded, configured, and activated using a wake-up command. Until version N is fully operational, the node remains in downtime and temporarily unable to collect or exchange data—an inherent vulnerability that must be managed.

5.3. Software Update in Edge Layer Affecting IoT and MES Nodes

The simple software update at the Edge level would be managed at the other levels by employing message queues. Incoming messages to the Edge level will be handled when it becomes operational again. Messages from the output queues of the Edge level will be processed until Edge components are offline. The connecting systems will raise an alarm if all the items are processed. The same will apply if the incoming buffers become fully loaded.

Considering this, it is crucial to define the buffers as wide and long enough to accommodate the amount of data that could be generated during more extended downtimes. In the scenario when the device from the Edge level must remain inactive for a period of deployment, and when there are no buffers or message queues implemented, the connected systems will run into an alarm state. Devices at the MES level will raise an alarm, but they will continue executing other actions that are not connected to the Edge level. Some functionalities will be temporarily stopped, but most work could continue.

Devices at the IoT level will not be in such an advantageous position in this case. Without a buffer, devices at the IoT level will get disconnected for the same amount of time as the Edge level devices. For IoT nodes, this will be a situation of a high alarm state, and they will execute the following course of events:

- Devices in IoT nodes detect disconnection event
- Devices raise the internal alarm
- Start reconnection procedure in predefined time frames

Without a buffer enabled, while the Edge level node is not running, IoT nodes will not have a destination where to send processed data. This will cause significant data loss for the complete deployment areas, which could be unacceptable if the process consumes an extensive amount of time. This problematic state will last until the Edge layer node starts running again. When a node from the Edge layer restarts and returns online, IoT nodes will get connected again and continue exchanging data.

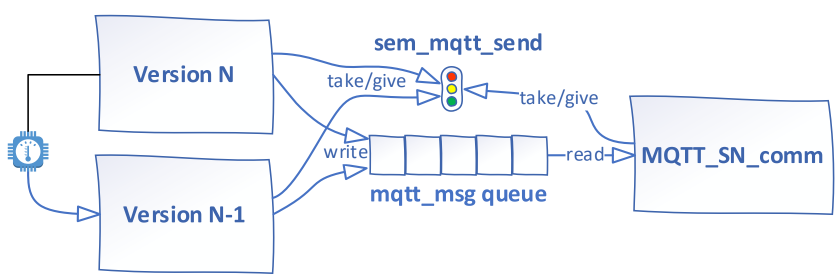

Figure 9: Software update scheme with message queue [9]

In some cases, IoT nodes will not be able to reconnect due to a change in communication protocol or a hardware error. In these cases, IoT nodes will run a general alarm, and then the Edge node must be moved back to the previous version. When an update is needed in both layers, the update notification signal will stop the general alarm, and then all IoT nodes will be updated one by one. The update will be driven from the backup node.

One of the commonly used solutions to reduce the necessity for frequent updates across the levels is the using a buffer between the layers (**Figure 9**). In this case, the buffer is implemented as the message queue. In most cases, when the communication protocol changes, only the synchronization buffer will be updated, while all the nodes in the IoT layer will continue to work. In this way, downtime will hit only one layer (in this case, the Edge layer) while the other layers will continue to run without interruptions.

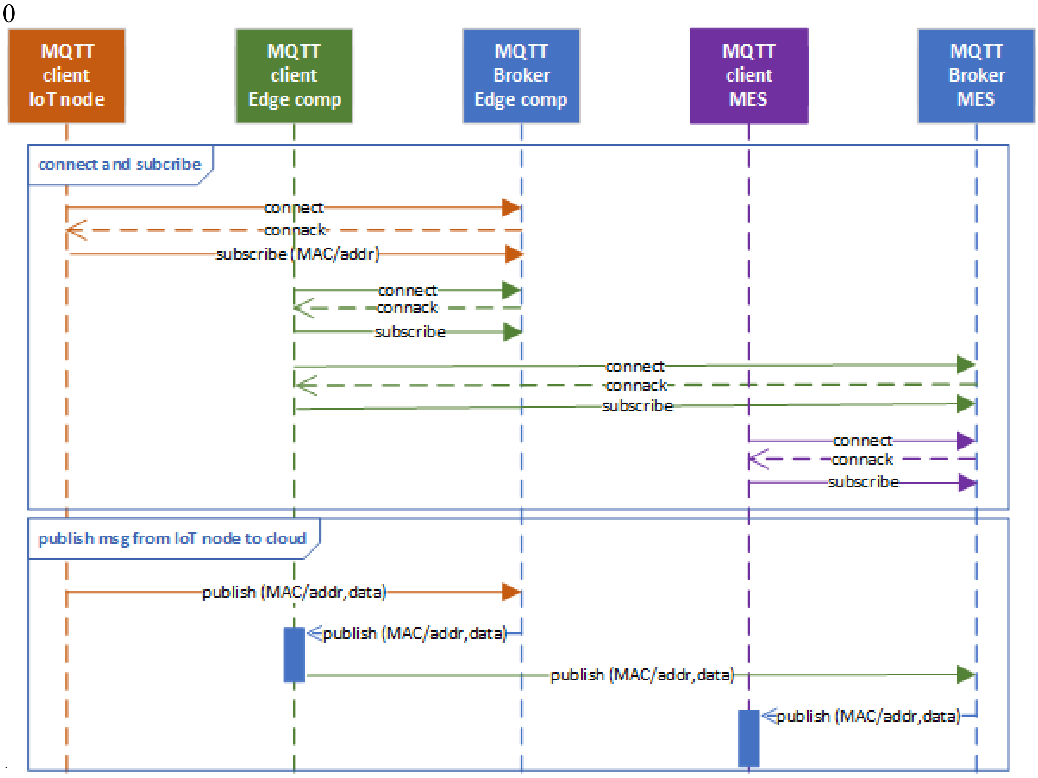

Figure 10 Reconnection sequence between IoT, Edge, and MES node

The introduction of a message queue solves the previously described issue but at the cost of a bit more complex setup and integration (Figure 10). Figure 10 This shows the process of integration with the Edge level. The approach is the same for IoT and MES nodes on opposite sides of the Edge node. The Edge nodes establish communication using message queues (MQTT in the case of the presented system). MQTT brokers and clients are installed at the Edge and the MES level. The IoT node needs only the client.

The connection is initiated from the client on one level to the broker on another. When this communication is established, the broker waits for the client's connection at its level and accepts the subscription request. In this way, MQTT clients in the IoT and Edge levels are connected through the broker at the Edge level. Similarly, MQTT clients from the Edge and MES levels will be connected through the broker at the MES level.

It must be stated that when transferring data using a message queue, data loss could happen during the software update. Message queues usually contain objects of specific types produced on one side and consumed on another. The two most common scenarios are when the connection between the message queue and one of the sides (producer or consumer) cannot be established, while another is when the data queue contains objects of unrecognizable type in the destination. The first situation is handled in a way that stops the producer until the connection is fully re-established. The second situation happens mostly when the version of consumer software is replaced in a way that stops supporting old message formats. In this case, the messages remaining in the queue will be lost. Synchronization through message queues is an essential aspect of the software update, but it goes beyond the scope of this paper.

6. Update Mechanism for MES Nodes

The main shift that could be done at the MES level is to integrate the software update mechanism into the solution. MES architecture, which we exploited in our environments, is service-oriented architecture (SOA) based on different technologies. On the server side, multiple services running to achieve necessary functionalities. The current setup is between single service and microservices since the system consists of main execution and multiple supporting services. While the supporting services could be turned on and off independently, the leading execution service must be active to put the system in run mode. In that sense, the update service is one of the services on the server side that is responsible for server and client updates. Ideally, the update service is configured to run in the independent node. It takes care of the order of the update and data buffers during the update process.

Depending on the requirements, the update service could take care of every single node in the system or equally distribute the updates depending on the node type. The update service takes care of sentinel/backup nodes (if configured) and monitors and switches different feature flags and A/B functionality variants on and off. The approach with the controllable update mechanism, driven from the single node, applies to any ISA95 level. Depending on the technology, implementation could be different, but the concept of maintaining the update process and the configuration from the single point makes the system fully controllable and maintainable. Moving these functionalities from the execution service and its connected microservices to an independent node avoids the well-known problem of the server bottleneck during the update process. In the cases where the execution service itself triggers and controls the update, the network traffic significantly rises during a brief period, which could lead to different synchronization problems.

The additional advantage of implementing such a node is the possibility of connecting it to the digital twin in the cloud. This feature makes the update over the air and synchronization with the digital twin possible. Having such a connection, a complete industrial facility could be controlled remotely, and the existing digital twin would always be available for any test and analysis.

Both client and server nodes will use the standard network protocols to operate at the MES and ERP levels. In the lower levels, the accessibility will depend on the implemented technology. Still, with the appropriate network adapters, the update node could achieve control also over the instances in Edge and IoT levels. The update node could also monitor configuration changes in production environments and take adequate action when the change is detected. Depending on the configuration or requirement, it could push the change to a digital twin, raise an alarm for the additional check, or overwrite the configuration.

The additional benefit is the more accessible support for testing and verification before moving the change production environment. As mentioned before, after the solution has been evaluated to a digital twin, test, or staging environment, the deployment for production could be ready significantly faster. The access to configurations already prepared in the digital twin environment allows the update manager to check the destination clients and easily spot if the local changes have been made. In that case, it could stop the deployment and raise the alarm to the technician to decide how to proceed. Alternatively, the update manager could override the configuration in the client machines and force the update.

The update node could also push the update for the server side. In the MES level, the server-side SOA system will also store all the actual and previous versions of the clients, allowing easier recovery and fallback in the case of unsuccessful deployment. In case of the configuration on multiple server instances, the update manager will track the order of the update, using the feature flag system to control the start and stop of all microservices. As has been mentioned, the leading service on the server side is the execution service required to be active to make the entire system run.

The server side of the update mechanisms is responsible for communicating with clients and other external systems – such as databases, configuration storage, and other external services. It could be configured to retrieve data from multiple sources and prepare the deployment packages according to the status set in the digital twin. As mentioned, its role is also to monitor the validity of the complete system to check if the configurations or client versions may change outside of the deployment process and to raise the alarm in case of misalignments.

Both clients and service exchange ping messages to keep the system communication status. Ping messages could contain distinct parameters and run in different periods. While some are used only to check if there are responses on the other side, others could be used to verify client versions and configurations. At the same time, regular messages that exchange data are used to maintain connectivity. Every message delivery failure could trigger an alarm and run the re-assessment process and eventual network reconfiguration. In some cases, the sentinel clients could take the server role for the group of clients and maintain connectivity in the alarm mode.

6.1. Update Node Routines

The *DeploymentHelper* component handles configuration updates in scenarios where the application reverts to an older version or when a specified time for updating specific clients has elapsed, necessitating updates for the remaining clients. This component is situated on the server side, as all configurations for this application reside on the same machine as the service. Consequently, the service possesses all necessary permissions for file modification and physical addresses where the files are located.

The base class diagram to support client updates is presented in Figure 11. Instances of class Update Status Info are used to store the info about the version and application name. The bare minimum of the data should be maintained for every client. They come to the MES or update service as part of ping messages from clients. Combining these pieces of information with the data in the internal cache, the process that keeps track of versions could maintain their activity tables regularly. Activity tables are kept in the update process and periodically synchronized with the digital twin environment. The objects of this class, either persisted in the memory or in a dedicated location in the file system, are also used as the contact point for the *DeploymentDispatcher*.

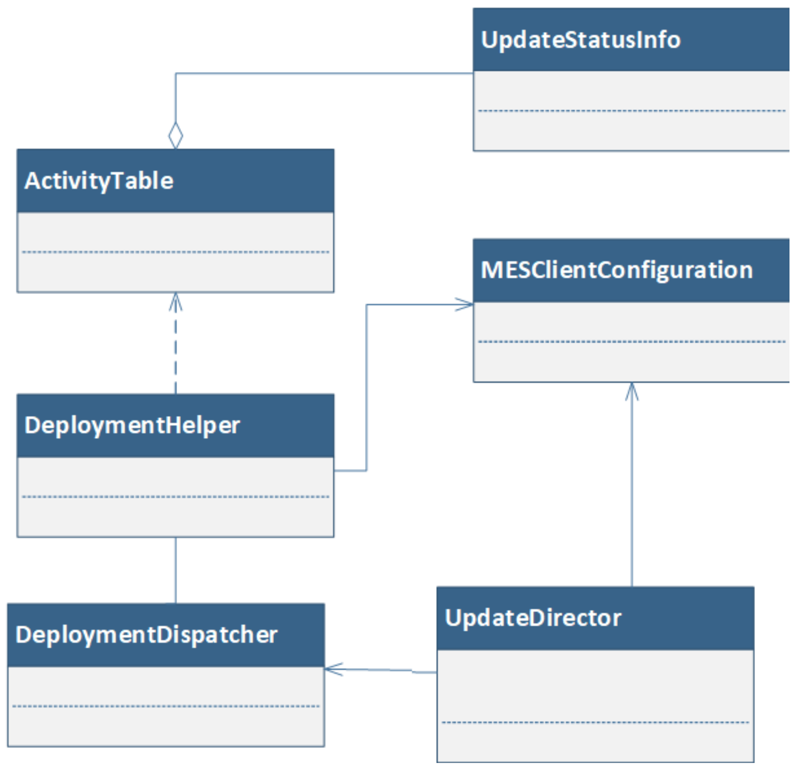

Figure 11 The relations between main entities in the deployment subsystem

On the single node level, the *DeploymentDispatcher* is the component responsible for the entire update process. It could be configured to ping the server or sentinel client to check for the new version or to wait for the update notification. Once the latest version is discovered, the update process will start and be executed in *UpdateDirector*.

The update thread will run in the background and gather all necessary configurations and binaries from the update node to form the new version of the client. After a new client is formed, it will trigger the rest of the process and perform possible additional steps, such as a backup of the previous version and a blue/green switch. When configured in the sentinel node, this functionality will propagate the installation to other nods in the group. As the ultimate step of the update process, the information about the software version will be pushed back to the update node and the digital twin to ensure the proper version info synchronization.

It is essential to point out that *DeploymentDispatcher* could progress both with complete client updates and partial functionality enabled/disabled. In that way, direct support for feature flags is implemented. The client could come with an updated version of the software, but in case of any problem, the additional features could be disabled. Also, configuration changes could be pushed from the server to ensure the required reconfiguration.

The update manager instance is created when the application is started. It is constantly active and periodically checks for recent updates if configured to run in active mode. During initialization, the update manager checks the application's version and all modules to ensure the up-to-date application signature is ready for comparison with the version on the server.

The update manager listens to the server's ping and notification commands in passive mode. In this scenario, the server notifies the client that the updated version is available, and the client starts the update process. Also, it is usual to configure both modes in the same and dedicate each process to a specific part of the update process. For example, the check for the new client version could be configured in active mode, while the configuration updates could be passive and pushed by the server instead of the client's request.

An instance of this class creates an object of the *DeploymentDispatcher* class and immediately invokes its primary function, as shown in **Figure 12**. This function manages a specific client's update process and could halt software updates if necessary. It is responsible for initiating the update process as long as the attribute's value that keeps the loop alive remains unchanged.

This method initially attempts to retrieve the file containing the necessary information for updating. If that file does not exist, the method returns a false value, indicating that it failed to obtain the appropriate file. If the file is successfully retrieved, relevant data required for updating is extracted from it. Subsequently, it checks whether beta updates are active. If they are and the specified time for this type of update has elapsed, the UpdatesManifest.xml file is updated. In this file, the active software version is set to the “beta” version, and updates of this type are marked as inactive. Next, it verifies whether the current client version matches the version that should be on our machine (**Figure 12**).

New client versions must be downloaded if the current client version is missing or differs from the version in the file while beta updates are inactive. In the case of active beta updates and the client still not being on the beta version, affirmative information is returned to download new files, but only if random access permits. This ensures that not all clients receive the updated value, only those with “luck” (**Figure 12**). All clients downloading the updated version exit the function and return a value *true*. If the random selection does not choose a client, the thread responsible for updating is put to sleep for a predefined number of minutes. Afterward, the thread is again put to sleep for a few seconds, triggering the update check.

The *DownloadUpdates* method retrieves updates from the corresponding file (the file path is specified in the update specification). If beta updates are active, it fetches the file named in *BetaFilePath*; otherwise, it retrieves the file named in *FilePath*. *BetaFilePath* is used when the A/B deployment must be supported, while for regular deployments, the filed *FilePath* directs to the update location. This approach also solves the issue of network connection interruptions to the new client, as the update is not applied until it is fully downloaded locally. Finally, the application that launches the latest client version is restarted.

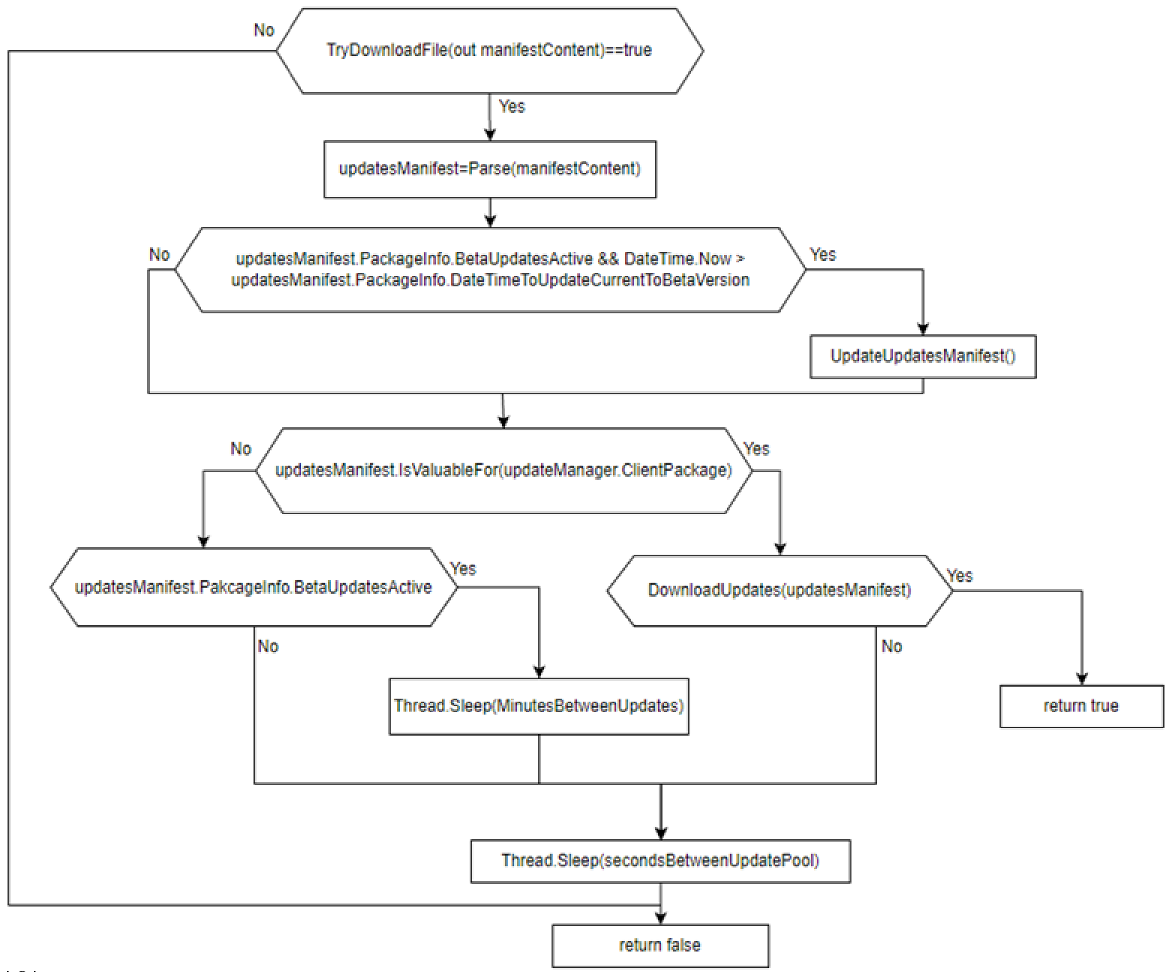

Figure 12 The sequence of choosing and verifying the correct software version

7. Results and Discussion

This research came out of the project and resulted in developing a complex industrial monitoring system aimed at all ISA95 levels – from IoT nodes through Edge and MES to ERP level. During the project, for more than 15 years, our team was focused on different aspects of development and implementation, starting from the improvements of CAD/CAM databases [31], through all different implementations at all levels, up to development for the software update system integrated with the cloud [9] [10].

The tests are conducted in a digital environment that resembles the industrial façade carpentry facility. Section 4 gives all the necessary details in the composition of the test environment. Such production is interesting since it combines different production types – from serial production up to one-of-a-kind configured products [32]. At the same time, such a facility combines processes based on various physical and chemical procedures in material treatment, thus requiring all kinds of digital interaction, starting from thermal sensors and actuators through intelligent industrial machines integrated with MES clients up to ERP software enhanced with different CAD and planning tools (Figure 13) [33].

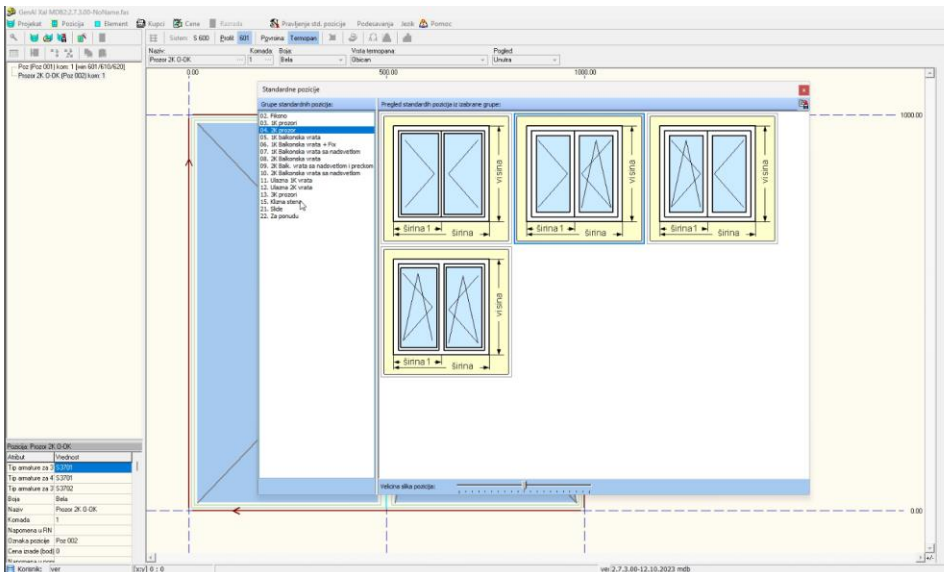

Figure 13 View on the ERP client - production order definition

Having experience with diverse types of software developed on different ISA95 levels, we identified the common problems in software updates and tend to generalize the update architecture, node structure, and processes. The results were preliminarily evaluated at the IoT and ERP levels because they have limited effects on the rest of the system, being connected only to the neighboring level. Following the results and recommendations from the previous work, we decided to expand the update system to the most challenging MES level (Figure 14).

7.1. Guidelines for Combining Different Deployment Strategies

Our research was led by the request to reduce the potential downtime during the software update in a challenging environment such as the industrial facility. The actual criticality of this request is not equal from level to level, but the customer requirement tends to go to 0 downtime regardless of the software system. To reach this goal, we decided to replace the standard deployment (stop-copy-run) with a combined strategy that should employ the benefits from different deployment processes. Looking at the single node, we aimed for the blue/green deployment as the base concept.

This concept could be enriched then with feature flags, dark mode, and A/B testing deployments to fine-tune the update process and to release new functionalities in the controllable environment. At the level of the node networks, the concepts of canary deployment were applied to the development of backup and sentinel nodes, which function as the group leads and will receive the first update and then push forward deployment into the subsequent nodes in its group. Combining these three well-known approaches in the proposed way, we tried to benefit from all the positive aspects we could get:

- Blue-green deployment gives the possibility for a fast version switch.
- Dark mode and feature flags allow simple enabling or turning off single functionalities.
- A/B testing allows running several feature variants to let the customer decide which to accept.
- Canary deployment allows prompt identification of deployment errors.
- The presence of a synchronization buffer allows us to keep one layer insulated and still operative while the connected layers are in downtime or performing an update.

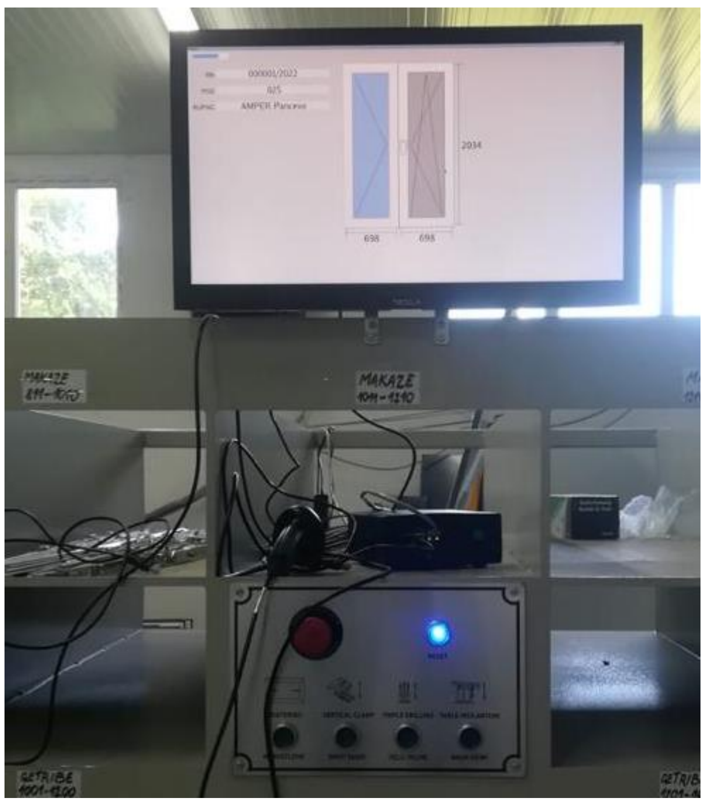

Figure 14 MES client set up in a factory environment - connected to cutting machine and the signals that bring measurement values

The proposed methodology is initially subjected to rigorous testing at the IoT level. This choice stems from the formidable constraints encountered in this stratum, encompassing software resources, network bandwidth, and energy consumption limitations. Additionally, deploying IoT systems in critical and hazardous environments underscores the need to minimize direct human intervention and avoid installing supplementary infrastructural components, such as power or network cables.

Complicating matters further, physical access to IoT nodes remains a challenging endeavor. This challenge arises not solely from technological considerations but also from mechanical and security protocols. Removing various mechanical elements in certain instances becomes necessary to reach IoT devices physically. Moreover, these devices often operate in environments hazardous to human safety, necessitating stringent procedures for device access.

Previously, a conventional update approach, or recreate deployment, was employed, wherein the software component was replaced either entirely or partially (via a stop-copy-start process). However, this standard update method posed several issues, which can be briefly summarized as follows:

- The downtime was always present. If the software component is in the updating process, the software device cannot be used.
- In case of an erroneous update, software should be restored to its previous version, which would lead to further downtime.
- The restore process sometimes drains the battery, requiring the personnel member to go to the hazardous area.
- Connected layers could not continue to work generally since they were flooded with alarm signals.

Table 3 The effects of the proposed deployment strategy on the IoT level containing 100 IoT nodes connected to a single Edge node (TD – time to shut down the software in the node, TU – time to start the software in the node, TS – time switch between the versions, IS – software instance size per node, NN – number of nodes). Combined from [9] and [10]

Measurement	With recreate deployment	With hybrid strategy
Number of software uploads to IoT level – successful deployment	NN	1 (only to the leading node)
Number of internal uploads – successful deployment	0	NN
Number of software uploads - unsuccessful deployment	Average 8% of NN	1 to the backup node
Security check on upload	NN	1 (only to backup node)
Number of internal software uploads – unsuccessful deployment	0	1
Rollbacks with unsuccessful deployments	8% of NN	1 + 1
Downtime per node	TD + TU (in seconds)	TS (in milliseconds)
Used space for software per node (with blue-green approach)	1 x IS	2 x IS
Used space for software with buffer node	NN x IS	NN x IS + IS
Update distribution	Manual or with a task scheduler	Optimized by backup node or pushed from the cloud
Downtime when connected layer update	If the update is running	Until the buffer has data

7.2. Achieved Results

Our results with the proposed combined deployment approach proved our expectations and varied between different software layers and scenarios. Applying the proposed strategy reduced the overall downtime and number of unnecessary rollbacks. This was achieved by the cost of implementing the backup node, the implementation of the buffer level, and a slight increase in data traffic. Table 3 shows the behavior of the network of 100 IoT nodes analyzed in a test environment.

Having the configuration with one leading node, the total number of updates coming from the update node or the cloud to the IoT network will be reduced from the total number of nodes (NN in further text) to one. The updated version will come from the outside system to update the node, which will guide the update for the rest of the IoT nodes. In this way, the bottleneck in communication between the IoT level and the rest of the system will be reduced or eventually avoided. This way, the number of security checks will be reduced to only one. In a scenario where every node gets an update outside the network, a security check will be performed every time due to standard security policies.

The proposed hybrid approach will require more space. If the clients can support blue/green deployment, they will need twice as much space as in the case of recreate deployment. One additional slot for the distributed version should be added to the space required. The sentinel client will use the distribution/sentinel/backup node to download the updated version and then forward the update.

Table 4 The estimated effects of the proposed deployment strategy in MES and ERP level (TD – time to shut down the software in the node, TU – time to start the software in the node, TS – time switch between the versions, TF – time needed to activate feature flags and A/B features, IS – software instance size per node, BS – buffer size, NN – total number of nodes, N1 – number of level 1 nodes (sentinel/backup nodes), G – number of level 2 groups, AG – average number of level 2 nodes per group AG = (NN – N1)/G)

Measurement	Recreate deployment	Hybrid deployment ERP level	Hybrid deployment MES level
Number of software uploads to level 1 nodes – successful deployment	NN	N1	1 + (N1 – 1)
Number of software up-loads to level 2 nodes (average per group, successful deployments)	0	AG	AG
Number of software uploads to level 1 (rollback needed)	NN	Up to N1	1
Number of software uploads to level 2 (rollback needed)	0	0	AG
Security check on upload	NN	N1	1 Only in the update node
Downtime per node	TD + TU	TS	TS + TF
Total space used	NN x IS	NN x (2 x IS + BS) + IS	NN x (2 x IS + BS)
Update distribution	Manual or with a task scheduler	Optimized by backup node	Over the air
Downtime when connected layer update	If the update is running	Until the buffer has data	0 – ERP Until the buffer has data – Edge / IoT

The concept proposed for IoT nodes in [10], further evolved and applied to the ERP nodes [10]. With further customization, it is successfully applied to the MES level. The expected effect is presented in Table 4. Both ERP and MES clients share similarities in size and software architecture. Both have more extensive software instances than those in the IoT and Edge levels. Due to the software's mentioned size, update distribution could cause problems comparable to those from the IoT level, primarily if the update is run from the same node where the server is running. In that case, the single node should run NN uploads, which could take significant network resources.

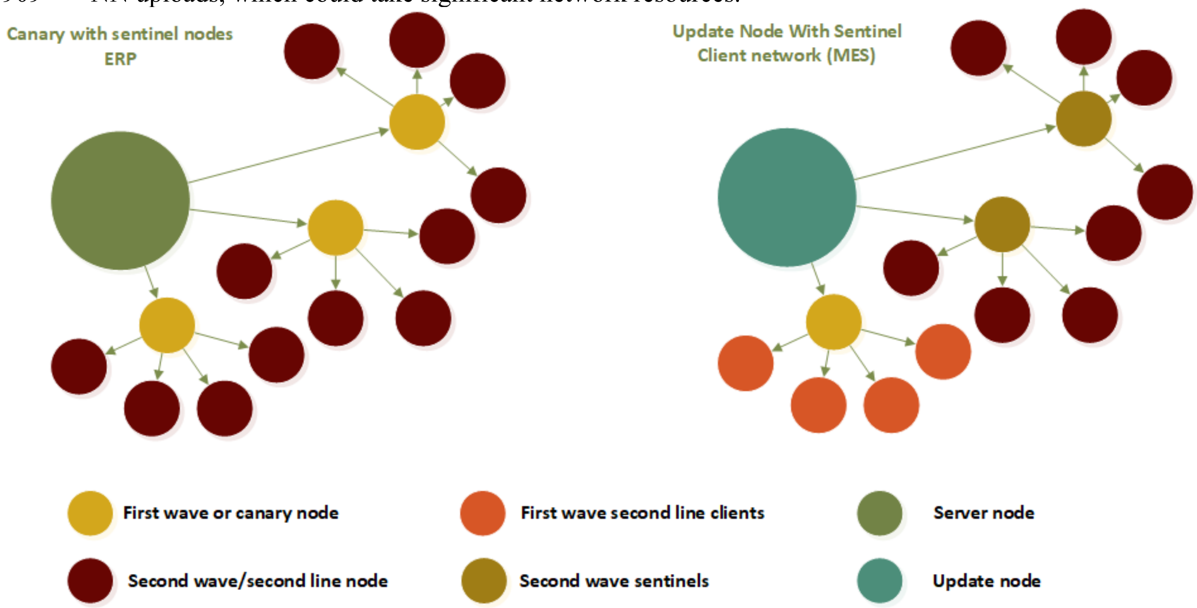

Figure 15 Differences in deployment approach for ERP (left, as presented in [10]) and MES clients (right)

To address this challenge, a strategic division of client nodes into N1 groups by AG clients is proposed (Figure 15). This approach draws inspiration from the canary deployment methodology, wherein a dedicated group of clients serves as the initial testing cohort. During the first iteration, updates are dispatched to sentinel nodes, responsible for essential testing. Subsequently, these sentinel nodes propagate the verified updates to the nodes within their respective groups. In the event of an error detected at the sentinel level, a rollback ensues, ensuring that most clients remain shielded from erroneous software versions.

This approach undergoes slight adaptation when applied to the MES layer. The rationale behind this modification lies in the inherent diversity of MES clients. Unlike ERP clients, which typically exhibit uniform features, MES clients cater to distinct operational stations, each potentially possessing a significantly separate set of functionalities. In the MES environment, an initial client group is selected for deployment. The updated version is relayed to its sentinel node, where thorough verification occurs. Upon successful verification, the updated version cascades to the remaining group members. Subsequently, the verified functionality extends to other sentinel nodes.

While this approach does not directly reduce total network traffic, it effectively distributes the load across update and sentinel nodes, mitigating network traffic hotspots. Anticipated downtime per node may be slightly higher for MES clients due to the activation of feature flags and A/B functionalities. Additionally, depending on configuration, MES clients may require time to establish connections with signal sources from distinct levels. Notably, integrating the update mechanism with the Cloud level and the digital twin introduces the prospect of fully controllable over-the-air deployment, potentially paving the way for a transition to software-as-a-service for specific system elements.

Table 5 Effects of different client deployment approach to MES and ERP level – 3 groups of 10 clients (STD – standard approach, WoD – Wave of Distribution)

Measurement	Recreate deployment ERP	Hybrid deployment ERP level (canary with sentinel)	Recreate deployment MES	Hybrid deployment MES level (groups with sentinel)
Number of update packages sent from the server to clients (1st WoD)	30	3	30	1 + 2
Amount of data sent from the server to clients (in GB, 1st WoD)	1.35	0.14	0.75	0.03 + 0.07
Network traffic peak (in %, server outbound, 1st WoD)	100	18.65	78.40	5.67
Distribution group size (2nd WoD)	-	10	-	10
Distribution time per group of clients (In seconds, 1st WoD)	64.28	7.55	41.19	2.77 + 6.01
Distribution time per group of clients (In seconds, 2nd WoD)	-	17.08		12.55
Single client switchover/update time (In seconds)	32.28	4.58	25.19 (only MES functionality) 31.22 (full connectivity)	2.41 (only MES functionality) 8.67 (full connectivity)
Single client switchover/restart time when rollback is needed (seconds)	34.10	6.78	26.49 (only MES functionality) 33.53 (full connectivity)	4.33 (only MES functionality) 9.02 (full connectivity)

We compared the update behavior for the array of 30 ERP and 30 MES clients running in the test environment to evaluate predicted values. They have been split into three groups of ten clients for the simulation. The findings, presented in Table 5, align with the estimation from Table 4. Due to their smaller size, MES clients create less network traffic than ERP clients. The amount of required space and network peaks are lower for the MES network.

7.3. Advantages and Drawbacks

The advantage of the approach shown in this work is that if it is applied to MES nodes, it results in faster recovery if the deployment error is noticed, compared to the one presented in [9] and [10]. Usually, it is enough to do the rollback only in one sentinel node. The next advantage is the possibility of running multiple versions of some functionality and quickly switching them on or off. Ultimately, integrating with cloud services and establishing a complete digital twin helps detect errors and change. The environment we used for the test is a demo digital twin for beta testing.

It is essential to note that two separate times must be measured when the MES client is started or when the switchover is handled. The most critical moment is when the client is in running mode and connects to the MES service, allowing it to perform standard MES functionality – operation execution, labor logging, etc. Next is the moment when the client is connected to other data sources. In our example, clients are connected to an OPC (object for process control) server that acts as a system that collects measurements from the sensors. Generally, these data sources could be different depending on the area of the industrial facility where the client is running.

The software update challenges discussed in this study constitute only a portion of the broader complexity. For over fifteen years, we have continuously relied on systems developed by our research group, honed through rigorous coordination, and field-tested in partner industrial facilities. The software update process encompasses several critical dimensions, including compatibility concerns, system stability, data migration intricacies, and the imperative of user adoption. Addressing compatibility issues necessitates comprehensive testing across diverse system configurations before deployment.

To this end, we advocate for establishing a dedicated test environment within our domain or creating a digital twin in the cloud. For instance, transitioning to a different platform version for Windows application development may introduce incompatibilities with OPC servers. Similarly, upgrading the database server to a newer version could disrupt continuous connectivity between MES or ERP systems until the connection driver is updated. Altering the data structure of messages stored in message queues poses the risk of data loss for existing records, rendering them unreadable by the current system.

User adoption hinges on effective communication and targeted training to elucidate the benefits of updates and familiarize users with new features. Soliciting feedback from users both before and following updates facilitates the identification and resolution of any emerging issues. The strategic inclusion of A/B deployment techniques further enhances this process.

The typical application of the proposed software update mechanisms is limited to some point. This means that the suggested set of updates could not be directly used for software not developed in the line of the examined software development and deployment approaches. For example, if the software has no properly exposed extension and configuration classes, there will not be the possibility to use feature flags or A/B approaches. On the other hand, blue/green and canary deployments could be implemented through a committed team supported with the necessary hardware and acquiring specific deployment routines. A deeper implementation of the proposed deployment solution would require additional pieces of software and/or additional adaptation in the target software.

During the development process, not all pieces of software were designed suitably and flexibly for such update mechanisms. Initially, the MES software was developed with fixed configuration files in which content was loaded on system startup, and the update was not possible while the software was running. This was primarily related to the server side. Any configuration change used to lead to service restart, which eventually results in execution disruption. For this reason, the blue/green deployment was the first that was included in the setup. It guaranteed reduced downtime and faster system operational availability. On the other hand, the software adaptation for MES clients came a bit later since it only needed to restart local clients in the operator's place, which had a limited impact. The next set of updates was the approach that could trigger configuration refresh through a database or file reload. With this approach, feature flags and later approaches became fully supported, and the software was ready to become a part of the complex deployment system, significantly reducing downtime when redeployed.

Mitigating system disruptions involves judiciously scheduling updates during off-peak hours and transparently communicating potential downtime to users. Meanwhile, prudent planning and rigorous testing of data migration procedures minimize complications arising from data transfer.

In summary, a carefully orchestrated update process, underpinned by thoroughly vetted software versions and executed at the opportune moment, constitutes the linchpin of a successful upgrade.

8. Conclusion

Having more than a decade and a half of experience with industrial systems, our research team went through different projects involving software development at all ISA95 levels. The challenges in development vary across the levels due to user requirements, technical complexity, and performance expectations. All these software instances must work in accordance and be a reliable element of the industrial facility. The common challenge for all the pieces of software is the system update. Usually, the system on one level consists of the server and several dozen or hundreds of clients. When it comes to the update, it should be done as fast as possible and with lower resource consumption without creating bottlenecks in the facility.

The research findings significantly advance the formulation of deployment strategies for intricate, layered industrial software systems. When deploying software updates, several common challenges arise, including downtime, increased network traffic, and storage space utilization. At lower levels, energy consumption during the deployment process also warrants consideration.

We introduce additional backup nodes into the system to address the limited storage space issue. Although these backup nodes exhibit a slightly larger volume than regular IoT nodes, this tradeoff is deemed acceptable given the achieved outcomes. Notably, total downtime has been dramatically reduced—from seconds to milliseconds—representing a reduction of less than one percent of the initial duration.

The approach used in IoT nodes [9], was successfully applied to ERP [10] and MES levels by improving the defined hybrid deployment mode. The findings align with those observed for IoT nodes, emphasizing the potential incorporation of novel features and deployment strategies. This adaptability makes the deployment process for ERP and MES clients more user-friendly, fostering higher user acceptance rates.

We devised a hybrid strategy that amalgamates blue-green, canary, and dark mode elements with feature flags, A/B testing, and enhanced standard deployments. This strategy is bolstered by an inter-layer buffer and the inclusion of specific nodes—the update node on the server side and backup and sentinel nodes on the client side. By implementing this approach, we effectively curtailed overall downtime, reducing the duration required for system restart to a period proximate to the switchover. Remarkably, this reduction translates to less than 10% of the time typically consumed by classic deployment methods. The most noticeable improvement is in the case of erroneous deployment when the error could be tracked down and stopped in the first sentinel node.

With the backup/sentinel node active, we reduced the number of software uploads in case of an erroneous update to the time needed for two switchovers of the single node. If chosen correctly, the initial sentinel node will provide an adequate test environment for error detection. Unlike the ERP clients, where the approach was to release the update to all sentinel nodes, with MES clients, the strategy was to send the update to a single sentinel, and then it would take care of its group. In the worst case, the targeted group needs to be reverted, but this will be done inside the group without the need for interaction with the server or the update node.

The changes in the deployment process applied to MES nodes are driven mainly by the Industry 4.0 paradigm and the requirements that came with it. MES and Industry 4.0 are transforming manufacturing practices by digitizing and making processes intelligent, enabling organizations to cater to individual customer requirements and achieve operational excellence. In short, MES and Industry 4.0 are revolutionizing manufacturing by integrating advanced technologies and data-driven systems to create a more interconnected and efficient production environment.

Enhancing the efficiency of the software update process stands as a pivotal element within an optimized production environment. The overarching objective is facilitating software updates beyond scheduled maintenance windows. Leveraging the proposed hybrid deployment method, seamless layer-wide updates become feasible, particularly when interactions with other levels remain unchanged. Notably, this approach significantly truncates downtime—from hours and minutes to mere seconds and milliseconds. Furthermore, our future trajectory involves extending our efforts to the Edge level. This strategic expansion aims to devise solutions that mitigate the impact of buffering and inter-level communication system modifications more effectively.

9. Acknowledgement

This work was supported by CERCIRAS – COST Action CA19135, funded by COST.

The Ministry of Science, Technological Development, and Innovation of the Republic of Serbia supported this work [grant number 451-03-65/2024-03/200102].

10. References

[1] ISA95, Enterprise-Control System integration- ISA (no date) isa.org. Available at: https://www.isa.org/standards-and-publications/isa-standards/isa-standards-committees/isa95 (Accessed: 28 April 2024).

[2] Shu, Zhaogang, et al. "Cloud-integrated cyber-physical systems for complex industrial applications." *Mobile Networks and Applications* 21.5 (2016): 865-878.

[3] Kondratenko, Yuriy, et al. "Complex industrial systems automation based on the Internet of Things implementation." *International Conference on Information and Communication Technologies in Education, Research, and Industrial Applications*. Springer, Cham, 2017.

[4] Sha, Kewei, et al. "Edgesec: Design of an edge layer security service to enhance IoT security." *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*. IEEE, 2017.

[5] Li, He, Kaoru Ota, and Mianxiong Dong. "Learning IoT in edge: Deep learning for the Internet of Things with edge computing." *IEEE network* 32.1 (2018): 96-101.

[6] Sajid, Anam, Haider Abbas, and Kashif Saleem. "Cloud-assisted IoT-based SCADA systems security: A review of the state of the art and future challenges." *IEEE Access* 4 (2016): 1375-1384.

[7] Coronado, Pedro Daniel Urbina, et al. "Part data integration in the Shop Floor Digital Twin: Mobile and cloud technologies to enable a manufacturing execution system." *Journal of manufacturing systems* 48 (2018): 25-33.

[8] Chofreh, Abdoulmohammad Gholamzadeh, et al. "Development of guidelines for the implementation of sustainable enterprise resource planning systems." *Journal of Cleaner Production* 244 (2020): 118655.

[9] Rajković, Petar, Dejan Aleksić, Dragan Janković, Aleksandar Milenković, and Anđelija Đorđević. n.d. “Resource Awareness in Complex Industrial Systems -A Strategy for Software Updates.” Accessed April 8, 2024. https://ceur-ws.org/Vol-3145/paper10.pdf.

[10] Rajković, Petar, Dejan Aleksić, Andjelija Djordjević, and Dragan Janković. 2022. "Hybrid Software Deployment Strategy for Complex Industrial Systems" Electronics 11, no. 14: 2186. https://doi.org/10.3390/electronics11142186

[11] Cozzani, Valerio, et al. "Quantitative assessment of domino and NaTech scenarios in complex industrial areas." *Journal of Loss Prevention in the Process Industries* 28 (2014): 10-22.

[12] Chen, Yusong, et al. "Research on software failure analysis and quality management model." *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2018.

[13] Usman, Muhammad, et al. "Compliance requirements in large-scale software development: An industrial case study." *International Conference on Product-Focused Software Process Improvement*. Springer, Cham, 2020.

[14] Kalunga, Joseph, Simon Tembo, and Jackson Phiri. "Industrial Internet of Things Common Concepts, Prospects and Software Requirements." *vol* 9 (2020): 1-11.

[15] Chen, Chao, Genserik Reniers, and Nima Khakzad. "A thorough classification and discussion of approaches for modeling and managing domino effects in the process industries." *Safety science* 125 (2020): 104618.

[16] Ren, Zihui, Cheng Chen, and Lijun Zhang. "Security protection under the environment of WiFi." *2017 International Conference Advanced Engineering and Technology Research (AETR 2017)*. Atlantis Press, 2018.

[17] Kim, Dae-Young, Seokhoon Kim, and Jong Hyuk Park. "Remote software update in trusted connection of long-range IoT networking integrated with mobile edge cloud." *IEEE Access* 6 (2017): 66831-66840.

[18] Asokan, N., et al. "ASSURED: Architecture for secure software update of realistic embedded devices." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018): 2290-2300.

[19] Mugarza, Imanol, Jorge Parra, and Eduardo Jacob. "Cetratus: A framework for zero downtime secure software updates in safety-critical systems." *Software: Practice and Experience* 50.8 (2020): 1399-1424.

[20] Stević, Stevan, et al. "IoT-based software update proposal for next generation automotive middleware stacks." *2018 IEEE 8th International Conference on Consumer Electronics-Berlin (ICCE-Berlin)*. IEEE, 2018.

[21] Mirhosseini, Samim, and Chris Parnin. "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?." *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017.

[22] Fowler, M. "Blue-green deployment, March 2010." (2016).

[23] Tarvo, Alexander, et al. "CanaryAdvisor: a statistical-based tool for canary testing." *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 2015.

[24] Killi, Bala Prakasa Rao, and Seela Veerabhadreswara Rao. "Towards improving resilience of controller placement with minimum backup capacity in software defined networks." *Computer Networks* 149 (2019): 102-114.

[25] Vincent, L. (2016), "Marketing Strategies for Commercialization of New Technologies ☆ ", Technological Innovation: Generating Economic Results (Advances in the Study of Entrepreneurship, Innovation and Economic Growth, Vol. 26), Emerald Group Publishing Limited, Leeds, pp. 257-287. https://doi.org/10.1108/S1048-473620160000026009

[26] Pleshko, L., Heiens, R. The contemporary product-market strategy grid and the link to market orientation and profitability. J Target Meas Anal Mark 16, 108–114 (2008). https://doi.org/10.1057/jt.2008.2

[27] Buzachis, A., Galletta, A., Celesti, A., Carnevale, L., & Villari, M. (2019, June). Towards osmotic computing: a blue-green strategy for the fast re-deployment of microservices. In 2019 IEEE Symposium on Computers and Communications (ISCC) (pp. 1-6). IEEE.

[28] Mampage, A., Karunasekera, S., & Buyya, R. (2022). A holistic view on resource management in serverless computing environments: Taxonomy and future directions. ACM Computing Surveys (CSUR), 54(11s), 1-36.

[29] Chien, C. (2020) What is rapid application development (RAD)?, Codebots. Available at: https://codebots.com/app-development/what-is-rapid-application-development-rad (Accessed: 28 April 2024).

[30] Munikanth (2023) Kubernetes Deployment Strategies, Medium. Available at: https://medium.com/@munikanthtech/kubernetes-deployment-strategies-fc1557d21e8f (Accessed: 28 April 2024).

[31] Aleksić DS, Janković DS (2009) The use of scripts in a CAD/CAM database. The X International Conference on Information, Communication and Energy Systems and Technologies (ICEST 2009), June 25–27, Veliko Tarnovo, Bulgaria

[32] Aleksic, Dejan S., Dragan S. Jankovic, and Petar Rajkovic. "Product configurators in SME one-of-a-kind production with the dominant variation of the topology in a hybrid manufacturing cloud." The International Journal of Advanced Manufacturing Technology 92 (2017): 2145-2167.

[33] Aleksić, D.S., Janković, D.S. & Stoimenov, L.V. A case study on the object-oriented framework for modeling product families with the dominant topology variation in the one-of-a-kind production. Int J Adv Manuf Technol 59, 397–412 (2012). https://doi.org/10.1007/s00170-011-3466-4

[34] Rajković, P., Aleksić, D., Janković, D. (2024). The Implementation of Battery Charging Strategy for IoT Nodes. In: Zeinalipour, D., et al. Euro-Par 2023: Parallel Processing Workshops. Euro-Par 2023. Lecture Notes in Computer Science, vol 14352. Springer, Cham. https://doi.org/10.1007/978-3-031-48803-0_4