

Inka Sorvala

ASSESSMENT AND IMPLEMENTATION OF AN AUTOMATIC DEPLOYMENT TOOL FOR INDUSTRIAL AUTOMATION APPLICATIONS

Master of Science Thesis
Faculty of Engineering and Natural Sciences
Examiners: Professor Jose Luis Martinez Lastra
University Instructor Luis Enrique Gonzalez Moctezuma
March 2026

ABSTRACT

Inka Sorvala: Assessment and Implementation of an Automatic Deployment Tool for Industrial Automation Applications
Master of Science Thesis
Tampere University
Master's Degree Programme in Automation Technology
March 2026

The modernization of industrial automation continues to increase the number of interconnected devices and the degree of digitalization in production environments. As systems become more complex, companies must update their engineering practices to maintain efficiency and competitiveness. Despite this technological progress, many industrial processes, such as the deployment of automation applications, remain inefficient due to manual and repetitive workflows. This creates a growing interest in applying established software engineering methodologies to industrial automation in order to streamline workflows and reduce manual effort.

This thesis investigates opportunities to automate parts of the deployment process for automation applications within a newly adopted, web-based Distributed Control System (DCS). The introduction of this modern DCS platform provides an opportunity to reassess and update the deployment workflow traditionally used to deploy these applications. To identify feasible areas for automation, the research examines deployment methodologies commonly used in traditional software development and evaluates their applicability in industrial environments.

The current deployment process was analyzed through discussions with production and engineering professionals, a review of existing documentation, and a detailed field study of an ongoing project. This analysis indicated that the application generation phase presents opportunities for improvement, as it involves manual and repetitive tasks that could be effectively supported through automation.

Based on these findings, the thesis designed, implemented, and tested a Proof of Concept (PoC) tool aimed at automating selected steps of the deployment workflow. The PoC was evaluated in a controlled test environment and with feedback from engineers, who reported positive experiences and confirmed that the tool effectively executed key deployment stages. The evaluation demonstrated that the PoC provides a solid technical foundation for future development toward a production ready deployment solution.

Overall, the results indicate that such a solution can substantially improve deployment efficiency, reduce manual engineering workload, and support the broader digitalization efforts taking place in industrial automation.

Keywords: Industrial Automation, Distributed Control System, Deployment Automation, Robot Framework, Jenkins Pipeline

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Inka Sorvala: Teollisuuden automaatio-sovellusten automaattisen käyttöönotto-työkalun arviointi ja toteutus
Diplomityö
Tampereen yliopisto
Automaatiotekniikan diplomi-insinöörin tutkinto-ohjelma
Maaliskuu 2026

Teollisuusautomaation modernisaatio lisää jatkuvasti toisiinsa kytkettyjen laitteiden määrää ja tuotantoympäristöjen digitalisaatiota. Järjestelmien monimutkaistessa yritysten on päivitettävä suunnittelukäytäntöjään tehokkuuden ja kilpailukyvyn varmistamiseksi. Teknologisesta kehityksestä huolimatta monet teolliset prosessit, kuten automaatio-sovellusten käyttöönotto, ovat edelleen tehottomia manuaalisten ja toistuvien työvaiheiden vuoksi. Tämä lisää kiinnostusta hyödyntää perinteisessä ohjelmistokehityksessä käytettäviä menetelmiä teollisuusautomaation työkalujen virtaviivaistamiseksi ja manuaalisen työn vähentämiseksi.

Tässä työssä tutkitaan mahdollisuuksia automatisoida osia automaatio-sovellusten käyttöönottoprosessista uudessa, verkkopohjaisessa hajautetussa ohjausjärjestelmässä (engl. Distributed Control System, DCS). Tämän modernin DCS-alustan käyttöönotto tarjoaa mahdollisuuden arvioida uudelleen ja päivittää sovellusten perinteinen käyttöönottoprosessi. Automaatiolle soveltuvien osa-alueiden tunnistamiseksi työssä tarkastellaan perinteisessä ohjelmistokehityksessä käytettyjä käyttöönottomenetelmiä ja arvioidaan niiden soveltuvuutta teollisiin ympäristöihin.

Nykyistä käyttöönottoprosessia analysoidaan keskustelemalla tuotannon ja suunnittelun ammattilaisten kanssa, tarkastelemalla olemassa olevaa dokumentaatiota sekä tekemällä yksityiskohtainen kenttätutkimus käynnissä olevasta projektista. Havaintojen perusteella sovellusten generointi osoittautui potentiaalisesti kehityskohteeksi, sillä se sisältää manuaalisia ja toistuvia työvaiheita, joita voisi sujuvoittaa automaatiolla.

Näiden havaintojen perusteella työssä suunniteltiin, toteutettiin ja testattiin Proof of Concept (PoC) -työkalu, jonka tavoitteena on automatisoida valittuja käyttöönottoprosessin vaiheita. Työkalua arvioitiin kontrolloidussa testiympäristössä sekä insinööreiltä saadun palautteen perusteella. Palautteen mukaan työkalu suoritti keskeiset käyttöönottovaiheet tehokkaasti ja tarjoaa vahvan teknisen pohjan tuotantokäyttöön soveltuvan ratkaisun jatkokehitykselle.

Tulokset osoittavat, että tämä ratkaisu voi merkittävästi parantaa käyttöönottoprosessin tehokkuutta, vähentää manuaalista suunnittelutyötä ja tukea teollisuusautomaation jatkuvaa digitalisointia.

Avainsanat: Teollisuusautomaatio, Hajautettu ohjausjärjestelmä, Käyttöönoton automatisointi, Robot Framework, Jenkins Pipeline

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

USE OF ARTIFICIAL INTELLIGENCE IN THIS WORK

Artificial intelligence (AI) has been used in generating this work:

- No
- Yes

The AI tools used in my thesis and the purpose of their use has been described below:

Names of the tools and versions: Microsoft 365 Copilot GPT-5

Purpose of the use: Use of AI aimed to enhance the readability and flow of text by spell checking and improving sentence structure.

Parts in which AI was used: The chapters were checked with AI to fix spelling and improve text quality.

Acknowledgement of risks

I hereby acknowledge, that as the author of this work, I am fully responsible for the contents presented in this thesis. This includes the parts that were generated by an AI, in part or in their entirety. I therefore also acknowledge my responsibility in the case, where use of AI has resulted in ethical guidelines being breached.

PREFACE

This thesis concludes my studies at Tampere University. From University, I would like to thank Professor Jose Luis Martinez Lastra and Luis Enrique Gonzalez Moctezuma for support and guidance throughout the thesis process.

I'm thankful for the target company for giving me the opportunity to write my thesis on this topic. I would also like to thank my thesis instructor Topias Leppänen. His enthusiasm for the topic and support during the thesis process helped keep me motivated and the work focused. I'm also grateful for all the colleagues who participated in this process and guided this project with their help and feedback.

Lastly, thank you to my family and friends who brighten up any day and have supported me throughout my studies. Also, special thanks to my siblings, you make my day. Kiitos!

In Tampere, 30th March 2026

Inka Sorvala

CONTENTS

| | | |
|-------|--|----|
| 1. | Introduction | 1 |
| 1.1 | Background and motivation | 1 |
| 1.2 | Objectives and research questions | 2 |
| 1.3 | Methodology | 3 |
| 1.4 | Thesis outline | 3 |
| 2. | Theoretical background | 4 |
| 2.1 | Industrial automation systems. | 4 |
| 2.2 | Fundamentals of software deployment | 9 |
| 2.2.1 | Continuous Integration/Continuous Delivery | 10 |
| 2.2.2 | Jenkins | 12 |
| 2.3 | Software deployment in industrial automation | 15 |
| 2.3.1 | Automation Application Development life cycle | 15 |
| 2.3.2 | Automating industrial software deployment | 17 |
| 2.4 | Deployment automation tools and methods | 18 |
| 2.4.1 | Vendor specific deployment tools. | 18 |
| 2.4.2 | General deployment tools. | 18 |
| 2.5 | Robot Framework | 19 |
| 3. | Analysis of the deployment process | 27 |
| 3.1 | Current deployment process workflow | 27 |
| 3.2 | Limitations, challenges and possibilities for automation | 31 |
| 4. | Proposed automated workflow | 33 |
| 4.1 | The ideal automated process proposal | 33 |
| 4.2 | Proof of Concept requirements | 38 |
| 4.3 | Automation tool assessment | 40 |
| 5. | Implementation | 44 |
| 5.1 | System overview | 44 |
| 5.2 | Workflow orchestration with Jenkins Pipeline | 46 |
| 5.3 | User interaction | 50 |
| 5.4 | Deployment task execution | 52 |
| 5.5 | Execution reports. | 54 |
| 6. | Tests and results. | 56 |
| 6.1 | Test scenarios and methodology. | 56 |
| 6.2 | Test execution results | 57 |
| 6.3 | Engineer feedback | 63 |

| | |
|-----------------------------------|----|
| 7. Conclusions | 64 |
| 7.1 Further development | 65 |
| References. | 67 |

LIST OF FIGURES

| | | |
|----|--|----|
| 1 | Five-layered architecture of an industrial automation system, modified from [8]. | 5 |
| 2 | Programmable Logic Controller, modified from [9]. | 6 |
| 3 | Diagram of a SCADA system, modified from [8]. | 7 |
| 4 | Architecture of a Distributed Control System, modified from [8]. | 8 |
| 5 | Software Development Life Cycle phases. | 10 |
| 6 | Centralized and distributed VCS schema, modified from [13]. | 11 |
| 7 | Declarative Jenkins Pipeline syntax, modified from [24]. | 13 |
| 8 | Scripted Jenkins Pipeline syntax, modified from [24]. | 14 |
| 9 | Automation application project phases, modified from [32]. | 16 |
| 10 | High-level architecture of Robot Framework [56]. | 20 |
| 11 | Example structure for a Robot Framework project [60]. | 21 |
| 12 | Example of Robot Framework syntax [55]. | 22 |
| 13 | Command line output of test execution [55]. | 24 |
| 14 | Report file of a failed test suite [55]. | 25 |
| 15 | Log file of a failed test suite [55]. | 26 |
| 16 | Full deployment process workflow. | 28 |
| 17 | Productized application generation workflow. | 30 |
| 18 | Ideal full deployment process utilizing automation. | 34 |
| 19 | Ideal application generation process utilizing automation. | 35 |
| 20 | Automated application generation process. | 37 |
| 21 | Automated runtime download process. | 38 |
| 22 | Weighted totals of each tool. | 43 |
| 23 | Overview of Proof of Concept system. | 45 |
| 24 | Robot task execution functions. | 47 |
| 25 | Pipeline post actions section. | 49 |
| 26 | Pipeline Build with Parameters page. | 50 |
| 27 | User input prompt presented upon task failure. | 51 |
| 28 | Setup and teardown keyword definitions. | 52 |
| 29 | Task suite tasks definition. | 53 |
| 30 | Layout of the report summary file. | 55 |
| 31 | Comparison of Pipeline stages execution time for product A and product B. | 58 |

| | | |
|----|---|----|
| 32 | Robot Framework report of product B deployment. | 59 |
| 33 | Robot Framework report of product A deployment. | 59 |
| 34 | Pipeline stage execution times for multiple product deployment. | 60 |
| 35 | Robot Framework report of multiple product deployment. | 61 |
| 36 | Product B Pipeline stages execution times. | 61 |
| 37 | Product B Pipeline validation stage phases with user interaction. | 62 |

LIST OF TABLES

| | | |
|---|--|----|
| 1 | Evaluation criteria weights | 40 |
| 2 | Point scores for each tool | 42 |
| 3 | Pipeline total execution times for product A and product B [minutes] | 58 |

LIST OF SYMBOLS AND ABBREVIATIONS

| | |
|--------|--|
| ADD | Automation Application Development |
| AI | Artificial Intelligence |
| CI/CD | Continuous Integration/Continuous Delivery |
| DCS | Distributed Control System |
| ERP | Enterprise Resource Planning |
| FAT | Factory Acceptance Test |
| FB | Function Block |
| HIL | Hardware-in-the-Loop |
| HMI | Human Machine Interface |
| I/O | Input/Output |
| IA | Industrial Automation |
| MES | Manufacturing Execution System |
| PLC | Programmable Logic Controller |
| PoC | Proof of Concept |
| RPA | Robotic Process Automation |
| RTU | Remote Terminal Unit |
| SCADA | Supervisory Control and Data Acquisition |
| SDLC | Software Development Life Cycle |
| SFC | Sequential Function Chart |
| SIL | Software-in-the-Loop |
| UI | User Interface |
| VCS | Version Control System |
| WAN | Wide Area Network |
| AC_t | Automation Capability score |
| IA_t | Industrial Automation Fit score |

| | |
|--------|------------------------------------|
| IW_t | Implementation Workload score |
| OC_t | Organizational Compatibility score |

1. INTRODUCTION

Industrial automation has become a cornerstone of modern manufacturing, driven by the increased demand on operational efficiency. Since the third industrial revolution, organizations have adopted automation technologies to reduce operational costs, minimize human error and improve efficiency and productivity. [1] In 2022, the global industrial automation market was valued at 205.86 billion USD and is anticipated to reach 395.09 billion USD by 2029 [2]. Within manufacturing, routine tasks not requiring creativity, complex communication or advanced problem solving are increasingly automated [1]. This thesis investigates the potential of automating a labor-intensive phase of an industrial automation application deployment process, with the aim of reducing manual effort and improving overall workflow efficiency.

This chapter consists of four sections. Section 1.1 presents the background and motivation for this thesis. Section 1.2 defines the objectives and research questions. Section 1.3 outlines the research methodology. Section 1.4 describes the structure of the thesis.

1.1 Background and motivation

Software deployment is a critical phase in industrial automation projects. It can be defined as the activity of taking software into use. It consists of configuring the software, deploying it to the target environment and monitoring and verifying the success of the deployment. [3] [4] Despite its importance, deployment is often done manually. This approach is time-consuming, labor-intensive, and requires expert knowledge of both the software and its configuration environment. This increases the likelihood of errors, missed steps and inconsistent results due to human factors. Automating the deployment process can save time and effort of expert employees, thus reducing the cost of deployment. Automation also enables exact repetition across different environments, making the deployment process reliable and easy to test repeatedly before production. [5] [6]

However, automating deployment introduces several challenges. It requires seamless collaboration, coordination and thorough planning between teams to automatically deploy software on a continuous basis. Automating the deployment process is also a large investment, requiring resources, infrastructure, tools, expertise and workload. Rigorous testing strategies are essential to prevent flawed code from entering production. [4] [7]

This thesis explores the potential of automating a manual deployment process in the process industry. The deployment process involves deploying multiple applications into an industrial automation system to configure it according to customer specific requirements. To improve efficiency, certain applications have been productized. Building on these developments, this work seeks to further streamlining the deployment workflow. Generating productized industrial applications in the target environment requires downloading specification files for each individual application into the system. This task is labor-intensive, as projects often include numerous applications, each with distinct specification files that must be imported in the correct order and downloaded into the production environment. When errors arise during the application generation, resolution can be time-consuming and may require restarting the process from earlier phases. Automating the tasks of generating application specification files, validating them, and importing them into the target environment has the potential to streamline the workflow, reduce error rates, and improve overall repeatability.

1.2 Objectives and research questions

The main objective of this thesis is to assess the possibility of automating part of the industrial application deployment process in a web-based Distributed Control System (DCS). The aim is to reduce repetitive, error prone tasks and improve the efficiency of internal production operations. This requires thorough analysis of the existing process. The evaluation of this process focuses on the workflow and the limitations and challenges of automating part of the process. Other objectives include defining an ideal automated deployment workflow and designing a deployment automation tool. In addition to the objectives, the research of this thesis aims to address the following research questions:

1. What are the characteristics and limitations of the current deployment process for automation applications?
2. What would be an ideal deployment workflow utilizing automation?
3. In what ways can different automation tools contribute to designing and implementing this ideal deployment workflow?
4. How can an automatic deployment tool be designed, implemented, and evaluated in practice?

The focus of the thesis is on the current deployment process and the design and partial implementation of an ideal automated deployment process. The first research question requires investigation of the workflow of the deployment process and assessing its limitations. To answer the second question, analysis of the deployment process phases and workflow are evaluated based on what parts are beneficial to automate and what aspects of the deployment process does the automation aim to solve. Automating the deployment

process requires information on different deployment automation tools and an assessment on which tools suit the requirements of the ideal automated deployment workflow. After tool selection, the design, implementation, and evaluation of the automated application generation workflow are carried out through a Proof of Concept (PoC). The PoC is tested using representative automation applications and evaluated through functional tests and engineer feedback.

1.3 Methodology

To gain an understanding of the current deployment process, existing documentations and prior research related to deployment practices is examined. This analysis provides insights into the process, its limitations, and the requirements for automation. In addition, the current process is studied through observing an automation application deployment project. This practical analysis enables the identification of key steps in the process and where automation would be most beneficial. Utilizing multiple methods for acquiring information ensures a thorough investigation of the deployment process from different perspectives, including its constraints and automation requirements.

In addition to the other research methods, a comprehensive review of the literature associated with the research is used as a basis for the theory of this thesis. Relevant articles, books and publications are reviewed to gain an understanding of deployment automation approaches in industrial automation and the tools and methods that can be utilized. This research contributes to the analysis of how automation can be utilized and implemented in the current deployment process.

1.4 Thesis outline

The structure of the thesis is divided into seven distinct sections. Section 1 introduces the thesis topic, motivation and the objectives. Section 2 presents the theoretical background, including industrial automation systems, fundamentals of software deployment, deployment practices in industrial automation, and the tools and methods relevant to deployment automation. Section 3 analyses the current application deployment process and identifies key limitations and opportunities for automation. Section 4 proposes an ideal automated deployment workflow and defines the requirements for the PoC.

Section 5 describes the implementation of the PoC system, including workflow orchestration, automation logic, and reporting mechanisms. Section 6 presents the results of the evaluation, including functional tests of the automated workflow, product deployment scenarios, and feedback from project and development engineers. Finally, Section 7 concludes the thesis and outlines directions for future work.

2. THEORETICAL BACKGROUND

Reliable software deployment is critical in industrial automation environments, where system failures can lead to costly downtime and safety risks. These environments typically incorporate Distributed Control Systems (DCS), Programmable Logic Controllers (PLC), and Supervisory Control and Data Acquisition (SCADA), which depend on precise configurations to enable plant operations. However, deployment in these systems often relies on manual and repetitive tasks, increasing the risk of configuration errors and extended deployment timelines.

This chapter provides the theoretical foundation for understanding deployment automation in industrial contexts. Section 2.1 introduces automation systems and their core components, including DCS, PLC and SCADA systems. Section 2.2 reviews fundamental software deployment practices and methodologies, with emphasis on modern approaches such as DevOps and Continuous Integration/Continuous Delivery (CI/CD). Section 2.3 examines deployment practices in industrial automation and the challenges associated with manual processes. Section 2.4 explores existing tools and methods for deployment automation, including vendor specific solutions, general IT tools, scripting approaches, and Robotic Process Automation (RPA). Finally, Section 2.5 highlights Robot Framework, an RPA tool with potential for automating repetitive tasks.

2.1 Industrial automation systems

Automation system architecture can be described as consisting of five layers: enterprise, supervisor, operator, control and field. This layered structure, illustrated in Figure 1, enables efficient coordination between business-layer operations and real-time process control. The lowest layer, the field layer, consists of sensors, valves, actuators, switches and input/output (I/O) modules that gather process data and execute control actions within the plant. The devices at this layer communicate with the second layer, which is the control layer. The control layer includes PLC, DCS and safety systems. These components process input signals from the field layer, manage alarms, and transmit control parameters to maintain stable operations. The third layer, the operator layer, provides human operators visibility and control through Human Machine Interfaces (HMI) or SCADA systems that display operational data, alarms and production information. [8] [9]

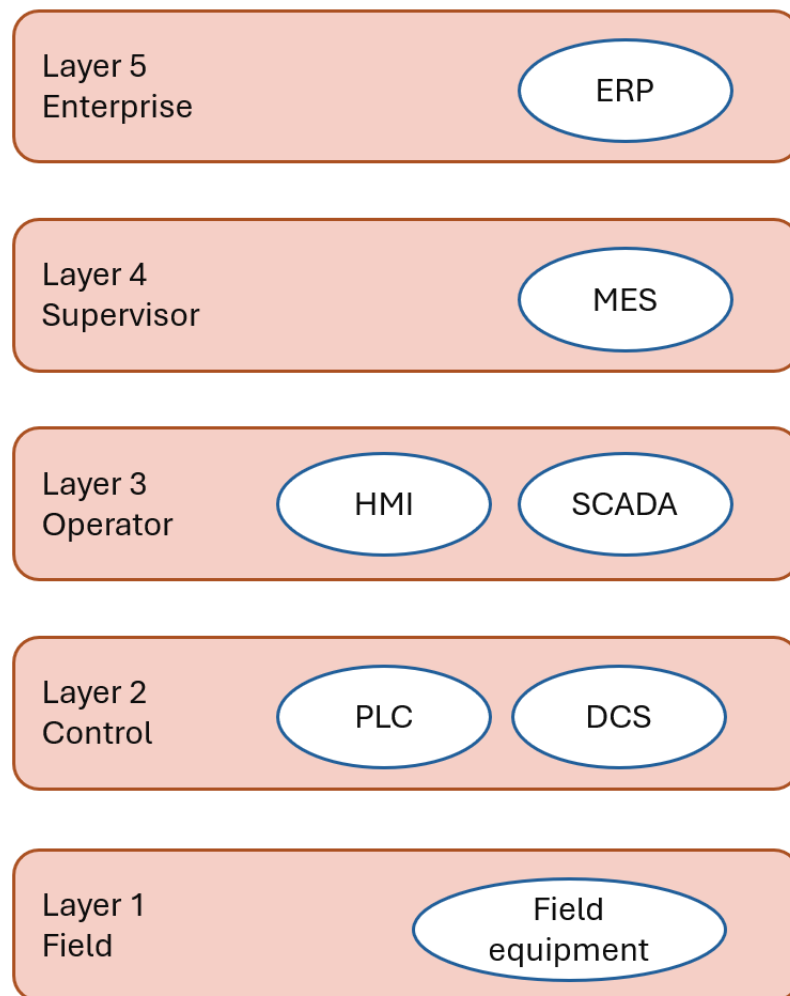


Figure 1. Five-layered architecture of an industrial automation system, modified from [8].

The fourth layer, the supervisory layer, handles plant scheduling, production execution, and performance analysis. These functions are typically executed using Manufacturing Execution Systems (MES). At the top of the hierarchy is the enterprise layer, which integrates business operations through Enterprise Resource Planning (ERP) systems. ERP systems provide information management, such as ordering, billing, shipment and long-term planning. [8] [9]

Programmable Logic Controller

Programmable Logic Controllers are microprocessor-based devices designed to monitor inputs and execute control instructions stored in programmable memory. PLCs can be utilized with DCS and SCADA systems or as stand-alone components. The basic structure of a PLC is illustrated in Figure 2. PLCs process multiple inputs to generate outputs to the process. Typically, PLCs can be used to execute functions such as counting, arithmetic operations, timing, logic and sequencing. [8]

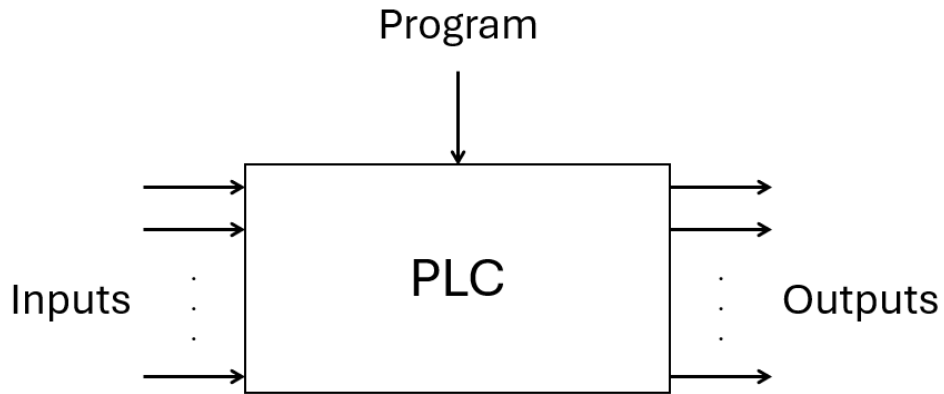


Figure 2. Programmable Logic Controller, modified from [9].

PLCs are widely adopted due to their flexibility for expansion, easy programmability and integrated software and hardware packaged design. They are designed to tolerate strenuous industrial environment conditions, such as humidity, vibration, noise and temperature fluctuations. PLC manufacturers report the mean time between failure of PLC operation is between 20 000 to 50 000 hours, making them an inexpensive, reliable and flexible solution for industrial automation. [8] [9]

Supervisory Control and Data Acquisition

Supervisory Control and Data Acquisition is an industrial control system designed for large processes that spread out geographically hundreds or thousands of kilometers. It is used to remote control and monitor processes from a central location, allowing operators to supervise alarms, operate valves and switches, make parameter changes to controllers and gather system information. SCADA is often used in industries such as gas and water distribution, electric power and environmental monitoring. [8]

A SCADA system typically includes operator consoles, a central host computer, a wide-area telecommunication system and Remote Terminal Units (RTU) as shown in Figure 3. RTUs are located at remote sites to acquire process data and provide control actions to field equipment. They enable real-time updates of field parameters to the operator consoles. The central computer communicates with RTUs through a Wide Area Network (WAN) to convey process values and control commands across the whole process. Operator consoles provide access to the system control, display history data, field data, alarms, trends and reports of the system. These consoles allow operators to monitor real-time data, update configurations, and issue commands to field equipment. [8] [9]

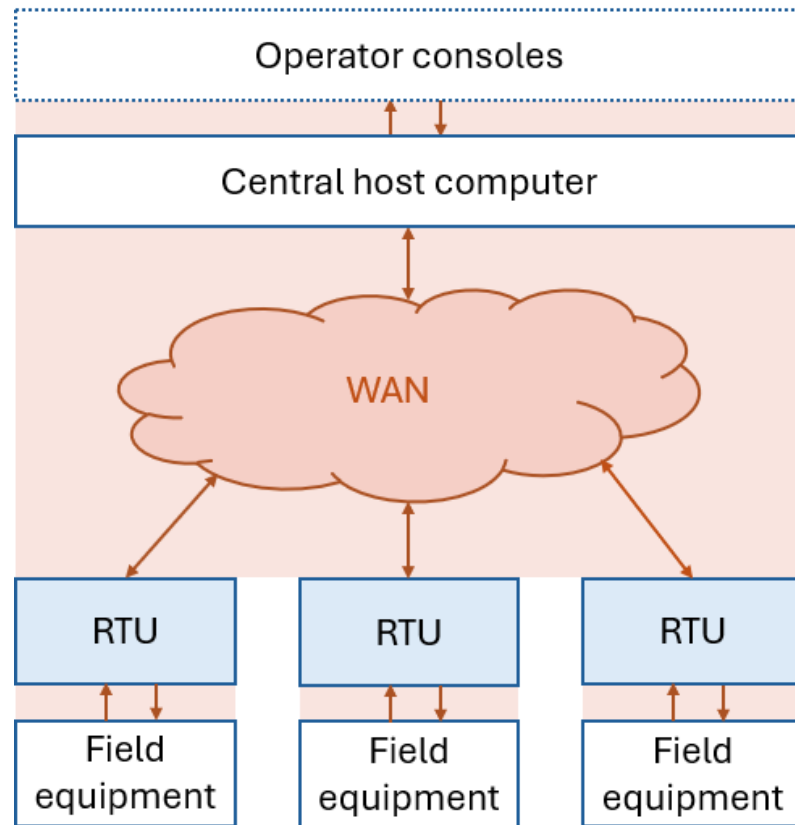


Figure 3. Diagram of a SCADA system, modified from [8].

The core functions of a SCADA system include data procurement, communication, control and visualization. SCADA systems collect large volumes of process data from field devices that must be well displayed, monitored and organized. Operator consoles presents data in graphical displays, diagnostics views, system views as well as industry and site specific displays. They also alert operators of alarms, filter alarms based on operator needs and gather alarm history information for later inspection. Additionally, consoles record system history data, that displays past trends of system parameters. [9]

Distributed Control System

DCS are designed for complex industrial processes requiring numerous control functions, advanced operations, and extensive alarm management. DCS are widely used in industries such as pulp and paper, chemical processing, and power generation. An important advantage of DCS is its ability for system reconfiguration during plant operation, which minimizes operational downtime. [8]

The architecture of a DCS aligns with the layered automation model as illustrated in Figure 4. The main components of a DCS are field devices, controllers, HMI workstations and a process control network. The foundation of a DCS is the process control network where control and input elements communicate. Similarly to SCADA, DCS field devices include

valves, switches, sensors and transmitters that provide process parameter values to the controllers or execute controller instructions. Controllers communicate instructions to field devices and HMI based on field device inputs. The HMI includes operator and application workstations, reporting tools, historians and databases to store data. Process parameters are transmitted to these workstations through the process control network. The data is then stored and displayed graphically to the operator for monitoring. [8]

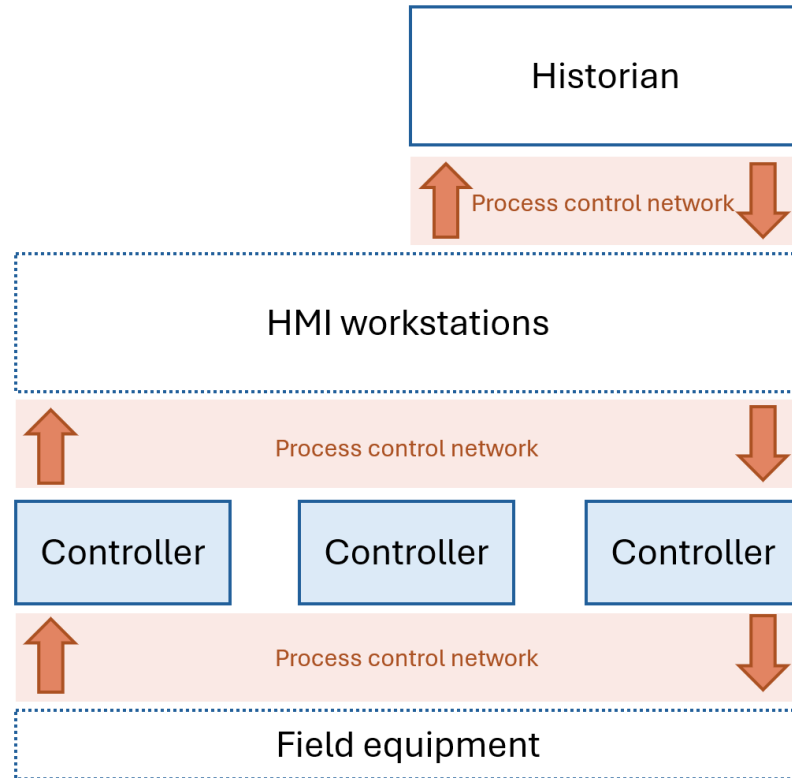


Figure 4. Architecture of a Distributed Control System, modified from [8].

Workstations in a DCS, also referred to as operator displays or consoles, form the primary interface between operators and the process. They are typically located in the plants operator room, where they provide access to history data, alarms, reports, logs, process graphics, and system diagnostics, as well as allow operators to manipulate control parameters. Consoles may be implemented as generic templates or customized according to plant requirements. Customization can involve physical elements, such as buttons, keyboards, and monitors, or visual configurations within the display itself. [8]

In a DCS, controllers are defined by the applications they execute. A single controller can perform multiple functions depending on the applications stored in its memory. Typically, the functionality of the application is defined by utilizing controller libraries. These libraries may be provided by the DCS manufacturer or developed internally by plant automation engineers to meet specific process requirements. Libraries include graphical programming language elements, such as Sequential Function Charts (SFC) and Function Blocks (FB). These interconnected SFC and FB are used to program the functionality of the applica-

tion. Once programmed, the applications are stored in a configuration database, ready to be downloaded through the process control network into the DCS controllers. [8]

Industrial automation applications developed for DCS and PLC controllers are typically engineered as project specific solutions, because their control logic and configurations must reflect the exact requirements of customer processes. This requires applications to be modified according to customer and project specific requirements. These modifications may include customized control logic, tailored FB configurations, or project specific HMI structures. [8]

As industrial applications are often customized to meet customer needs, the applications are not fully productized. In this thesis, productized applications refer to applications that have been developed, standardized, and packaged into products intended to function as reusable templates for project teams. Such applications aim to reduce project development effort by providing a consistent, predefined starting point that can be adapted to customer needs with minimal modification.

2.2 Fundamentals of software deployment

Software deployment is an essential part of the software development process. To understand deployment, it is important to examine its role within the Software Development Life Cycle (SDLC). SDLC includes phases such as requirements gathering, design, coding, testing, and release as illustrated in Figure 5. The requirements phase involves the collection and analysis of application requirements. In the design phase, these requirements are reformed into detailed application specifications. During the coding and testing phases, the specified features are implemented and validated. After successful testing, the software is released. Software deployment is a part of this release phase. Additionally, the final phase of the SDLC includes software maintenance and decommissioning of obsolete systems. [10] [11]

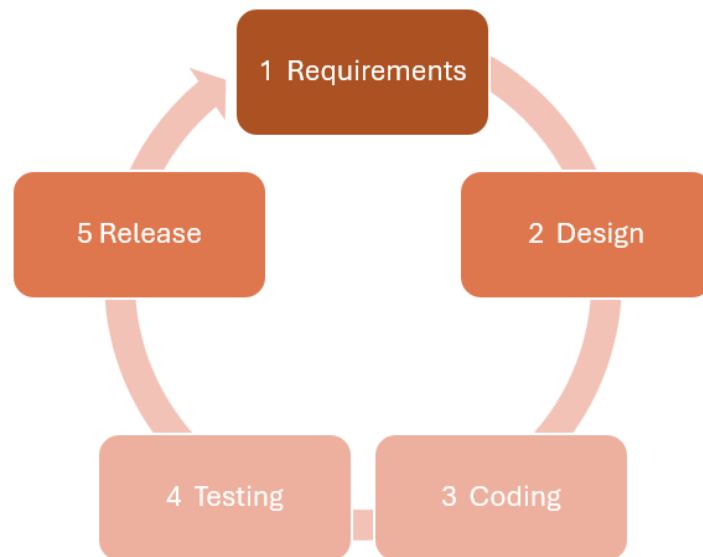


Figure 5. *Software Development Life Cycle phases.*

The growing complexity of systems and demand for rapid update cycles have introduced new challenges in software development. While traditional software deployment strategies were sufficient earlier, in the modern era teams must adopt improved methods to stay competitive in the market. As a solution to this, the development process has evolved to implement automation in to the life cycle of software applications. One of the most influential approaches to modern software deployment is DevOps, which integrates Development (Dev) and Operations (Ops) into a unified process. Implementing these practices in software application projects saves operational costs by enabling fast and frequent deployment of quality software and provides companies competitive advantage. The collaborative culture of DevOps allows developers to produce software that better suits the target environment and enables smoother handovers between development phases. [12] [13] [14]

2.2.1 Continuous Integration/Continuous Delivery

CI/CD is a deployment automation methodology that is often associated with DevOps practices. With CI/CD, the deployment and testing of software is automated enabling faster and more reliable delivery cycles. Automation not only accelerates software delivery, but also ensures consistency and reduces manual errors. Continuous Integration allows developers to efficiently integrate new features into software projects. In order to achieve the benefits of CI practices, developers must frequently merge their software features into a central repository. This triggers an automatic build and testing sequence that verifies the code changes. Continuous Delivery is the next step after software features have successfully passed the CI phase. In the CD phase features are automatically deployed to testing or pre-production environments. Testing in the CD phase is more

comprehensive than in the CI phase, as it considers all system dependencies and tests the application as a whole. After the application has been verified with CI/CD practices it can be deployed to the end environment. Implementing CI/CD practices to the workflow of software development reduces manual labor, human errors and enables faster release cycles. [12] [13] [14]

To implement CI/CD effectively, teams need the right tools. For CI practices, the development team needs a Version Control System (VCS) as a central repository for code changes. As illustrated in Figure 6, VCS can be divided into centralized and distributed systems. Central VCS, have one central repository that every developer is connected to. Distributed VCS have a central repository and distributed local repositories for each developer. The local repositories are copies of the central repository. An example of a distributed VCS is Git, which is an open source distributed version control system created by Linus Torvalds. Git allows developers to develop code in their own repository first and later merge it into the central repository. Developers can also pull the latest changes from the central repository to update their local repository. [13] [15] Studies show that using distributed VCS, instead of centralized, makes developers split commits to smaller more concise units and increases commit frequency [16]. This enables higher quality revisioning and improves team collaboration [16].

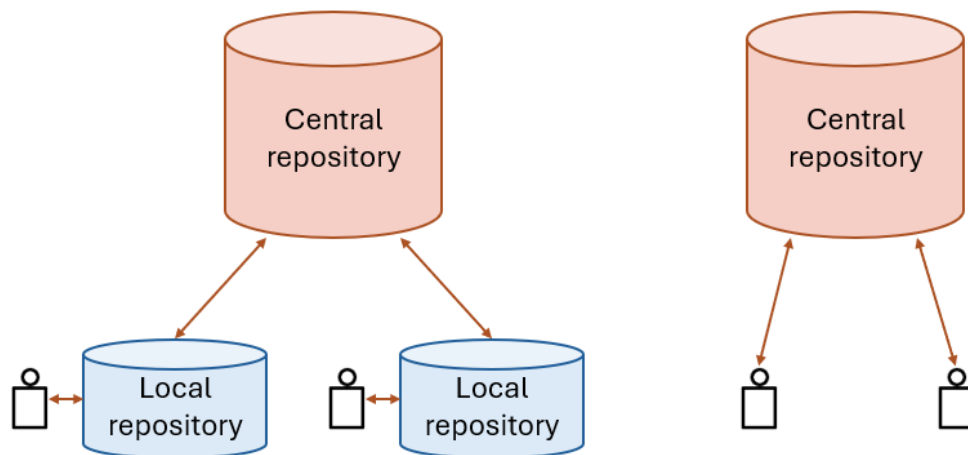


Figure 6. Centralized and distributed VCS schema, modified from [13].

In addition to a suitable VCS, the development team needs an external server for automatic building and testing. This server can either be a cloud server or a local server. Jenkins is an example of a local CI server. Git can be connected to Jenkins so that every commit of code to a certain repository branch triggers an automatic build and test sequence in Jenkins. In Jenkins, developers can create jobs that connect to Git repository branches. Tools such as Jenkins can improve deployment performance compared to manual procedures. As described in [17], a Jenkins-based CI/CD pipeline implementation significantly reduces deployment time, lowers human error rates, and increases consistency through automated testing before deployment. By combining these software

deployment practices and tools, modern software deployment achieves speed, reliability, and adaptability in today's fast-paced software industry. [13] [15]

2.2.2 Jenkins

Jenkins is an open-source, cross-platform CI/CD automation tool that integrates with version control systems and enables control of different stages in the software development process. Jenkins has an extensive community and a large plugin ecosystem, which allows integration with a wide range of external tools. Additionally, Jenkins supports pipeline features, which enable the definition of automated workflows as code. [18] [19]

At its core, Jenkins is an orchestration tool. It does not perform deployment or configuration tasks, but instead coordinates the execution and order of external tools and scripts within an automated workflow. Jenkins is based on a node architecture consisting of Agents and controllers. The controller is the main Jenkins node responsible for managing Agents, organizing workflows and loading required plugins. Agents are machines or containers that execute jobs in isolated environments. Job execution requests are distributed by the controller to available Agents, enabling workload balancing. This architecture also allows parallel job execution, as a single controller can manage multiple Agents with different execution environments. [19] [20]

Jenkins Pipeline

A Jenkins Pipeline defines an automated workflow as code. Pipelines are commonly connected to version control repositories, which enables automated execution on code changes and supports continuous delivery practices. The implementation of a Pipeline is defined in a Jenkinsfile, which is a text-based script describing the workflow. The Jenkinsfile is typically stored alongside the project source code and maintained under version control. [21] [22]

Jenkinsfile syntax can be divided into declarative and scripted syntax. Declarative syntax provides a more limited but structured format with a predefined layout, whereas scripted syntax is based on Groovy and allows more flexible control flow. Groovy is a dynamic, Java syntax compatible language created as a complementary counterpart to Java with a focus on extensibility and rapid innovation [23]. As illustrated in Figure 7, a declarative Pipeline is enclosed in a `pipeline` block. The `agent` section defines where the Pipeline or an individual stage is executed. Pipeline execution is divided into stages, each containing one or more steps that define the actions performed. [22]

```

1 pipeline {
2   agent any
3   stages {
4     stage('Build') {
5       steps {
6         sh 'make'
7       }
8     }
9     stage('Test') {
10      steps {
11        sh 'make check'
12        junit 'reports/**/*.xml'
13      }
14    }
15    stage('Deploy') {
16      steps {
17        sh 'make publish'
18      }
19    }
20  }
21  post {
22    always {
23      junit '**/target/*.xml'
24    }
25    failure {
26      mail to: team@example.com, subject: 'The Pipeline failed :('
27    }
28  }
29 }

```

Figure 7. Declarative Jenkins Pipeline syntax, modified from [24].

Stages are typically used to represent distinct phases of a workflow, such as build, test, and deployment. A declarative Pipeline may also define additional sections such as environment, parameters, options, and post. The environment section defines environment variables available during execution, while the parameters section specifies user-provided inputs that are requested when the Pipeline is triggered. The options section allows configuring general Pipeline behavior, such as execution timeouts, console output formatting, and build-level execution constraints. The post section defines actions that are executed after Pipeline or stage completion. This is where the Pipeline failure handling is defined with different post conditions. Post conditions include `always`, `success`, `failure`, `unstable`, and `changed`, which correspond to different Pipeline statuses. These conditions allow additional steps such as cleanup actions or logging to be executed based on the outcome of Pipeline execution. [24] [25]

Scripted syntax resembles a traditional scripting language and provides greater flexibility than declarative syntax when defining Pipeline behavior. An example of a scripted syntax

is presented in Figure 8. Scripted Pipelines are executed sequentially following the order of the script, control flow is implemented using standard constructs such as `if` and `else`. Unlike declarative Pipelines, scripted Pipelines do not provide a structured `post` section. As a result, error handling and cleanup actions must be implemented explicitly within the Pipeline logic, typically using `try`, `catch` and `finally` constructs. [24] [25]

```

1  node {
2      stage('Build') {
3          sh 'make'
4      }
5      stage('Test') {
6          try {
7              sh 'make check'
8          } finally {
9              junit '**/target/*.xml'
10         }
11     }
12     stage('Deploy') {
13         sh 'make publish'
14     }
15     stage('Example') {
16         if (env.BRANCH_NAME == 'main') {
17             echo 'I only execute on the main branch'
18         } else {
19             echo 'I only execute elsewhere'
20         }
21     }
22 }

```

Figure 8. Scripted Jenkins Pipeline syntax, modified from [24].

Jenkins uses Pipeline steps and plugins to integrate infrastructure automation tools. This is typically done with Agents that provide the necessary runtime environment and credentials. As pipelines consist of stages, they allow processes to be divided into distinct phases for execution. This enables the logical separation of infrastructure and environment provisioning, configuration, deployment, and cleanup within a single Pipeline workflow. [22] [26]

The Input Step plugin provides functionality that allows a Pipeline to pause and wait for user input. In addition to approving or canceling Pipeline execution, the Pipeline can request input parameters from the user. Pipeline stages can also be executed conditionally based on parameters or execution state. Together, these features enable the construction of partially automated workflows, where user interaction is introduced at predefined points during Pipeline execution. [27] [28]

2.3 Software deployment in industrial automation

Deployment of software in industrial environments has different requirements than traditional software deployment. In industrial processes software is interconnected with application specific hardware, communication channels and process management systems. Industrial applications are often dedicated to specific and complex process functions. Deployed software might communicate with software systems from other vendors or with specialized systems of individual applications. This makes software deployment in industrial settings not as flexible or adaptable as in traditional IT environment. Industrial environments require software to be deployed across various devices with different dependencies. In industrial processes software reliability and availability are fundamental. In critical processes software failures can cause discontinuations. This means deployment must be completed without interruptions to process execution at a defined operational downtime or while the process is running. [29] [30] [31]

2.3.1 Automation Application Development life cycle

Deployment can be characterized as a part of the Automation Application Development (ADD) life cycle. As illustrated in Figure 9, the life cycle of an automation application project is divided into four principal phases: requirements and design, development, testing and deployment and commissioning. The first phase is where the application requirements are gathered. This includes detailed descriptions of required devices, connections and the relationships of process elements. The first phase is an iterative phase where the requirements and design of the system are clarified in cooperation with the customer. The second phase is where the main components of the automation application are developed. The application control and HMI are developed by a development team according to the requirements gathered in the first phase. [32]

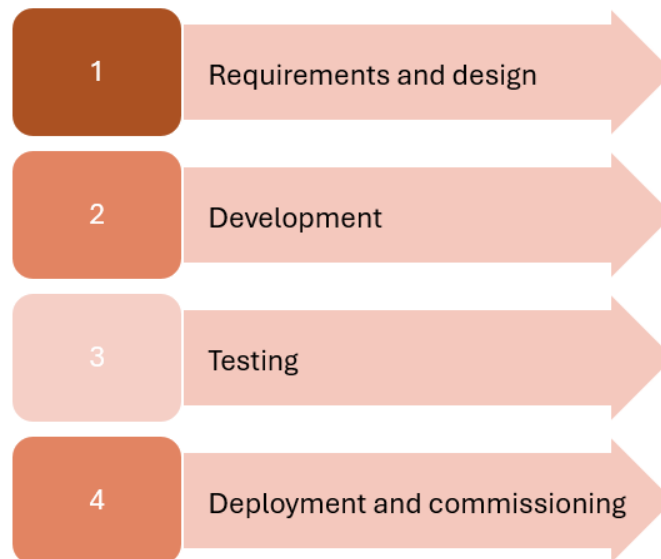


Figure 9. Automation application project phases, modified from [32].

After the development phase the system is ready to be tested. Thorough testing is the foundation of an automation application development project. Typically, testing in ADD is a manual process due to the lack of a unified framework for testing industrial controller code. The testing phase includes developer written module tests, integration tests as well as Factory Acceptance Testing (FAT), which is the acceptance testing performed to verify that a system meets its functional and design requirements before delivery to the customer [33]. Module tests and integration tests are performed to confirm correct functionality and connections inside the developed application, to ensure the system works as expected during execution with the final system configurations. [32]

The final phase is the commissioning of the system. Because of the nature of critical industrial application processes the commissioning phase is typically divided into virtual commissioning and actual commissioning phase. In the virtual commissioning phase the application is tested in a virtual environment that simulates the final process system. In the actual commissioning phase, the system is installed to the customer plant and tested with all final system connections and real production data. Even though most of the project development has been done in the development phase, some system refinement is done at the commissioning phase. This includes fine tuning the system according to customer requirements and fixing possible connection or control issues. The commissioning phase is ready when customer and developer company agree that the system is ready. [32]

In modern practice, virtual commissioning often utilizes methods such as Software-in-the-Loop (SIL) and Hardware-in-the-Loop (HIL), to reduce deployment risk before customer equipment integration. In SIL, the control algorithms are executed entirely in a virtual simulation environment, enabling early functional verification of logic without requiring

physical hardware. In HIL, the real controller hardware is connected to a simulated plant environment for validation. Together with digital twin models, they allow engineers to validate system connections and test control algorithms in a variety of scenarios. This accelerates the deployment and commissioning process by allowing functionality tests and system fine tuning before customer hardware is available. [34] [35]

2.3.2 Automating industrial software deployment

Traditionally, deployment in industrial contexts is a manual process. PLC-based control applications typically rely on proprietary tools, which limits integration with modern version control systems. As these tools might not support external version control systems, tracking different code version updates is generally not implemented with PLC systems. [36] With the rise of interconnected devices in industrial environments, the amount of communication software in these systems is increasing [37]. This highlights a growing need to utilize traditional software development and maintenance methodologies in industrial settings.

However, incorporating such methodologies into industrial settings is challenging due to system heterogeneity. In addition to the need to integrate distinct system components, industrial systems must remain compatible with elements deployed decades earlier due to long life cycles of industrial systems. Compatibility requirements in concurrence with the high expenses of suspending industrial processes make customers hesitant to update their processes to new solutions. [37] Therefore, deployment automation can lower customers perceived investment risk, smooth the transition to updated systems, and enable both faster bring-up of new systems and more cost-efficient updates to existing applications.

In answer to the rising need for new techniques in industrial software handling, automated software deployment solutions have been investigated and implemented in industrial settings. Studies of implementing DevOps and CI/CD in industrial environments presented in [36], [38] and [39] show how the implementation of traditional software engineering tools can save time in industrial software deployment. The solution implemented in [36] proposed an alternative workflow to the traditional development and deployment of PLC applications. The original workflow included editing code inside vendor PLC software, manual testing with hardware and project version control, which was handled completely by the project developer maintaining versions in hardware test setup and a version control system [36]. The new solution included an external version control system for the project's PLC input files and automated testing with hardware via scripts [36]. The study does not implement automatic download of PLC software, but proposes that the new workflow saves time and effort in project development [36]. The study of DevOps practices for server-based industrial software applications found a decrease of 90% in the time

used for deployment with the implementation of a DevOps workflow [38]. Automation of laborious configuration phases in the application deployment was the key contributor to these time savings [38]. A framework for automatic building, testing and deploying software presented in [39] proposes the benefits of faster and more reliable deployment of software. The study also concludes that in order for industrial organizations to remain competitive, they need to implement modern solutions such as DevOps and CI/CD in to their development practices [39].

2.4 Deployment automation tools and methods

Several approaches exist for deployment automation, including vendor specific solutions and general-purpose tools. Vendor specific tools are provided to work with vendor specific components, such as plant PLCs. General tools can be implemented into any process. As outlined in Section 2.3, the key aspects of automating deployment in industrial environments is to implement industrial DevOps practices to the deployment workflow. The tools presented in this chapter facilitate industrial DevOps practices and enable the automatization of software application deployment in industrial settings.

2.4.1 Vendor specific deployment tools

One example of a vendor specific tool is Siemens Totally Integrated Automation (TIA) Portal for plant operations. It includes a tool for simulating plant HMI and control operations for testing before deployment. [40] TIA Portal also has add-ins that connect with version control systems such as Git for industrial code management and tracking. [41] Another example of vendor tools is Rockwell FactoryTalk AssetCentre designed for automation asset management. [42] FactoryTalk AssetCentre provides a comprehensive version control system for application and configuration code. [43]

2.4.2 General deployment tools

Examples of general and more traditional IT software tools include Azure DevOps, Gitlab CI/CD, Ansible and JFrog Artifactory. Azure DevOps and Gitlab CI/CD enable automatic software building, testing and deploying as well as version control. Both tools are designed to implement CI/CD principles and have integrated modern generative Artificial Intelligence (AI) to their processes. [44] [45] Ansible is an open source software application designed for system configuration, continuous software deployment and workflow orchestration. It is based on a simple language called Ansible Playbooks, which can be used to automate tasks, such as ensuring a system stays in a desired state. Ansible also supports rolling updates that enable software to be updated without system downtime. [46] [47] An example of a software artifact management systems is JFrog Artifactory. Software arti-

facts are the interconnected building blocks of a software application. The term software artifact refers to different packages, binaries, containers, releases and configuration files. The purpose of JFrog Artifactory is to be a central repository management system for software artifacts enabling enterprise wide software development life cycle management. [48] [49]

Scripting can be utilized in deployment automation to automate manual deployment phases. Deployment scripts define the sequence of tasks required to deploy an application, enabling consistent execution across environments. [50] [51] These scripts can be developed with various scripting languages, such as Python or PowerShell [51] [52]. Python is an open source high level programming language with extensive libraries that support automation and scripting tasks [52]. PowerShell Desired State Configuration (DSC) is a configuration platform used in [51] to automate deployment tasks.

RPA is an automation solution that is based on software robots that execute tasks that are traditionally done by humans in a software system. The main benefits of RPA are increased operational efficiency and reduced human errors. RPA is typically utilized to automate internal administrative tasks in fields such as banking and insurance. It is best suited for tasks with a clear defined scope that are repetitive and occur frequently. Although the adoption of RPA in industrial engineering has been limited, the characteristics of RPA and the growing integration of software technologies in industrial automation highlight the potential benefits of implementing RPA in industrial processes. [53] [54]

2.5 Robot Framework

Robot Framework is an example of an RPA tool with significant potential for deployment automation. Robot Framework, primarily known as a test automation tool, is an open source python-based automation framework. It uses a simple keyword driven syntax and has a modular architecture, enabling easy implementation even with complex software applications. Its extensive library ecosystem and support for custom extensions make it suitable for automating complex software interactions and repetitive tasks [55] [56]

Although Robot Framework is often associated with testing, its modular architecture and keyword-based implementation logic allows it to be used as a general automation framework. Documented case studies on task automation remain limited, but the framework has developed to support general automation. Since version 3.1 Robot Framework includes an official task syntax, allowing the development of task focused implementations. Combined with its library ecosystem, keyword-driven approach and scalability, this update formally positions Robot Framework as a tool capable of supporting task automation in workflows such as web interface operations and data processing. [57] [56]

Architecture

The high-level architecture of Robot Framework is illustrated in Figure 10. Test data includes all test or task implementations and resource files with keywords and variables. When a test is launched the framework reads the test data, and calls keywords from test libraries through the test library API (Application Programming Interface). The API executes these keywords and creates logs and reports of the execution. This enables the framework itself to not be directly in contact with the target environment, but by utilizing keywords to execute the actions in the system under test. If the system under test cannot be accessed directly with libraries, additional tools can be implemented to enable testing. [56] [58]

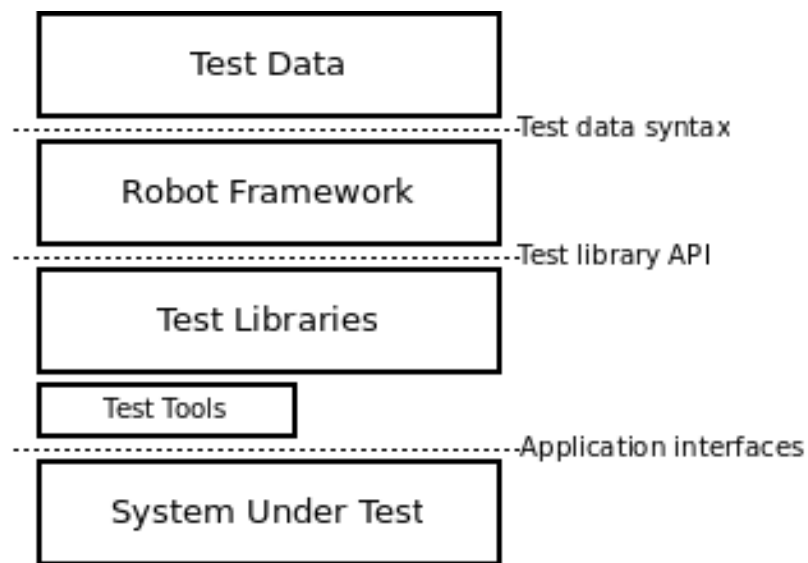


Figure 10. High-level architecture of Robot Framework [56].

Figure 11 illustrates a simple example project structure, with tests and resources. The contents of the project file structure vary between projects. Typically, a Robot Framework project contains `requirements.txt` for different dependencies, `README.md` to describe the project and `.gitignore` to prevent unwanted files and folders from getting committed to git. Additionally the structure contains at least a tests and a resources folder. Tests include all test suite files required by test execution. The resources folder contains all reusable keywords and variables that are utilized throughout the project. These can be defined in `.resource` files or in custom python files. [59] [60]

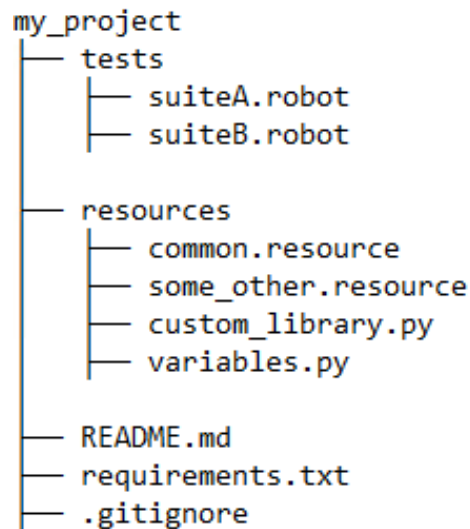


Figure 11. Example structure for a Robot Framework project [60].

Test suites and test cases

Test suites combine defined test cases into a suite that provides test execution with a clear structure. This structure is illustrated in Figure 12. The suite file is divided into four sections: settings, variables, test cases and keywords. In the `*** Settings ***` section all necessary libraries, resource files and variable files are imported. The section can also contain documentation to describe the suite file and metadata to attach additional suite information. The `*** Variables ***` section defines all suite variables used inside the suite file, such as constant values or global variables. The `*** Test Cases ***` section contains all of the suite tests. The definition of a test case typically includes documentation, tags and keywords. Tags are used to classify test cases, include and exclude tests from execution and collect test statistics. The last section contains suite specific keywords composed from the low-level keywords defined in resource and library files. These keywords can be utilized only inside the suite file, and typically include definitions for case specific actions or customized setup and teardown keywords. Setup and teardown keywords can be utilized to execute actions at the start and the end of suite or test execution. [55] [61]

Figure 12 has example definitions that represent a typical test case syntax. Test cases are defined with a test name, followed by indented rows of settings and keywords. The settings can include documentation and tags, as well as additional settings such as custom setup and teardown, template keyword and a test case timeout. After the settings the required keywords are called with their respective arguments. Test case execution starts with a test setups and then continues to execute each keyword in order, followed by a test teardown. [55]

```

Run Suite | Debug Suite | Load in Interactive Console
1  *** Settings ***
2  Documentation      Example file using the space separated format.
3  Library            OperatingSystem
4
Load in Interactive Console
5  *** Variables ***
6  ${MESSAGE}        Hello, world
7
8  *** Test Cases ***
Run | Debug | Run in Interactive Console
9  My Test
10     [Documentation]  Example test.
11     [Tags]           test_tag
12     Log              ${MESSAGE}
13     My Keyword       ${CURDIR}
14
Run | Debug | Run in Interactive Console
15  Another Test
16     [Documentation]  Another example test.
17     [Tags]           another_test_tag
18     Should Be Equal  ${MESSAGE}    Hello, world!
19
20  *** Keywords ***
Load in Interactive Console
21  My Keyword
22     [Documentation]  Example keyword.
23     [Tags]           keyword_tag
24     [Arguments]     ${path}
25     Directory Should Exist  ${path}

```

Figure 12. Example of Robot Framework syntax [55].

The keywords that test cases utilize can be from imported resource files and libraries or defined in the `*** Keywords ***` section in the suite file. Keywords have a similar syntax to test cases, and can also include the same settings as the tests. In addition to these settings, keywords typically expect arguments. With custom keywords these arguments can be defined in the keyword settings section. All keywords should be uniquely named to prevent confusion between keywords during execution. [55]

With the support for task-based development suite files can also contain a `*** Tasks ***` section. The task section would replace the `*** Test Cases ***`, since Robot Framework does not allow both tasks and tests in the same suite. The suite will then become a task suite, which will only change the naming of tests to tasks and not effect the suite functionality in any way. [62]

Libraries

There are three types of libraries in Robot Framework, standard, external and custom libraries. Standard libraries are provided by Robot Framework and contain extensive

keywords that can be utilized with different test cases, such as `OperatingSystem`, that enables execution of operating system level tasks, and `Screenshot`, that enables capturing and storing of screenshots. [63] External libraries are not installed with the core framework and need additional installation. Various different libraries implemented and maintained by the Robot Framework open source community can be utilized for test and task development. Additionally, Robot Framework supports custom libraries that can be developed to implement case specific keywords. [55] [64]

An example of an external library created by the Robot Framework community is Selenium. Selenium is a web testing library, that contains several low-level keywords to interact with a web-based application. It uses the Selenium WebDriver module to launch and control a browser. Selenium keywords recognize elements from the browser with predefined locator elements that are passed as keyword arguments. Selenium supports different locator strategies such as `id`, `name`, `class`, `css` or `xpath`. These strategies are used to match the locator to the correct web element and successfully execute the desired actions defined in the keyword. [65]

Examples of Selenium Library keywords are `Open Browser`, `Create WebDriver`, `Click Element`, `Capture Page Screenshot` and `Page Should Contain Element`. `Open Browser` and `Create Webdriver` launch a Selenium WebDriver instance that creates a window. Windows contain all website content. The keyword `Page Should Contain Element` checks that the specified locator element exists on the current window and the `Click Element` keyword mouse clicks the specified locator element. [65] The extensive volume of web testing keywords in the Selenium library enables a quick implementation of test cases for different web-based applications. Although, typically it is recommended to create customized libraries that utilize Selenium keywords, since they are low-level and require implementation specific locators. [66]

Execution

Test cases can be executed from the command line, which offers various configuration options. The simplest way to run a test suite is by calling `robot my_suite.robot`. Individual test cases can also be selected by test name, suite name or tag name. During execution, Robot Framework processes each test and its keywords in the order they appear in the suite file. Each run produces a command line output, a log file, a report file and an output file. Figure 13 presents an example of the command line output generated during execution. [55]

```

=====
Example test suite
=====
First test :: Possible test documentation                | PASS |
-----
Second test                                           | FAIL |
Error message is displayed here
=====
Example test suite                                    | FAIL |
2 tests, 1 passed, 1 failed
=====
Output:  /path/to/output.xml
Report:  /path/to/report.html
Log:     /path/to/log.html

```

Figure 13. Command line output of test execution [55].

As illustrated in Figure 13, the command line output lists all executed suites and cases together with their statuses. Case and suite statuses can be either PASS, FAIL or SKIP. When a test passes it gets the PASS status, but in order for a test suite to pass, all tests in the suite must pass, otherwise the suite will fail. Typically, a test failure can be caused by a keyword failing inside the test. Tests can also be skipped, either by specifying on the command line to skip a certain test, or if tests are configured to skip if previous conditions fail. A test suite can also be skipped, if all tests inside the suite are skipped, or the suite contains no tests. [55]

During execution, Robot Framework also generates HTML reports and logs. These files provide structured representations of the run and allow deeper inspection of the results. Figure 14 presents an example of a failed suite report, which summarizes execution information such as elapsed time and the counts of passed, failed, and skipped tests [55].

QuickStart Report

Generated
20220323 10:16:29 UTC+02:00
3 years 304 days ago

LOG

Summary Information

Status: 1 test failed

Start Time: 20220323 10:16:28.163

End Time: 20220323 10:16:29.428

Elapsed Time: 00:00:01.265

Log File: [log.html](#)

Test Statistics

| Total Statistics | Total | Pass | Fail | Skip | Elapsed | Pass / Fail / Skip |
|------------------|-------|------|------|------|----------|---|
| All Tests | 5 | 4 | 1 | 0 | 00:00:01 | <div style="width: 100%; height: 10px; background: linear-gradient(to right, green 80%, red 80%, black 100%);"></div> |

| Statistics by Tag | Total | Pass | Fail | Skip | Elapsed | Pass / Fail / Skip |
|-------------------|-------|------|------|------|----------|---|
| database | 1 | 1 | 0 | 0 | 00:00:00 | <div style="width: 100%; height: 10px; background-color: green;"></div> |
| example | 4 | 3 | 1 | 0 | 00:00:01 | <div style="width: 100%; height: 10px; background: linear-gradient(to right, green 75%, red 25%, black 100%);"></div> |
| quickstart | 5 | 4 | 1 | 0 | 00:00:01 | <div style="width: 100%; height: 10px; background: linear-gradient(to right, green 80%, red 20%, black 100%);"></div> |
| smoke | 4 | 3 | 1 | 0 | 00:00:01 | <div style="width: 100%; height: 10px; background: linear-gradient(to right, green 75%, red 25%, black 100%);"></div> |
| variables | 1 | 1 | 0 | 0 | 00:00:00 | <div style="width: 100%; height: 10px; background-color: green;"></div> |

| Statistics by Suite | Total | Pass | Fail | Skip | Elapsed | Pass / Fail / Skip |
|---------------------|-------|------|------|------|----------|---|
| QuickStart | 5 | 4 | 1 | 0 | 00:00:01 | <div style="width: 100%; height: 10px; background: linear-gradient(to right, green 80%, red 20%, black 100%);"></div> |

Test Details

All
Tags
Suites
Search

Suite:

Test:

Include:

Exclude:

[Help](#)

Figure 14. Report file of a failed test suite [55].

In the report view, success and failure states are clearly distinguished by background color which is green if all of the tests passed, red if any test failed and yellow if all tests were skipped. While the report give a high-level overview, the log file illustrated in Figure 15, offers more detailed information about each executed test and keyword, including status, timing, and keyword names. [55].

QuickStart Log

Generated
20220323 10:16:29 UTC+02:00
3 years 304 days ago

REPORT

Test Statistics

| Total Statistics | | | | | | | |
|---------------------|-------|------|------|------|----------|--|--|
| | Total | Pass | Fail | Skip | Elapsed | Pass / Fail / Skip | |
| All Tests | 5 | 4 | 1 | 0 | 00:00:01 | <div style="width: 100%;"><div style="width: 80%;"></div></div> | |
| Statistics by Tag | | | | | | | |
| | Total | Pass | Fail | Skip | Elapsed | Pass / Fail / Skip | |
| database | 1 | 1 | 0 | 0 | 00:00:00 | <div style="width: 100%;"><div style="width: 100%;"></div></div> | |
| example | 4 | 3 | 1 | 0 | 00:00:01 | <div style="width: 100%;"><div style="width: 75%;"></div></div> | |
| quickstart | 5 | 4 | 1 | 0 | 00:00:01 | <div style="width: 100%;"><div style="width: 80%;"></div></div> | |
| smoke | 4 | 3 | 1 | 0 | 00:00:01 | <div style="width: 100%;"><div style="width: 75%;"></div></div> | |
| variables | 1 | 1 | 0 | 0 | 00:00:00 | <div style="width: 100%;"><div style="width: 100%;"></div></div> | |
| Statistics by Suite | | | | | | | |
| | Total | Pass | Fail | Skip | Elapsed | Pass / Fail / Skip | |
| QuickStart | 5 | 4 | 1 | 0 | 00:00:01 | <div style="width: 100%;"><div style="width: 80%;"></div></div> | |

Test Execution Log

```

- SUITE QuickStart 00:00:01.265
  Full Name: QuickStart
  Source: /Users/jth/Code/QuickStartGuide/QuickStart.rst
  Start / End / Elapsed: 20220323 10:16:28.163 / 20220323 10:16:29.428 / 00:00:01.265
  Status: 5 tests total, 4 passed, 1 failed, 0 skipped
  + SETUP Clear login database 00:00:00.001
  + TEST User can create an account and log in 00:00:00.151
  + TEST User cannot log in with bad password 00:00:00.147
  + TEST User can change password 00:00:00.299
  - TEST Invalid password 00:00:00.451
    Full Name: QuickStart.Invalid password
    Tags: example, quickstart, smoke
    Start / End / Elapsed: 20220323 10:16:28.822 / 20220323 10:16:29.273 / 00:00:00.451
    Status: FAIL
    Message: Expected status to be 'Creating user failed: Password must be a combination of lowercase and uppercase letters and numbers' but was 'SUCCESS'
    + KEYWORD Creating user with invalid password should fail abCD5, ${PWD INVALID LENGTH} 00:00:00.073
    + KEYWORD Creating user with invalid password should fail abCD567890123, ${PWD INVALID LENGTH} 00:00:00.072
    + KEYWORD Creating user with invalid password should fail 123DEFG, ${PWD INVALID CONTENT} 00:00:00.072
    - KEYWORD Creating user with invalid password should fail abcd56789, ${PWD INVALID CONTENT} 00:00:00.085
      Start / End / Elapsed: 20220323 10:16:29.040 / 20220323 10:16:29.125 / 00:00:00.085
      + KEYWORD LoginLibrary.Create User example, ${password} 00:00:00.080
      - KEYWORD LoginLibrary.Status Should Be Creating user failed: ${error} 00:00:00.003
        Start / End / Elapsed: 20220323 10:16:29.121 / 20220323 10:16:29.124 / 00:00:00.003
        10:16:29.124 FAIL Expected status to be 'Creating user failed: Password must be a combination of lowercase and uppercase letters and numbers' but was 'SUCCESS'.
      + KEYWORD Creating user with invalid password should fail AbCdEFGH, ${PWD INVALID CONTENT} 00:00:00.072
      + KEYWORD Creating user with invalid password should fail abCD56+, ${PWD INVALID CONTENT} 00:00:00.074
      + TEARDOWN Clear login database 00:00:00.002
    + TEST User status is stored in database 00:00:00.154
  
```

Figure 15. Log file of a failed test suite [55].

3. ANALYSIS OF THE DEPLOYMENT PROCESS

The process described in this chapter is the deployment of automation applications in a DCS within the process industry. These applications are designed to control runtime parameters in production environments. The current deployment workflow is new, as the company recently began utilizing a fully web-based DCS. The introduction of a state-of-the-art system provides an opportunity to review the current deployment workflow and explore potential areas for automation to improve efficiency. The contents of this chapter are based on discussions with production and design professionals and a study of company-wide process documentation on deployment. A field study of an ongoing application deployment project was also conducted as research for the process.

The chapter is divided into two main sections. Section 3.1 outlines the current deployment workflow and details the application generation process. Section 3.2 examines the challenges and limitations related to the existing manual process and highlights key opportunities for automation.

3.1 Current deployment process workflow

Deploying automation applications involves several stages, illustrated in Figure 16. This workflow ensures a structured progression from customer order to final delivery. Typically, a single project engineer manages most deployment tasks, regardless of project size. This includes specification and application generation, execution of FAT phases, and final plant installation and commissioning. Additional personnel may be required in the FAT phases or plant installation and commissioning, depending on project complexity. Other stakeholders, such as salesperson, system engineer, integrator and customer, also participate at different stages.

The process begins with obtaining the order Bill Of Materials (BOM) from the sales team. This document specifies the products included in the order. Once the order details are confirmed, the project engineer defines the product generation files and specifies any non-standard applications. Majority of the applications are productized, which results in a more standardized and straightforward specification process. However, most projects also include non-standard applications that require close collaboration with the customer and must be engineered and integrated into the project environment. For complex prod-

ucts specifications are created with an automatic specification generation tool, which produces the required product generation files based on project specific inputs. Simpler products can be manually defined during the application generation phase.

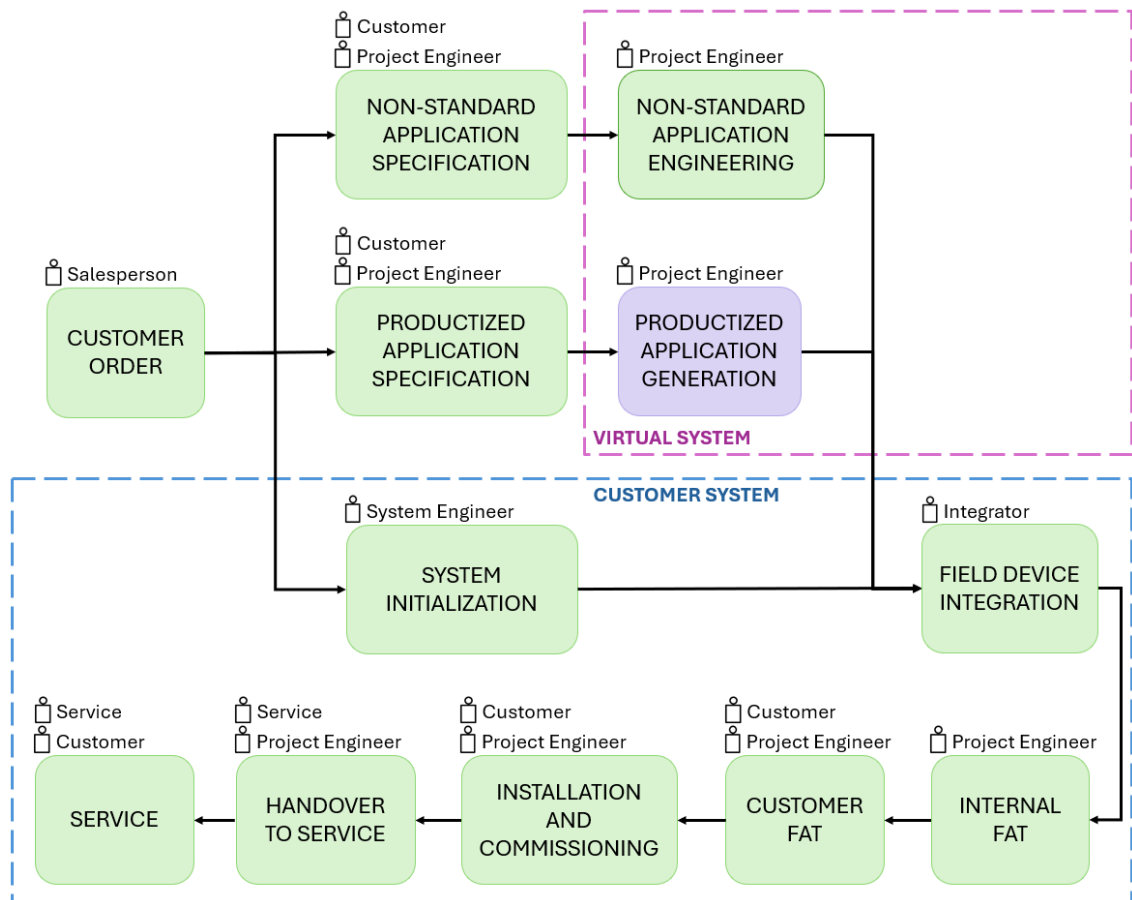


Figure 16. Full deployment process workflow.

After the application specifications are ready, the next stage is the productized application generation phase. Application generation starts with initializing a virtual environment for the project. It is not connected to any hardware and is used for early implementation of project products while the real customer system is initialized. The project engineer then proceeds with generating and deploying the product applications in the virtual environment and incorporating customer specific modifications. This phase must be completed before the system engineer hands over the customer system to the project. System initialization happens simultaneously to the specification and application generation processes. This phase includes initializing customer system, hardware setup and software configuration as well as network integration. Connectivity tests are performed by system engineers to ensure compatibility with system specifications. Once initialization is complete, the environment from the virtual system is transferred to the customer system.

After the customer system is configured, field device integration can start. Field device integration involves hardware component connections. Once integrators have verified

hardware functionality, the project engineer performs internal and customer FAT. In internal FAT system connections and applications are tested with real hardware or through simulation. In customer FAT the system is tested with customer specific hardware configurations with the customer participating in the process. After successful FAT, the system is installed and commissioned at the customer site. This stage includes hardware installation, system tests, and verification of production readiness. The project engineer performs calibrations, mechanical adjustments as well as test runs the system. While the process is running, parameters are fine tuned. Once these adjustments are complete, the system is ready for production and handed over to service operations, marking the end of the deployment process.

Productized application generation

Generating productized applications is a fundamental step in the deployment process. It is where the basis of the applications are created and configured for customers. The process consists of multiple phases, which include: virtual environment initialization, library import, application generation, user interface creation, system configuration, validation, customer specific configurations, final download to runtime and functionality tests. The overall workflow is illustrated in Figure 17.

The process begins with the creation and configuration of a virtual environment. This virtual environment is a simulation environment typically hosted on a local server. After initializing the virtual environment, the next step is importing. In this phase the project engineer imports product libraries and generates product applications by importing specification files into the environment. Libraries are imported first because product generation utilizes library contents. Each project typically includes multiple products with distinct specifications, which must be individually imported to the system. The system provides product templates with a basic product structure, which can be configured with system specific details to instantiate simpler products. Complex products are generated by importing their specification files. This step is repeated for all products.

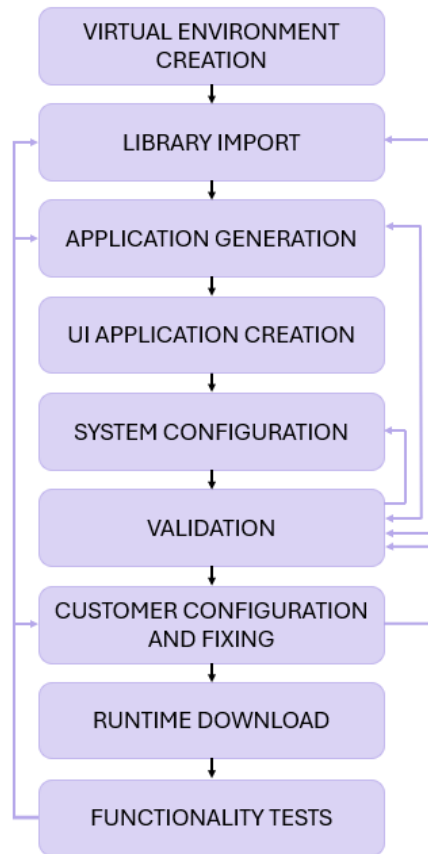


Figure 17. Productized application generation workflow.

Once the importing phase is complete, the following stage is the creation of the HMI of the system, which is the application User Interface (UI). Each product consists of multiple UI displays, which must be hierarchically ordered into several product UI applications to generate the required operator displays. Before this, all products must be ready in the system. The UI applications are created by configuring a UI application template with project specific parameters. Once the UI applications are prepared, the system configuration begins. This includes defining controllers and setting system parameters based on product specification values.

With the system configuration established, the first validation run of the system is conducted. Validation is an internal check that validates consistency of controller definitions in the applications and reveals any errors that require fixing. This might require returning back to the importing, application generation or system configuration phases. After validation, the project engineer adds the required connections between products and incorporates customer specific configurations to the applications. This iterative process requires the project engineer to make modifications and perform validation runs repeatedly until the system passes or the remaining errors are confirmed as resulting from missing customer hardware or other components that will be added later.

At this point, the system is ready for runtime download. Runtime download connects the generated applications with the simulation environment controllers. This reveals any configuration errors or other issues that validation missed. After this, the product engineer performs functionality tests that ensure all system connections are configured correctly. Finally, after fixing download errors and confirming that the application operates correctly in the runtime environment, the productized application generation phase is complete.

3.2 Limitations, challenges and possibilities for automation

The current deployment process includes some factors that affect efficiency and reliability. It involves manual steps and repetitive tasks, which increases the risk of errors. Addressing these errors can be time-consuming, reducing the time available for other engineering work. Some of the simpler deployment steps could be automated, enabling project engineers to focus on complex tasks requiring creativity and expertise.

One challenge relates to the product generation specification tool, which is not yet full alignment with the newly adopted web-based DCS architecture. As a result, some products must be imported without specification files by manually inserting specification parameters to templates. Manual template filling may introduce additional error potential, particularly in larger configurations. Correcting misplaced or missing parameters can be difficult, sometimes requiring re-execution of earlier steps. These challenges are largely due to the introduction of the newly adopted DCS and have been analyzed in earlier research, corrective actions are already underway to enhance the tools reliability and usability.

Product importing can be time-consuming, depending on project scale and configuration complexity. Each library and product import must be individually initiated by the project engineer. During import operations, the systems ability to perform parallel tasks is limited, leading to passive waiting periods. Additionally, during validation and downloading, errors may occur when parameters are incomplete or incorrectly inserted. Resolving issues at this stage can require considerable time depending on the scale of the product. In complex cases, regeneration of the product may be the most practical recovery method.

After the operator displays are generated, additional configuration work is often required. Such adjustments might involve additional signals and data elements or connections. Executing these adjustments requires thorough knowledge of the system level configuration.

Considering these limitations and challenges, automation offers considerable potential to improve efficiency and reduce errors. Automating specification file generation could eliminate manual specification filling by directly specifying generation files from the information on the sales BOM. Specification parameters and required libraries could be automatically fetched from the project database to the specification generation tool. Sim-

ilarly, automating the productized application generation process would reduce waiting and minimize human error. The overall workflow contains many recurring steps, several of which present clear opportunities for automation. For example large scale imports could be scheduled overnight, saving project engineers working hours during the day from active waiting. Furthermore, automation could extend to transferring the virtual environment to the customer system and transferring customer system to service. Introducing automation in these areas could support the transition toward a more modern and efficient deployment workflow.

4. PROPOSED AUTOMATED WORKFLOW

This chapter presents the proposed automated deployment workflow based on the challenges and opportunities identified in the previous chapter. The transition to a fully web-based DCS and the limitations of the current manual deployment process create an opportunity to redesign the workflow with improved efficiency, reduced manual effort, and increased reliability. The proposed workflow integrates automation into stages where tasks are repetitive or error-prone, while also enabling human intervention when necessary.

The contents of this chapter are organized into three sections. Section 4.1 proposes an ideal automated workflow, including a fully automated application generation process. Section 4.2 presents the functional and non-functional requirements for the PoC implementation. Finally, Section 4.3 evaluates deployment tools and identifies those best suited to the proposed automated workflow.

4.1 The ideal automated process proposal

To address the limitations discussed in the previous section, a proposition of an ideal automated workflow for application deployment was designed. Figure 18 illustrates the proposed process, highlighting the division of manual and automated tasks. Automated actions are shown in blue, manual tasks in green, and the automated application generation process, which will be discussed later in detail, is highlighted in violet. The figure is divided into columns for automation on the right, initialization on the left, and other deployment operations in the middle. This provides an explicit outlook on the integration of automation in to the deployment process.

The deployment process begins when an order is handed over to the project and the sales BOM is acquired. The system automatically reads the sales BOM information and initiates the product specification process. Product specifications and configuration parameters are retrieved from a project database, and presented to the project engineer for review and modification before approval. Once approved, the automated application generation process is initiated, which will be described in detail in the following section.

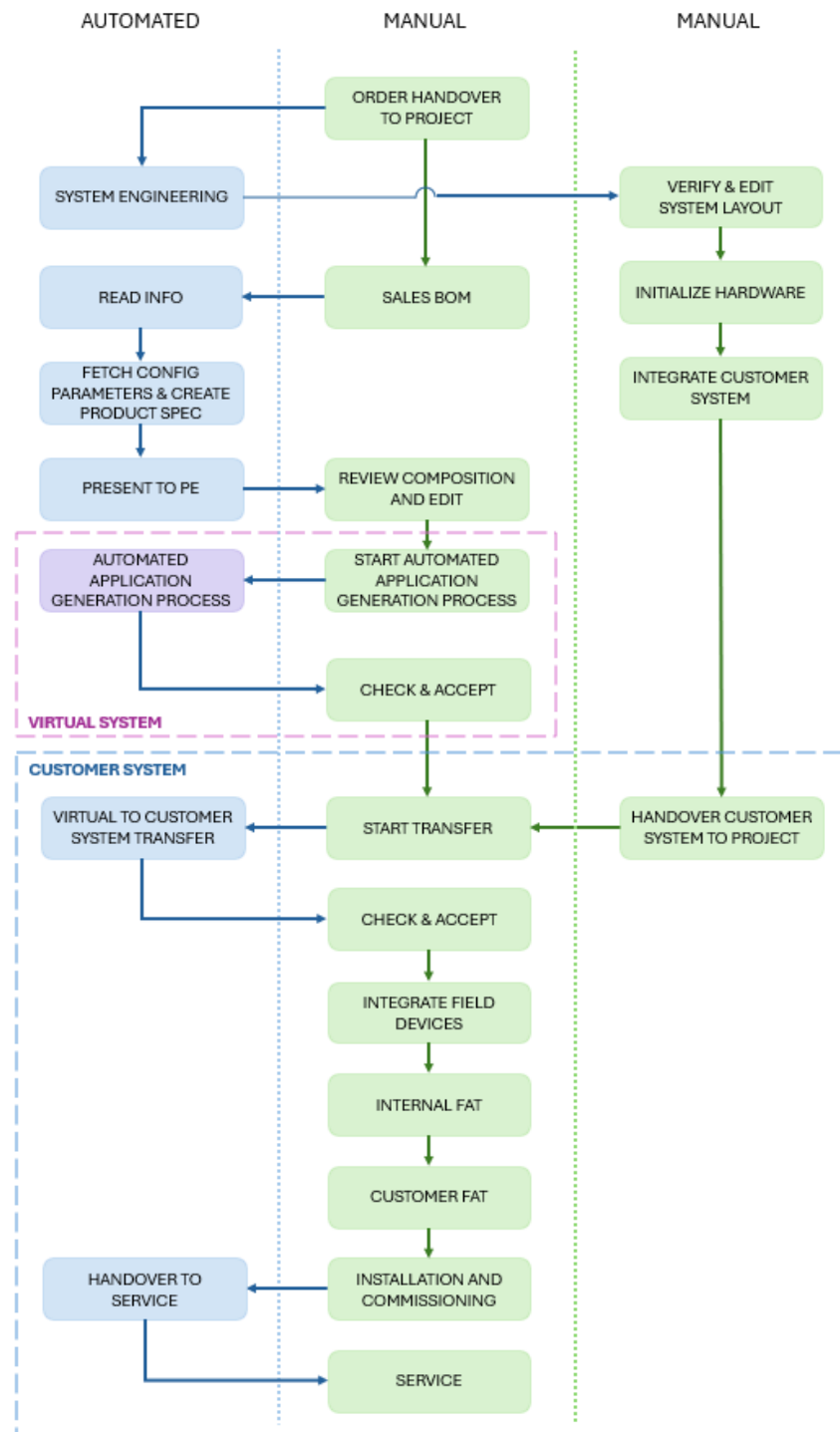


Figure 18. Ideal full deployment process utilizing automation.

Customer system initialization occurs simultaneously, beginning with automatic system engineering. Using the information provided by the sales order, the system layout is automatically generated. This layout is then reviewed and verified by system engineers, after which they proceed with the previously described initialization activities. After customer system and applications are ready, the project engineer starts the automatic virtual to customer system transfer. Once the transfer is verified, the remaining steps, including field

device integration, internal FAT, customer FAT, and plant installation, proceed as previously described. After successful plant installation the project engineer initiates handover to service, where the customer system, applications and parameters are transferred to a product environment ready for service operations.

Ideal automated application generation process

The ideal automated application generation process is illustrated in Figure 19. The workflow is divided into two columns to distinguish automated operations, shown in violet, from the manual interventions, shown in green.

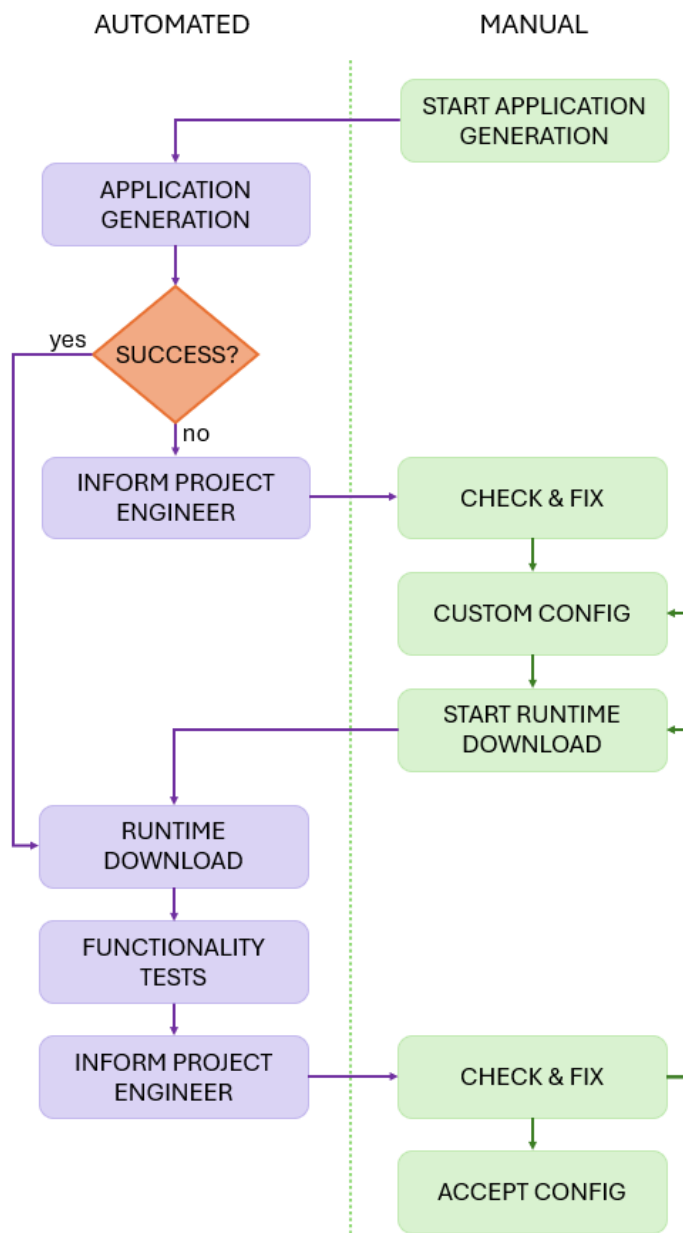


Figure 19. Ideal application generation process utilizing automation.

The ideal process begins when the project engineer initiates application generation. If application generation was successful, the system proceeds with runtime download and provides the results to the project engineer. The project engineer verifies the results and corrects possible errors and restarts the runtime download process. After the iterative process of fixing and downloading to runtime is finished, the project engineer performs custom configurations, which also need to be downloaded to runtime. If the application generation fails initially, the systems outputs an error report for the project engineer. The project engineer reviews the report and fixes errors before moving on to custom configurations. Lastly, the project engineer accepts the application configuration, concluding the ideal process of application generation.

Due to the complexity of this process, two additional flowcharts were created to illustrate the sub-processes of application generation and runtime download. Figure 20 presents the application generation workflow. The system begins by creating and configuring the virtual environment. After the virtual environment has been established, it fetches and imports the required libraries and specification files. The import has three possible results, success, recognized errors, and unrecognized errors. Recognized errors trigger a process of fixing and re-importing, while unrecognized errors terminate the process and generate an error report for the project engineer. A successful import moves the process to UI application generation. The system creates the main UI application and starts configuring the environment. Once configurations are done, the system starts validation, which follows the same logic as importing. Successful validation allows the system to progress to the next step, recognized errors trigger automated fixing and unrecognized errors terminate the process for manual intervention.

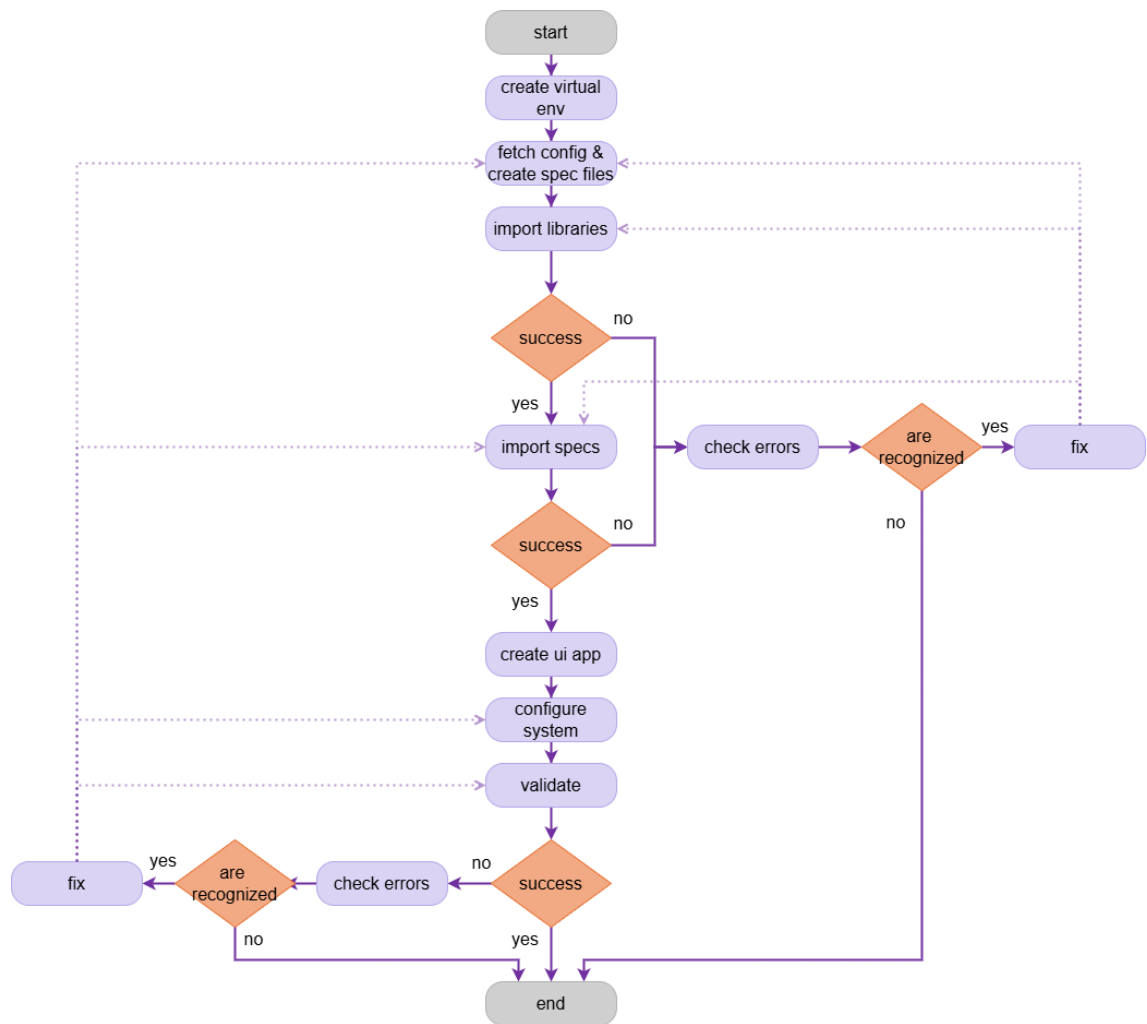


Figure 20. Automated application generation process.

The runtime download process begins immediately after successful validation of the application generation process. The flowchart of the runtime download is illustrated in Figure 21. The application is downloaded to the runtime environment and similarly to the results of validation and import processes, downloading also leads to three possible outcomes. A successful download or unrecognized errors terminate the process, while recognized errors trigger automatic correction and re-download. The corrections might require reverting back to phases in the application generation process.

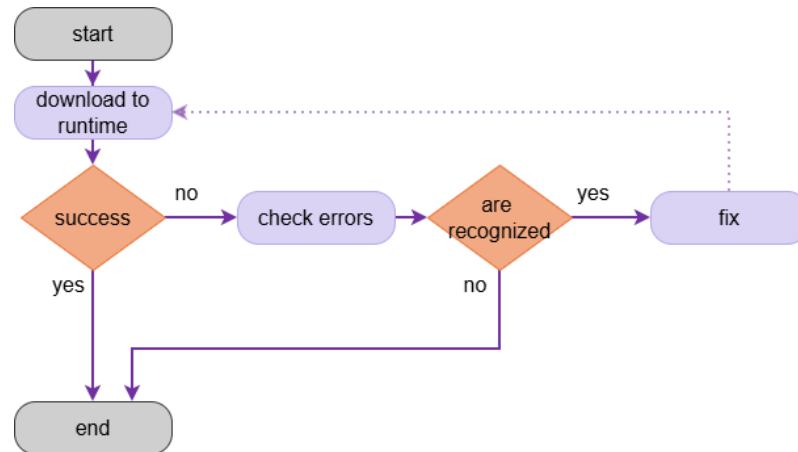


Figure 21. Automated runtime download process.

The aforementioned figures collectively illustrate the proposed ideal process for application deployment. The analysis and research conducted in Section 3 revealed notable opportunities to improve of the deployment process with automation. However, due to the extensive coverage of the proposed workflow, full implementation is beyond the scope of this thesis. Thus, the implementation described in Chapter 4 focuses on implementing the automated application generation process as a PoC, excluding the virtual environment creation and automated fixing.

4.2 Proof of Concept requirements

The functional and non-functional requirements listed in this section define the expected behavior and quality attributes of the PoC. These requirements serve as the basis for the design and implementation decisions presented in the following chapter. As this work focuses on a PoC, the requirements describe only the essential capabilities needed to demonstrate the feasibility of the automated application generation process. The requirements are based on document analysis of company deployment procedures, discussions with production and development professionals, field observation of an ongoing deployment project, and the ideal workflow described in Section 4.1.

Functional requirements:

- **FR1:** The system shall allow user to initiate a deployment run through a UI.
- **FR2:** The system shall fetch and import required product libraries.
- **FR3:** The system shall fetch specification files from a pre-configured source and generate product applications from them.
- **FR4:** The system shall fetch product information from the provided product specification files.
- **FR5:** The system shall retry import three times upon import error.

- **FR6:** The system shall create product UI applications and display hierarchies from generated products in the target environment.
- **FR7:** The system shall apply system configurations to the target environment based on product specification files.
- **FR8:** The system shall perform internal validation in the target environment to generated applications.
- **FR9:** The system shall perform runtime download after successful validation.
- **FR10:** The system shall produce a structured error report of all import, validation and download warnings and errors.
- **FR11:** The system shall filter error report to include errors first, important warnings second and expected warnings last.
- **FR12:** The system shall collect run artifacts, such as log information and error report for each product deployment run.
- **FR13:** The system shall support deployment parameter configuration through UI.
- **FR14:** The system shall pause deployment upon deployment step error and wait for user input.

Non-functional requirements:

- **NFR1:** Re-running with identical inputs and target environment version shall yield functionally identical output.
- **NFR2:** Solution shall operate on Windows.
- **NFR3:** The system's UI shall provide clear and intuitive user interaction for initiating deployment and handling user interference.
- **NFR4:** The system shall present error messages in a structured and human-readable format.
- **NFR5:** The system's deployment logic shall be modular to support future modifications.
- **NFR6:** The system shall include clear logging that supports debugging, traceability, and future maintainability.
- **NFR7:** The solution shall support addition of new product libraries without requiring redesign of the core pipeline.
- **NFR8:** Deployment execution reports and logs shall be uniquely identifiable per run to support distinct execution analysis.

4.3 Automation tool assessment

This section evaluates the general deployment automation tools introduced in Chapter 2.4 to determine their suitability for implementing the proposed automated deployment workflow. The assessment utilizes a point scoring approach to ensure systematic and transparent decision-making. Four criteria were selected based on organizational priorities and the constraints of PoC: Industrial Automation compatibility (IA Fit), Automation Capability, Implementation Workload, and Organizational Compatibility. The tools are evaluated with a weighted point system for each criteria, with emphasis on leveraging existing infrastructure and minimizing development effort. The weights of each criteria are presented in Table 1.

Table 1. Evaluation criteria weights

| Criteria | Weight (%) |
|------------------------------|------------|
| IA Fit | 15 |
| Automation Capability | 20 |
| Implementation Workload | 30 |
| Organizational Compatibility | 35 |

As illustrated in Table 1, the highest weights are assigned to organizational compatibility and implementation workload. The DCS system has accumulated multiple tools suited to the web UI, and leveraging these pre-existing implementations reduces cost while aligning the PoC with current organizational processes. Prioritizing tools that require minimal customization directly reduces implementation workload, which is essential given the PoCs limited resources and time constraints. Automation capability carries a lower weight because the workflow can be implemented with multiple tools rather than relying on a single solution. IA Fit has the lowest weight because the PoC scope focuses on virtual environment creation and operations performed through the DCS web UI. This favours general IT tools rather than specialized industrial solutions.

To ensure transparency and reproducibility, each tool is scored on a scale of 1-3 using predefined grading rules explained below. A score of 3 indicates strong alignment with the criteria, whereas a score of 1 indicates minimal alignment. The weighted total for each tool is calculated using the given alignment points and the criteria weights presented in Table 1.

Evaluation criteria grading:

- **IA Fit:** Compatibility with industrial automation systems.
 - **3 (High):** Fully compatible with Industrial Automation systems.
 - **2 (Medium):** Partial compatibility, needs some customization.
 - **1 (Low):** Minimal IA relevance, primarily IT-focused.
- **Automation Capability:** Ability to automate proposed application deployment tasks.
 - **3 (High):** Automates most deployment tasks independently.
 - **2 (Medium):** Automates some tasks, needs complementary tools.
 - **1 (Low):** Limited automation scope, only supports the automation of one task.
- **Implementation Workload:** Development effort and customization required for implementation.
 - **3 (Low effort):** Minimal customization required.
 - **2 (Medium effort):** Some configuration needed.
 - **1 (High effort):** Extensive development required.
- **Organizational Compatibility:** Available organization infrastructure and tool cost.
 - **3 (High):** Fully integrated or widely adopted internally.
 - **2 (Medium):** Partial fit, moderate integration effort.
 - **1 (Low):** No integration, significant effort required for integration.

The weighted total score for each tool t is calculated as

$$S_t = 0.35 \cdot OC_t + 0.30 \cdot IW_t + 0.20 \cdot AC_t + 0.15 \cdot IA_t,$$

where OC_t is Organizational Compatibility score, IW_t is Implementation Workload score, AC_t is Automation Capability score and IA_t is IA Fit score $\in \{1, 2, 3\}$.

The tool grading illustrated in Table 2 presents the points of all criteria, with a weighted total score for each tool.

Table 2. Point scores for each tool

| Tool | IA | AC | IW | OC | Weighted total |
|-------------------|-----------|-----------|-----------|-----------|-----------------------|
| Azure DevOps | 2 | 3 | 2 | 1 | 1,85 |
| GitLab CI/CD | 2 | 3 | 2 | 1 | 1,85 |
| Ansible | 1 | 2 | 2 | 3 | 2,2 |
| JFrog Artifactory | 3 | 1 | 3 | 3 | 2,6 |
| Python scripting | 3 | 3 | 1 | 3 | 2,4 |
| PowerShell DSC | 2 | 2 | 1 | 2 | 1,7 |
| Robot Framework | 2 | 2 | 3 | 3 | 2,65 |
| Jenkins | 2 | 1 | 3 | 3 | 2,45 |

Scores were assigned based on the tool descriptions in Section 2.4, analysis of existing organizational infrastructure, and feasibility within the PoC scope. As illustrated in Figure 22, PowerShell DSC received lower weighted totals due to limited workflow orchestration capabilities and tighter connection with specific vendor ecosystem. GitLab CI/CD and Azure DevOps achieved moderate scores, because they offer strong automation features but lack pre-existing organizational infrastructure, which increases implementation effort.

The highest weighted totals were achieved by Robot Framework, JFrog Artifactory, Jenkins, Python scripting and Ansible. These tools demonstrate high feasibility for automating different parts of the PoC while aligning with existing organizational processes. Robot Framework and Jenkins provide complementary orchestration and task automation, JFrog Artifactory supports efficient artifact life cycle management, Python enables flexible logic for complex deployment steps and Ansible facilitates virtual environment configuration.

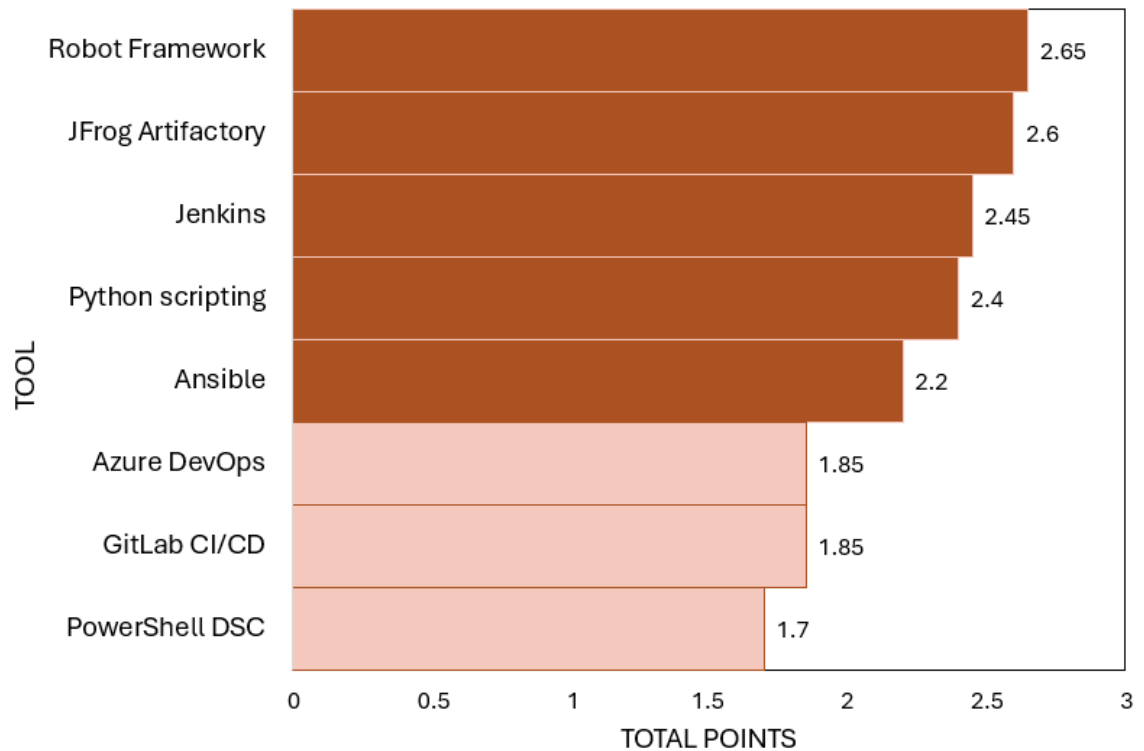


Figure 22. Weighted totals of each tool.

Final Selection:

- **Robot Framework** for automated task execution and deployment workflow.
- **Python scripts** for complex deployment logic.
- **JFrog Artifactory** for artifact storage and life cycle management.
- **Jenkins** for CI/CD workflow orchestration and test execution.

Based on this assessment, the proposed implementation will utilize the above selection of tools. These tools were chosen due to their high IA compatibility, strong automation capabilities, and alignment with existing infrastructure. These tools collectively enable the implementation of a PoC deployment workflow that fits the scope of the application generation process, leveraging existing organizational infrastructure.

5. IMPLEMENTATION

This chapter presents the implementation of the PoC system for automated deployment. Section 5.1 provides an overview of the PoC architecture and its main components. Section 5.2 describes how Jenkins orchestrates the deployment workflow, including environment provisioning and task execution. Section 5.3 explains how the project engineer interacts with the deployment system through Jenkins. Section 5.4 details the Robot Framework automation suite used to perform target-environment interactions. Finally, Section 5.5 presents how the system preserves and archives logs and reports for debugging, traceability, and future maintainability.

5.1 System overview

The implementation integrates existing company infrastructure to demonstrate automated deployment of automation applications. It combines Robot Framework automation, Git repositories, Jenkins orchestration, and an external artifact management system into a Pipeline capable of deploying products to a target DCS environment. Figure 23 presents the high-level architecture of this PoC. The system consists of five primary components: Jenkins controller, Jenkins Agent, Git repositories, external artifact manager, and target environment. Together, these components form the automation workflow used to execute and validate deployment.

The Jenkins controller manages the overall Pipeline by executing a declarative Jenkinsfile that defines the sequential automation steps. When a Pipeline run is triggered, the controller delegates the execution to a Jenkins Agent running on a Windows 11 workstation with the required access credentials. The Agent retrieves all necessary repositories over SSH, including a general Robot Framework automation repository, a robot test repository, and a PoC specific deployment task repository. This separation preserves modularity and isolates PoC specific logic from pre-existing test automation implementations.

Before the deployment tasks, the Agent provisions a clean Python 3.11 virtual environment and installs required dependencies. These dependencies include libraries for UI automation and data processing. It also ensures compatibility between the Chrome browser and ChromeDriver versions used by Selenium. This guarantees consistency across executions and prevents environment specific configuration issues.

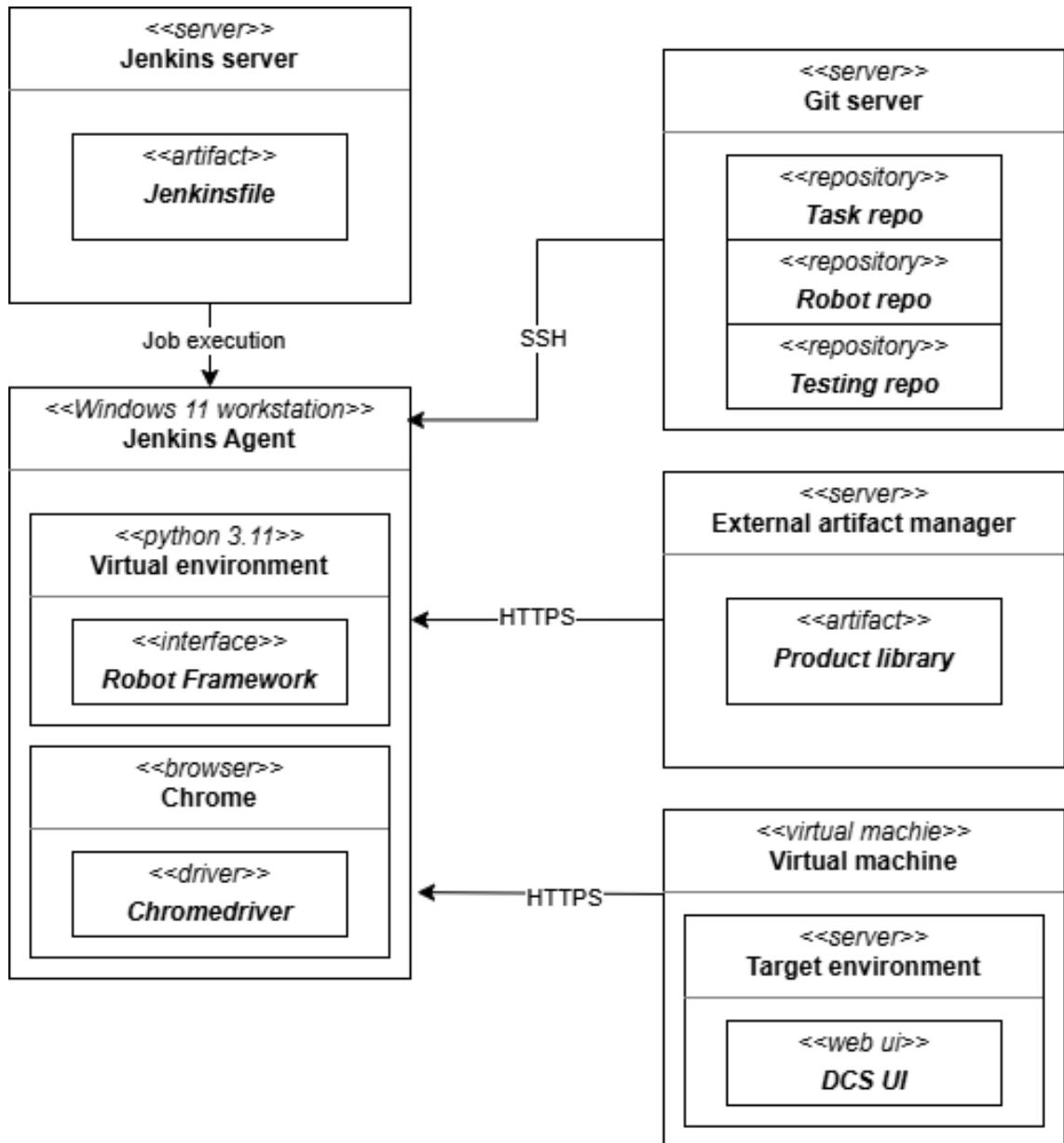


Figure 23. Overview of Proof of Concept system.

Robot Framework acts as the main automation interface. Using the Selenium Library, it interacts with the target DCS environment over HTTPS, treating the systems web-based UI as a standard browser application. The pre-built product libraries utilized to instantiate products in the target environment are retrieved from an external artifact management system via HTTPS. This approach ensures that deployment relies on up to date and validated components consistent with production setups.

The entire PoC operates in a controlled test environment that includes a local Jenkins installation and an isolated Windows workstation dedicated to Agent execution. This setup mirrors real deployment conditions while maintaining separation from production systems as well as providing a safe and realistic platform for evaluating the automated deployment approach.

5.2 Workflow orchestration with Jenkins Pipeline

The Jenkins Pipeline orchestrates the complete deployment workflow, from repository checkout to reporting. The Pipeline is implemented as a Jenkins Pipeline project configured to use a declarative Jenkinsfile stored in version control. SSH credentials are configured to grant the Agent access to the required Git repositories. Build history is maintained with automatic cleanup, retaining only the last 10 builds to manage storage.

The Pipeline is composed of three main phases:

- **Setup Phase** (Stages 1-7): Repository checkout, virtual environment creation, dependency installation, product specification processing and browser compatibility validation.
- **Execution Phase** (Stages 8-12): Sequential execution of Robot Framework tasks using predefined tags.
- **Reporting Phase** (Post actions): Collection and archiving of execution artifacts, summary reports, and Robot Framework outputs.

Stages 1, 2 and 4 handle the checkout of three Git repositories, task, robot and test. Stage 3 extracts the product specification files provided by the user and transfers them to the correct file location inside the task repository. Stage 5 establishes an isolated virtual environment within the robot repository directory if it is not already configured. Dependencies from both the robot automation repository and data processing libraries for summary report generation are installed to this environment. The product information required by the Robot Framework task code is fetched in stage 6. In stage 7, browser compatibility is validated by checking installed Chrome and ChromeDriver versions and downloading new versions when necessary.

In the execution phase, stages 8-12 execute the core deployment workflow through Robot Framework tasks identified by specific tags. These tags form the logical structure of the deployment workflow. Each tag represents an isolated functional operation in the target environment. This enables modular execution of tags and simplifies debugging. Tag-based execution also ensures that only the relevant Robot Framework tasks are triggered for each stage of the deployment process.

- **import**: Load product libraries and specifications into the target environment.
- **create-ui**: Create and configure UI applications.
- **configure-system**: Apply system-level configurations and parameters.
- **validate**: Execute validation tests.
- **download**: Deploy validated configurations to runtime environment.

Each tag is triggered through helper functions, illustrated in Figure 24. These functions are defined directly in the Jenkinsfile and determine the logic for executing tasks.

```

1  def runRobotTask(tag, product) {
2      dir(env.ASSET_DIR) {
3          timeout(activity: true, time: 20, unit: 'MINUTES') {
4              def result = bat(returnStatus: true, script: """
5                  call ".venv\\Scripts\\activate.bat" &&
6                  python -u run.py --headless --exitonfailure
7                  -v SERVER:%SERVER%
8                  -v OSTYPE:%OSTYPE%
9                  -v PRODUCT_IDENTIFIER:${product}
10                 -v DEPLOYMENT_ID:%DEPLOYMENT_ID%
11                 -i ${tag}
12                 set RF_EXIT_CODE=%ERRORLEVEL%
13                 copy output.xml output_${product}_${tag}.xml
14                 exit /b %RF_EXIT_CODE% """
15                 echo "Robot task ${tag} for product ${product} returned exit code: ${result}"
16                 if (result != 0) {
17                     echo "Task ${tag} for product ${product} failed with exit code ${result}"
18                     handleTaskFailure(tag, product) }
19             }
20         }
21     }
22
23     def handleTaskFailure(taskName, productId) {
24         def userChoice
25         timeout(activity: true, time: 20, unit: 'MINUTES') {
26             userChoice = input message: "${taskName} ${productId} failed. Choose action:",
27                             parameters: [choice(
28                                 choices: ['Retry', 'Skip', 'Abort'],
29                                 name: 'ACTION')] }
30         switch(userChoice) {
31             case 'Retry':
32                 dir(env.WORKSPACE) { runRobotTask(taskName, productId) }
33                 break
34             case 'Skip':
35                 echo "Task ${taskName} ${productId} skipped by user"
36                 currentBuild.result = 'UNSTABLE'
37                 break
38             case 'Abort':
39                 error "Build aborted by user after ${taskName} ${productId} failure"
40                 break
41             default:
42                 error "Build failed after ${taskName} ${productId} no valid action selected"
43         }
44     }

```

Figure 24. Robot task execution functions.

The script defines two helper functions, `runRobotTask` and `handleTaskFailure`, which together control the execution flow of each Robot Framework task. The `runRobotTask` function initiates the task run by activating the Python virtual environment, starting the `run.py` execution script and applying the required tag through the `-i` argument. Pipeline parameters such as `SERVER`, `OSTYPE`, `PRODUCT_IDENTIFIER` and `DEPLOYMENT_ID` are forwarded to Robot Framework using variable arguments. All tasks are executed in headless mode, which suppresses the Selenium browser UI.

The function is wrapped in a 20 minute activity timeout to prevent endless execution if execution freezes. After the task completes, the function captures the Robot Framework exit code and stores a copy of the `output.xml` file with a product and tag specific filename for later reporting. If the returned exit code indicates a task failure, the Pipeline delegates error handling to `handleTaskFailure`.

The `handleTaskFailure` function presents an interactive prompt in Jenkins, pausing Pipeline execution waiting for input from the project engineer. The prompt offers three options: retry failed task, skip task and continue the Pipeline, or abort the entire build. Selecting retry recursively triggers `runRobotTask` with the same parameters, while skipping the task marks the build as UNSTABLE. Abort results in an immediate Pipeline failure. This mechanism provides controlled intervention during deployment and supports iterative troubleshooting without restarting the full Pipeline run.

Additionally, the Pipeline has a defined post actions phase, which is executed after each deployment run. As illustrated in Figure 25, the Pipeline post phase collects Robot Framework artifacts, including logs, reports, and the generated Excel summary file and archives them.

```

255 post {
256     always {
257         script {
258             // Try run report task
259             try {
260                 env.EXTRACTED_PRODUCTS.split(',').each
261                 { product -> runRobotTask('report', product.trim()) }
262             } catch (Exception e) {
263                 echo "Report task failed: ${e.getMessage()}"
264             }
265
266             // Merge Robot Framework output files
267             dir(env.ASSET_DIR) {
268                 bat """
269                 call .venv\\Scripts\\activate.bat
270                 powershell -Command "$files = Get-ChildItem 'output_*.xml' |
271                 Sort-Object CreationTime | Select-Object -ExpandProperty Name;
272                 if (\$files) {
273                     rebot --outputdir . --output output.xml --log log.html
274                     --report report.html \$files; Remove-Item 'output_*.xml' -Force
275                 } else {
276                     Write-Host 'No output files found to merge' }"
277                 """
278             }
279
280             // Try copy Excel report
281             dir(env.ASSET_DIR) {
282                 bat """
283                 powershell -Command "try {
284                 Get-ChildItem -Path
285                 '${REPORT_PATH}\\${BUILD_NUMBER}_*\\*.xlsx' |
286                 ForEach-Object {
287                 Copy-Item \$_ .FullName .; Write-Host \"Copied report: \$_ .Name\" }
288                 } catch {
289                 Write-Warning \"Failed to copy Excel report: \$_ .Exception.Message\" }"
290                 """
291             }
292         }
293
294         robot outputPath: "${ASSET_DIR}",
295             logFileName: 'log.html',
296             reportFileName: 'report.html',
297             outputFileName: 'output.xml'
298
299         archiveArtifacts artifacts: \
300             "${ASSET_DIR}/*_error_warning_summary.xlsx, ${ASSET_DIR}/log.html,
301             ${ASSET_DIR}/report.html, ${ASSET_DIR}/output.xml",
302             fingerprint: true,
303             allowEmptyArchive: true
304     }
305 }
306 }

```

Figure 25. Pipeline post actions section.

In the post section the Jenkins Robot Framework plugin parses these outputs and presents detailed execution results within the Jenkins interface. The post section calls the report task, which generates an `error_warning_summary.xlsx` file. All artifacts are marked with `fingerprint` for comparison between builds and with `allowEmptyArchive` to handle scenarios where certain reports may not be generated due to task failures or Pipeline

execution aborts. As a result, the post-execution workflow offers a robust method for capturing, structuring, and retaining all relevant deployment artifacts.

5.3 User interaction

The execution of the deployment system is initiated through the Jenkins UI, from the Jenkins Build with Parameters view, shown in Figure 26. Prior to execution, the project engineer configures the runtime parameters defined in the Jenkinsfile parameters section. These include the target environment server address and the operating system type required by the Robot Framework executor. Parameterization ensures that the same Pipeline definition can be applied across different environments and products while maintaining a consistent operational interface. Additionally, to convey product information to the Pipeline, the project engineer inserts the product specification files to the Jenkins workstation directory for later retrieval. Once parameters are set and the specification files are in the correct location, the Pipeline execution can be initiated.

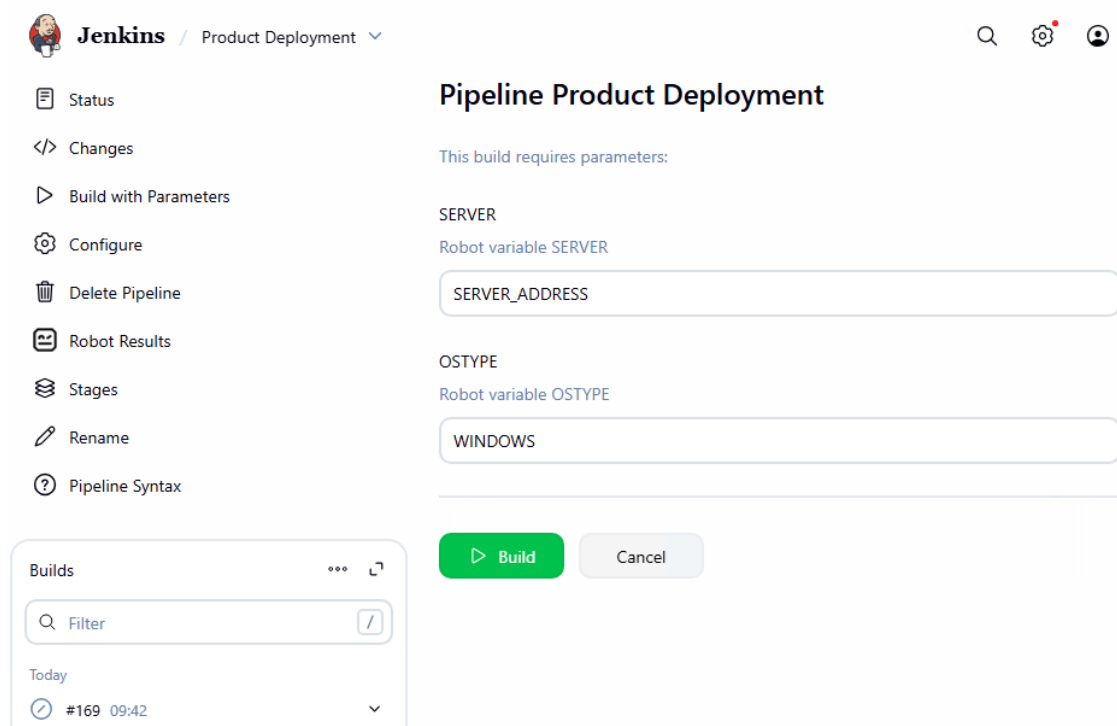


Figure 26. Pipeline Build with Parameters page.

Execution progress is observed through the Console Output view, which provides a sequential, time-stamped log of Pipeline actions and messages emitted by each stage execution. This allows the project engineer to follow the progression of tasks, identify the stage at which a failure occurs, and correlate log messages with the deployment operations.

With customized error handling in the Pipeline robot task execution, the project engineer may interfere upon task failure. When a failure is detected, the Pipeline halts execution and waits for user input. While the Pipeline execution waits, the project engineer may perform corrective actions in the target environment, such as applying configuration changes, resolving missing asset or adding custom configurations. After these corrective actions, the project engineer returns to the Pipeline and responds to the input prompt shown in Figure 27. The prompt presents a drop-down selection of possible actions, such as retrying the failed task, skipping the failed task or terminating the execution. This enables the partial automation of the Pipeline that allows the project engineer to intervene and continue execution with a clear record of chosen action in the console output.

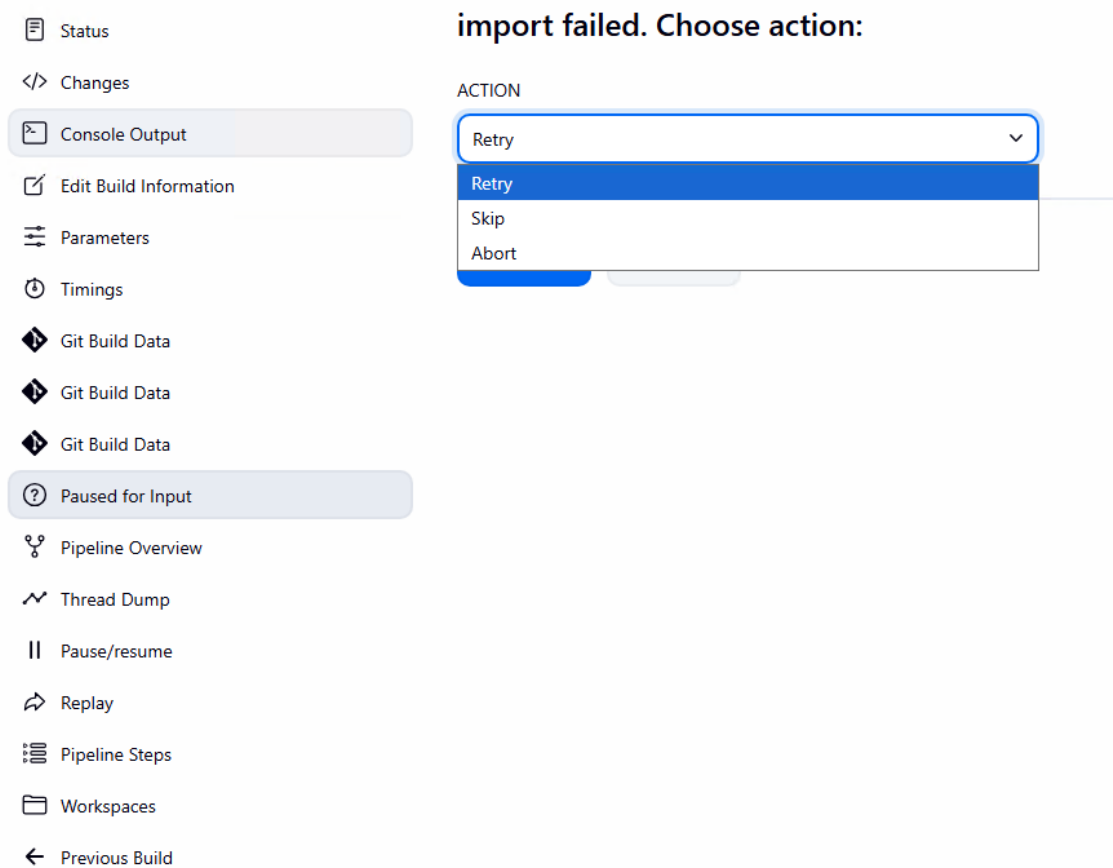


Figure 27. User input prompt presented upon task failure.

This user interaction implementation supports both automatic execution and manual intervention when needed. If validation completes successfully, the Pipeline proceeds automatically to the runtime download stage without requiring further input, as outlined in Section 4.1. Otherwise, if validation or any other task fails, the Pipeline waits for user input according to the defined Pipeline logic. This design allows the project engineer to easily monitor, control and interfere with the deployment Pipeline execution to address possible issues and implement custom configurations.

5.4 Deployment task execution

Interactions with the target DCS environment are implemented using a Robot Framework suite that automates browser-based operations via Selenium WebDriver. The suite performs UI-driven configuration tasks and exports diagnostic data from the system to support debugging and traceability.

The suite defines custom setup and teardown keywords, illustrated in Figure 28. Upon a task execution call, the suite setup acquires suite-level parameters for the selected product, creates a dedicated export directory for runtime artifacts, and opens the browser to the DCS web interface. Suite parameters are defined in an external file that defines parameters, such as `BASE`, `PRODUCT_NAME`, `PRODUCT_FOLDER` and `PRODUCT_IDENTIFIER`. These parameters are fetched and assigned by a custom keyword that sets their scope to suite-wide, ensuring consistent availability throughout execution. The suite teardown logs out the user and closes the browser, ensuring proper termination of the browser session.

```

57  *** Keywords ***
    Load in Interactive Console
58  Suite Setup
59      Load Product Configuration    ${PRODUCT_IDENTIFIER}
60      Create Export folder
61      Open Browser To UI Browser
62
    Load in Interactive Console
63  Task Setup
64      Check Browser Is Open
65
    Load in Interactive Console
66  Task Teardown
67      Run Keyword If Test Failed    Capture Page Screenshot
68
    Load in Interactive Console
69  Suite Teardown
70      Logout and Close Browsers    ${CONF_LOGIN_USER}    ${UI_LOGOUT_BUTTON}

```

Figure 28. Setup and teardown keyword definitions.

To support task-level diagnostics, the suite defines a task setup and teardown. Task setup validates that the browser is open, enabling a fail-fast behavior that terminates execution immediately if the prerequisite is not met. If the keyword fails, the Robot Framework report clearly indicates the cause, enabling fast identification of the cause of failure. Task teardown automatically captures a screenshot of the user interface upon task failure to preserve visual evidence of the page state, including potential error messages and unexpected navigation outcomes. The keyword `Run Keyword If Test Failed`, provided by the Robot Framework BuiltIn library, is utilized regardless of the naming `test`, because the keyword functionality is not affected by semantics.

Figure 29 outlines the suites tasks that are invoked from the deployment Pipeline. Each task is assigned a unique tag to enable selective execution from Jenkins, which supports

modular orchestration and clear separation between Pipeline stages. First is the product library and specification importing. The product libraries are fetched from an external artifact manager server and imported to the target environment.

```

Run Suite | Debug Suite | Load in Interactive Console
1  *** Settings ***
2  Documentation      A simplified/redacted version of the task suite.
3  Resource           $(PATH)/resources/resource_file.resource
4
5  *** Tasks ***
Run | Debug | Run in Interactive Console
6  Import Libraries and Specs
7  [Documentation]    Imports product libraries and specification files.
8  [Tags]            import
9  Wait Until Keyword Succeeds    3x    300s    Download and Import Library
10 ... $(PRODUCT_IDENTIFIER)    $(PRODUCT-LIBRARIES_PATH)/job/$(PRODUCT_NAME)    $(FILE_NAME)
11 Export Log Result Details
12 Wait Until Keyword Succeeds    3x    300s    Import Product Specs    $(PRODUCT_NAME)
13
Run | Debug | Run in Interactive Console
14 Create Ui App
15 [Documentation]    Creates Ui Application and Hierarchy.
16 [Tags]            create-ui
17 Create App And Hierarchy    $(PRODUCT_IDENTIFIER)    $(PRODUCT_FOLDER)
18 Configure Hierarchy    $(PRODUCT_IDENTIFIER)    $(BASE)    $(PRODUCT_FOLDER)
19
Run | Debug | Run in Interactive Console
20 Configure System
21 [Documentation]    Configures system.
22 [Tags]            configure-system
23 @{controllers}= Get Product Controllers
24 FOR $(controller) IN @{controllers}
25 | Create Controller    $(controller)
26 END
27
Run | Debug | Run in Interactive Console
28 Validate
29 [Documentation]    Validates product elements and exports results.
30 [Tags]            validate
31 IF '$(PRODUCT_NAME)' == 'product1'
32 | Validate Packages    @{product1folder}
33 ELSE IF '$(PRODUCT_NAME)' == 'product2'
34 | @{base_children}= Get Package Children    $(PRODUCT_FOLDER_PATH)    $(BASE)
35 | Validate Items In Path    $(PRODUCT_FOLDER_PATH)    @{base_children}
36 END
37
Run | Debug | Run in Interactive Console
38 Download To Runtime
39 [Documentation]    Downloads packages and system node to runtime.
40 [Tags]            download
41 Download Package and System Node    $(PRODUCT_FOLDER)
42 @{controllers}= Get Product Controllers
43 FOR $(controller) IN @{controllers}
44 | Download Control Loops    $(controller)
45 END
46 Download Control Loops    $(PRODUCT_FOLDER)
47
Run | Debug | Run in Interactive Console
48 Generate Final Error And Warning Report
49 [Documentation]    Summarizes all warnings and errors from exported Excel files into a final report
50 [Tags]            report
51 Summarize Result Log Warnings And Errors    $(CURRENT_TEST_RUN_EXPORT_FOLDER)
52 ... $(CURRENT_TEST_RUN_EXPORT_FOLDER)    $(PRODUCT_NAME)

```

Figure 29. Task suite tasks definition.

Next is creating the UI app, which creates the product UI application and establishes its display hierarchy. After UI application creation, the target system is configured. Con-

troller definitions, which vary by product and must match the specification precisely, are extracted from the imported product artifacts using the `Get Product Controllers` keyword. After the extraction, the controllers are instantiated and configured in the environment.

After imports, UI application creation, and system configuration, the target environment elements are validated using an internal check tool. In the example implementation shown in Figure 29, there are two example products defined in the implementation to illustrate different product compositions. Product A is simpler and can be validated more directly, whereas Product B is more complex and requires deeper traversal to locate the elements that need validation. After successful validation, a runtime download is executed for all required product elements to complete target environment actions.

The final keyword, which is called at the end of the deployment Pipeline, executes the generation of a final error report. It utilizes report logs that are exported after each import, validation and download action from the target environment. The task calls a custom python keyword `Summarize Result Log Warnings And Errors`, with parameters specifying the deployed product and the export directory paths. The details of the summary generation are presented later in Section 5.5.

To handle importing errors in the target environment, the import task implementation illustrated in Figure 29, utilizes a `Wait Until Keyword Succeeds` keyword from Robot Frameworks `BuiltIn` library. The keyword takes as parameters the number of retries, time-out of one retry and the keyword that will be run. As most importing failures in the DCS system are caused by transient internal issues, the chosen mitigation is to retry the import. The retry count is set to three to provide sufficient attempts without risking prolonged execution upon major system issues. The timeout per import attempt is set to 300 seconds to accommodate larger product libraries and specification files while ensuring that the overall execution terminates in a timely manner if importing does not succeed.

5.5 Execution reports

To evaluate the results of the deployment, the project engineer may acquire logs and reports of the deployment task execution from the Jenkins artifact storage. These reports verify the functionality of the deployment tool and are used to assess the success of the automated workflow. Robot Framework generates the standard log, output, and report files during task execution. These are automatically archived and made available through the Jenkins UI.

In addition to the built-in Robot Framework outputs, an export keyword is executed after each import, validation, and download action. This keyword captures structured result data and stores it in a predefined folder hierarchy, allowing the exported information to

be categorized per action and used later to generate a consolidated summary report. All archived artifacts collected by the Pipeline can be downloaded directly from the Jenkins interface for further inspection.

The final summary report of the deployment is generated by a custom keyword implemented in the `excel_report_processor.py` file. The Jenkins Pipeline triggers this processor during the post actions phase by running a dedicated tag. The processor fetches all of the exports that were captured during task execution, and summarizes their contents into one main report. The general structure of this summary file is illustrated in Figure 30. The report lists the key information extracted from each import, validation and download action. This information typically includes important error messages, locations and detailed information on the element that caused the error. The report reveals errors related to the automation tools execution and specification, to enable the project engineer to easily assess the limitations and make fixes to the system.

| | A | B | C | D | E | F | G |
|----|---------|-------------|-------------|-------------|--------------|------------------------|-------------|
| 1 | Status | Object name | Type | Location | Message type | Description | Source File |
| 2 | error | object_name | object_type | object_path | ERROR | system error message | source |
| 3 | error | object_name | object_type | object_path | ERROR | system error message | source |
| 4 | error | object_name | object_type | object_path | ERROR | system error message | source |
| 5 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 6 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 7 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 8 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 9 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 10 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 11 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 12 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 13 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 14 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 15 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 16 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 17 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 18 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 19 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 20 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 21 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 22 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 23 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 24 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 25 | warning | object_name | object_type | object_path | WARNING | system warning message | source |
| 26 | warning | object_name | object_type | object_path | WARNING | system warning message | source |

Figure 30. Layout of the report summary file.

To improve readability, the processor formats the report with consistent column widths, cell alignment, and color coding. The exports may contain hundreds of lines, many of which offer limited value for debugging. To address this, the processor filters and prioritizes the content based on severity. The most crucial information is all lines labeled as errors, which are inserted to the very top of the summary report and marked with a red background. Secondary, but still important, are lines labeled as warnings, these are inserted after the errors and marked with yellow background. This prioritized layout ensures that the project engineer can quickly identify significant issues and focus on the most relevant diagnostic information.

6. TESTS AND RESULTS

This chapter evaluates the Proof of Concept system developed for automated product deployment. Section 6.1 introduces the test environment and describes the scenarios used to assess the functional correctness, robustness, and scalability of the PoC. Section 6.2 presents and analyses the outcomes of the test executions. Section 6.3 summarizes feedback from engineers regarding the practical relevance and usability of the implementation.

6.1 Test scenarios and methodology

Three test scenarios were defined to evaluate the core functionality of the PoC. Each scenario represents typical operating conditions or failure situations encountered in application deployment projects. All tests were executed in the same environment described in Chapter 5. A target environment separate from production operations was used for testing purposes. This allowed test to be executed in a controlled environment, that simulates a real project scenario of deploying a product in to a new environment. To ensure consistency between different test cases, each test run introduced in this section is executed in the same environment. Additionally, the environment was cleaned after each run to ensure all tests started from an identical baseline.

Two test products with different characteristics were selected:

- **Product A:** A small product with a limited number of parameters and short deployment time.
- **Product B:** A larger and more complex product containing significantly more parameters and a longer deployment time.

These products were chosen to observe how product size and complexity affect Pipeline behavior, execution time, and output structure. The selected test scenarios focus on assessing the functional correctness, consistency across repeated runs, and the automation tools ability to handle failures. Broader performance, stress, and error tolerance testing were out of scope for this thesis.

Test cases

CASE 1: End-to-end deployment of a single product

- **Purpose:** Verify that the Pipeline executes the full deployment workflow for products with different complexity consistently.
- **Setup:** The Pipeline is executed three times for Product A and for Product B using identical parameters.
- **Expected outcome:** Successful completion of all Pipeline stages for both products. Deployment of product B is expected to take noticeably longer than product A due to the difference in product complexity. The execution time of each consecutive deployment is expected to stay the same within one product.

CASE 2: End-to-end deployment of multiple products.

- **Purpose:** Evaluate how the Pipeline handles multiple products in a single execution.
- **Setup:** The Pipeline is provided with a combined zip containing both product A and product B specifications.
- **Expected outcome:** The Pipeline executes all stages sequentially for both products without interference. Individual summary files are expected for each product, as well as a consolidated Robot Framework log and report covering the entire deployment process for both products.

CASE 3: Validation failure leading to user intervention.

- **Purpose:** Test whether the Pipeline correctly handles a failure in the validation process and demonstrates the user interaction mechanism.
- **Setup:** A scenario where validation fails is executed. Required corrections are applied manually in the DCS, after which the Pipeline is resumed using the Jenkins UI prompt option "Retry".
- **Expected outcome:** The Pipeline pauses at the validation stage, requests user input, and resumes correctly after the issue is fixed. Validation should succeed on retry, allowing the Pipeline to continue and complete successfully.

6.2 Test execution results

The execution of each test case produced consistent and technically successful results for the PoC. Overall, the tests indicate that the implementation meets its core functional objectives.

CASE 1: End-to-end deployment of a product

Single product deployment completed successfully, producing the expected report and log files as output. For each Pipeline execution, the implementation correctly recognized the product from the provided specification files. As illustrated in Table 3, execution times differed predictably between the two products. Product B consistently required more time than Product A. The times remained consistent across the three runs, which indicates stable performance.

Table 3. Pipeline total execution times for product A and product B [minutes]

| | Product A | Product B |
|---------|-----------|-----------|
| Build 1 | 14 | 29 |
| Build 2 | 14 | 29 |
| Build 3 | 14 | 29 |

Figure 31 illustrates the breakdown of Pipeline stage durations. The results show that Product B requires more time in stages related to importing product libraries and specification files, as well as during validation and runtime download. Other stages remain similar between the two products, which is expected since those parts of the process are not influenced by product complexity.

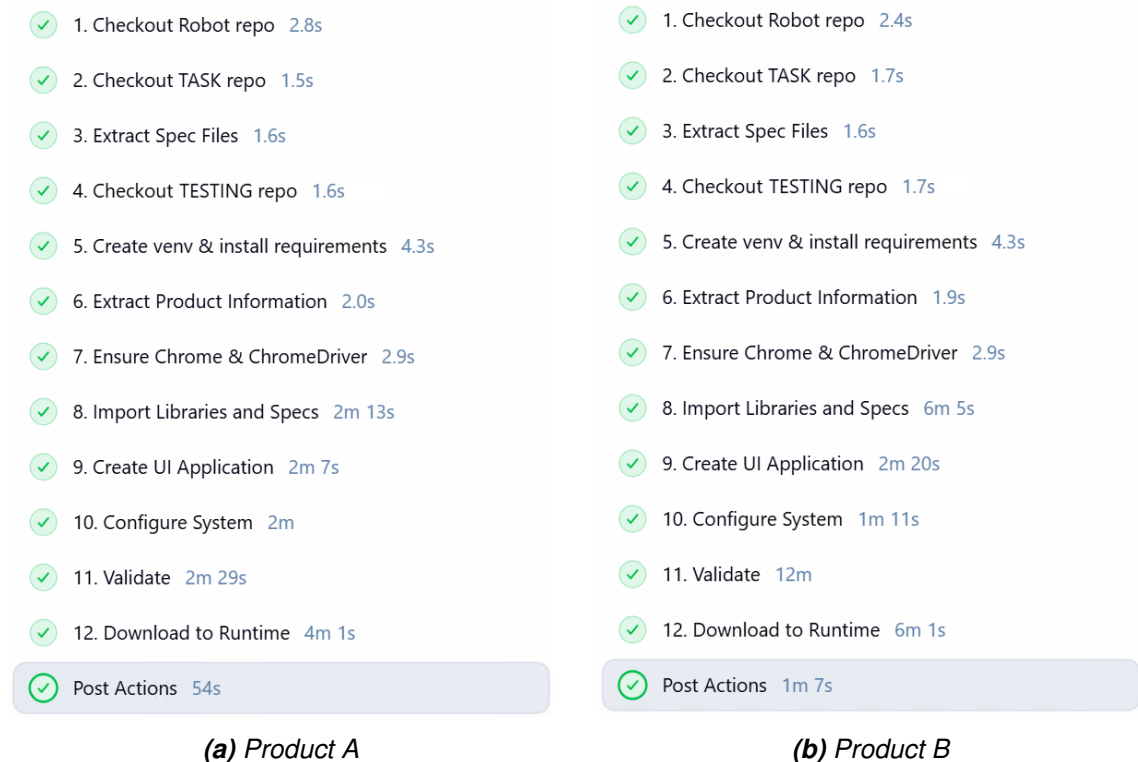


Figure 31. Comparison of Pipeline stages execution time for product A and product B.

Each test run produced the correlating Robot Framework log and report files, as well as the product deployment summary file. The Robot Framework log in Figure 32 for product B shows each task tag was executed successfully.

| Summary Information | | | | | | | |
|---------------------|------------------|------|------|------|----------|--------------------|--|
| Status: | All tasks passed | | | | | | |
| Elapsed Time: | 00:26:02.865 | | | | | | |
| Log File: | log.html | | | | | | |
| Task Statistics | | | | | | | |
| Total Statistics | Total | Pass | Fail | Skip | Elapsed | Pass / Fail / Skip | |
| All Tasks | 6 | 6 | 0 | 0 | 00:24:02 | ██████████ | |
| Statistics by Tag | Total | Pass | Fail | Skip | Elapsed | Pass / Fail / Skip | |
| configure-system | 1 | 1 | 0 | 0 | 00:00:25 | ██████████ | |
| create-ui | 1 | 1 | 0 | 0 | 00:01:24 | ██████████ | |
| download | 1 | 1 | 0 | 0 | 00:05:11 | ██████████ | |
| import | 1 | 1 | 0 | 0 | 00:05:20 | ██████████ | |
| report | 1 | 1 | 0 | 0 | 00:00:12 | ██████████ | |
| validate | 1 | 1 | 0 | 0 | 00:11:30 | ██████████ | |

Figure 32. Robot Framework report of product B deployment.

Both products follow an identical execution workflow at the tag level, and the variation between them is the tag execution times. Figure 33 presents the equivalent report for Product A, demonstrating the same operational sequence.

| Summary Information | | | | | | | |
|---------------------|------------------|------|------|------|----------|--------------------|--|
| Status: | All tasks passed | | | | | | |
| Elapsed Time: | 00:10:55.658 | | | | | | |
| Log File: | log.html | | | | | | |
| Task Statistics | | | | | | | |
| Total Statistics | Total | Pass | Fail | Skip | Elapsed | Pass / Fail / Skip | |
| All Tasks | 6 | 6 | 0 | 0 | 00:09:00 | ██████████ | |
| Statistics by Tag | Total | Pass | Fail | Skip | Elapsed | Pass / Fail / Skip | |
| configure-system | 1 | 1 | 0 | 0 | 00:01:14 | ██████████ | |
| create-ui | 1 | 1 | 0 | 0 | 00:01:21 | ██████████ | |
| download | 1 | 1 | 0 | 0 | 00:03:15 | ██████████ | |
| import | 1 | 1 | 0 | 0 | 00:01:27 | ██████████ | |
| report | 1 | 1 | 0 | 0 | 00:00:02 | ██████████ | |
| validate | 1 | 1 | 0 | 0 | 00:01:41 | ██████████ | |

Figure 33. Robot Framework report of product A deployment.

These results indicate that Pipeline execution time scales predictably with product size. The stability across repeated executions also suggests that the Pipeline is not influenced by external factors. This behavior is important for deployments in project environments where repeatability and traceability are required.

CASE 2: End-to-end deployment of multiple products

When deploying multiple products, the Pipeline successfully extracted each product from the given specification files and deployed them to the target environment. As illustrated in Figure 34, each Pipeline stage completed successfully with both products. The total execution time of the deployment was 42 minutes, which is as expected with product A and product B.

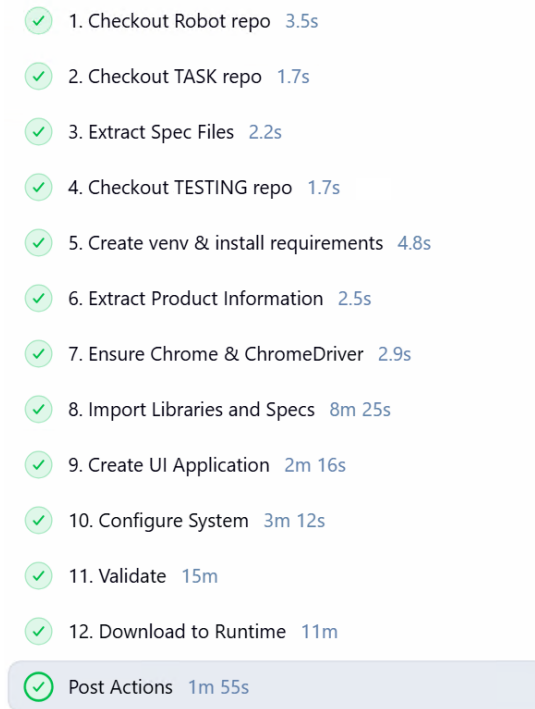


Figure 34. Pipeline stage execution times for multiple product deployment.

For further details about the deployment run, the execution produced Robot Framework report and log files that summarized the full deployment, as well as individual summary files for each product. The Robot Framework report illustrated in Figure 35 shows each task result and execution time for the full deployment process.

| Summary Information | | | | | | | |
|---------------------|--------------------------|------|------|------|----------|--------------------|--|
| Status: | All tasks passed | | | | | | |
| Elapsed Time: | 00:36:41.669 | | | | | | |
| Log File: | log.html | | | | | | |
| Task Statistics | | | | | | | |
| Total Statistics | Total | Pass | Fail | Skip | Elapsed | Pass / Fail / Skip | |
| All Tasks | 11 | 11 | 0 | 0 | 00:33:05 | ██████████ | |
| Statistics by Tag | Total | Pass | Fail | Skip | Elapsed | Pass / Fail / Skip | |
| configure-system | 2 | 2 | 0 | 0 | 00:01:40 | ██████████ | |
| create-ui | 1 | 1 | 0 | 0 | 00:01:27 | ██████████ | |
| download | 2 | 2 | 0 | 0 | 00:09:23 | ██████████ | |
| import | 2 | 2 | 0 | 0 | 00:06:55 | ██████████ | |
| report | 2 | 2 | 0 | 0 | 00:00:15 | ██████████ | |
| validate | 2 | 2 | 0 | 0 | 00:13:25 | ██████████ | |

Figure 35. Robot Framework report of multiple product deployment.

The results demonstrate that the PoC can reliably handle multiple products in a single execution, that produces clear reports for inspection. This is a key requirement for real projects, where deployments often involve multiple products of different complexities.

CASE 3: Validation failure triggering user intervention

Figure 37 shows the Pipeline stages and their durations for the simulated validation failure scenario. The validation stage for Product B took considerably longer because it included the failure event and the manual fix process.

- ✓ 1. Checkout Robot repo 3.5s
- ✓ 2. Checkout TASK repo 2.0s
- ✓ 3. Extract Spec Files 2.8s
- ✓ 4. Checkout TESTING repo 1.6s
- ✓ 5. Create venv & install requirements 8.4s
- ✓ 6. Extract Product Information 4.8s
- ✓ 7. Ensure Chrome & ChromeDriver 4.6s
- ✓ 8. Import Libraries and Specs 8m 50s
- ✓ 9. Create UI Application 2m 4s
- ✓ 10. Configure System 1m 52s
- ✓ 11. Validate 32m
- ✓ 12. Download to Runtime 11m
- ✓ Post Actions 1m 43s

Figure 36. Product B Pipeline stages execution times.

To better illustrate the behavior inside the validation stage, Figure 37 shows the execution of the validation stage in detail. The validation failure triggered the Pipeline to request user input as intended. The user was prompted twice, as the first manual fix did not resolve the issue. After the second manual correction, validation succeeded, and the Pipeline continued the execution as expected.

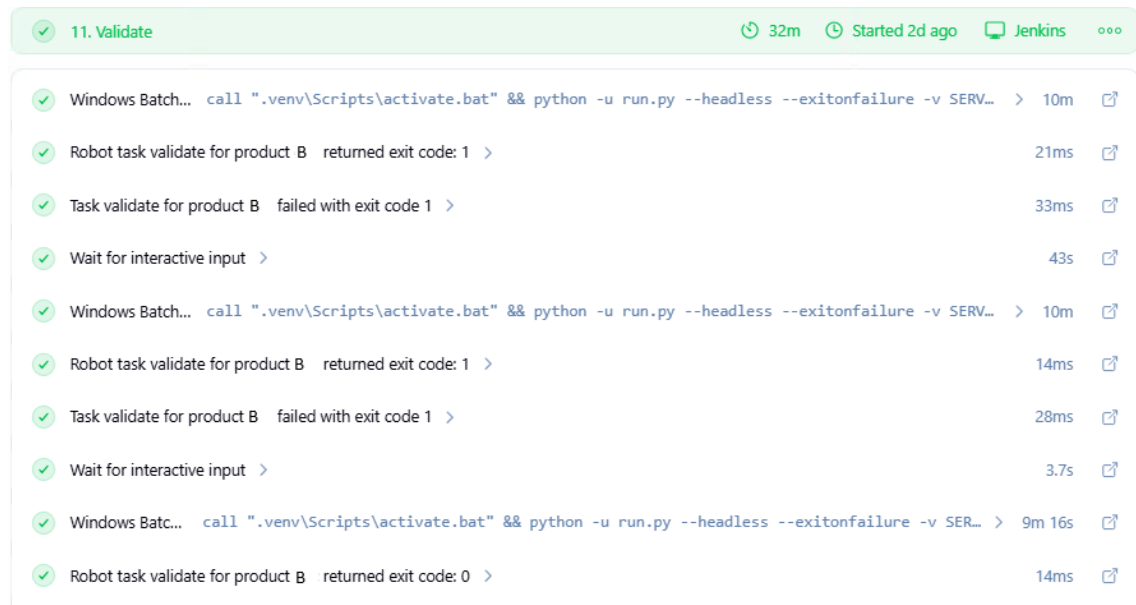


Figure 37. Product B Pipeline validation stage phases with user interaction.

This confirms that the implementation successfully supports user intervention when a task fails. However, some limitations became evident. The user has to execute the fixes and respond to the prompt in a specific time frame, otherwise the Pipeline aborts. The intervention steps are also visible only in the Jenkins console output, not in the generated reports or logs. In a real project scenario, engineers may not be actively monitoring Jenkins, which limits how practical this interaction model is unless extended with notification mechanisms. This suggests that the user interaction model is technically feasible, but proposes usability challenges that must be addressed before applying the method in real customer projects.

These results support the conclusion that the PoC provides a viable foundation for automated product deployment. The system performs reliably in single and multi-product scenarios, and its behaviour during a failure demonstrates controlled handling of user intervention. The main limitations relate to the practical usability of the user intervention functionality, particularly its visibility and reliance on time-constraints. While further testing is needed before production adoption, the presented results confirm that the architecture and implementation choices made during development are technically feasible.

6.3 Engineer feedback

Discussions with project and development engineers indicated that the PoC implementation aligns well with the deployment workflow currently used in projects. The sequence of automated actions was viewed as realistic and functional in deployment projects. The generated reports were considered especially useful. This positive reception was largely due to the fact that the generated summary reports follow the same structure as existing system exports, making them familiar to engineers.

Engineers also identified several areas requiring further refinement. Usability issues in the Jenkins interface were highlighted, particularly the presence of non-essential interface elements and the lack of visibility when the Pipeline pauses for user input. Since the main view does not indicate that the Pipeline is waiting, users must access the console output to identify required actions. This reduces the transparency of actions and requires effort. Additionally, feedback emphasized the need to scale the implementation for multi-product deployments and to prevent a failure in one product from aborting the entire Pipeline. Robust error handling due to incomplete or incorrect product specification parameters was also deemed essential for real production scenarios.

Overall, the feedback indicates that while the PoC provides a strong foundation and aligns with current engineering practices, some enhancements related to usability, error handling, and scalability are necessary before broader adoption.

7. CONCLUSIONS

This thesis examined the deployment process of automation applications in a web-based DCS and developed a Proof of Concept to evaluate the feasibility of automating part of this workflow. The PoC combined a Jenkins Pipeline for process orchestration with Robot Framework tasks for executing product specific operations in the DCS. Test runs and feedback from project engineers showed that it successfully automated the deployment workflow in a controlled environment. These results demonstrate that the approach provides a solid foundation for further development toward a practical deployment tool capable of reducing manual work and minimizing error prone tasks.

The work addresses the research questions defined in Section 1.2. The first question focused on identifying the characteristics and limitations of the current deployment process for automation applications. The analysis showed that the workflow includes manual and repetitive stages that require engineering effort. In particular, the application generation stage was identified as a potential area for automation due to its repetitive structure. This confirmed that the existing process has some limitations that can be effectively improved through automation.

The second research question examined what an ideal deployment workflow utilizing automation should look like. Based on observations from analysing the existing process, an ideal automated deployment workflow was defined. This workflow emphasizes modular process execution, minimal manual intervention, and structured reporting. Automation was targeted particularly at phases involving mechanical repetition and systematic error detection. The proposed workflow applies modular automation layers that automate library imports, product instantiation, configuration, validation, and runtime deployment, reserving human involvement for tasks that require engineering expertise.

The third research question considered how different automation tools can contribute to designing and implementing such a workflow. The evaluation showed that Robot Framework, which is already widely used for testing productized applications on the DCS, is well suited for web-based automation. Jenkins provides a flexible orchestration platform with support for declarative Pipelines and integration of human interaction steps. Together, these tools aligned well with the technical environment and existing engineering practices within application deployment projects.

The final research question addressed how an automatic deployment tool can be designed, implemented, and evaluated in practice. The developed PoC demonstrated that partial automation of the deployment process is feasible. It successfully automated deployments for two products and produced clear and structured reports that engineers found useful. While the PoC reliably executed deployments for the tested products, the evaluation also revealed limitations related to usability, product coverage, and error handling. User interaction through Jenkins was identified as a bottleneck, as the platform offers limited visibility of paused pipeline execution and requires monitoring of the console output. In addition, the PoC currently supports only a limited set of products, and failures caused by incomplete or incorrect specification parameters can interrupt the pipeline. These observations highlight areas where future development is required to improve robustness and usability.

Overall, the results demonstrate that partial automation of the deployment process is both feasible and beneficial. The PoC offers a concrete foundation for improving efficiency, reducing manual effort, and establishing a more reliable and repeatable deployment process. Although additional development is needed to enhance usability, scalability, and error-handling capabilities, the findings indicate that automation can play a meaningful role in improving deployment practices in projects.

7.1 Further development

Future work should focus on enhancing the usability, robustness, and scalability of the PoC so that it can serve as a practical tool in real projects. The engineer feedback collected during testing highlights several key directions for improvement.

The usability of the Jenkins interface should be enhanced. The current view contains several elements that are not directly relevant to the deployment workflow. This makes navigation more difficult, especially for users who do not regularly work with Jenkins. Improving the clarity of the interface and making relevant functions easier to locate would support more efficient use. In particular, the visibility of user prompts should be enhanced. Jenkins does not clearly indicate when the pipeline is waiting for input, requiring users to open the console output to find the prompt. Making pipeline states more visible in the main view would reduce unnecessary navigation and improve overall workflow transparency.

Handling of error scenarios should also be further developed. In projects, deployment failures often originate from incorrect or incomplete parameters in the product specification files. The system should include checks that identify such issues and provide clear feedback on the cause of the error. Clearer error messages and better retry logic for known errors would make the pipeline more reliable and reduce manual troubleshooting work.

Scaling the implementation to support deployments involving multiple products is essen-

tial for real production scenarios. The PoC currently supports two products, but the implementation must be expanded to additional products to be viable in full-scale projects. The pipeline should also be improved so that failures in one product do not stop the entire deployment pipeline.

Finally, while Jenkins served effectively as an orchestration platform for the PoC, long-term development may benefit from evaluating alternative CI/CD or orchestration tools with stronger interaction or usability features. Platforms with more advanced UI capabilities, improved pipeline interaction models, or better integration with industrial engineering workflows could address many of the usability limitations identified during testing and engineer feedback.

The PoC serves as a strong starting point for building an automated deployment solution for automation applications. With further development in usability, error handling and scalability, the system has the potential to significantly reduce manual workload and improve consistency in deployment projects.

REFERENCES

- [1] B. M. Szeszák et al. “Industrial Revolutions and Automation: Tracing Economic and Social Transformations of Manufacturing”. In: *Societies* 15.4 (2025), p. 88. DOI: 10.3390/soc15040088.
- [2] Fortune Business Insights. *Industrial Automation Market Size, Share & COVID-19 Impact Analysis, By Component (Hardware, Software), By Industry (Discrete Automation, Process Automation), and Regional Forecast, 2022-2029*. URL: <https://www.fortunebusinessinsights.com/industry-reports/industrial-automation-market-101589> (visited on 10/31/2025).
- [3] A. Dearle. “Software Deployment, Past, Present and Future”. In: *Future of Software Engineering (FOSE '07)*. 2007, pp. 269–284. DOI: 10.1109/FOSE.2007.20.
- [4] AutoMQ. *Fully Automated Deployment vs. Manual Deployment: A Comprehensive Guide*. May 22, 2025. URL: <https://www.automq.com/blog/fully-automated-deployment-vs-manual-deployment-comparison> (visited on 10/09/2025).
- [5] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. 1st ed. Addison-Wesley Professional, 2010. 512 pp.
- [6] Netdata. *Deployment Automation: Benefits, Process, Tools & Best Practices*. URL: <https://www.netdata.cloud/academy/deployment-automation/> (visited on 10/10/2025).
- [7] M. Shahin, M. Ali Babar, and L. Zhu. “Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices”. In: *IEEE Access* 5 (2017), pp. 3909–3943. DOI: 10.1109/ACCESS.2017.2685629.
- [8] B. R. Mehta and Y. J. Reddy. *Industrial process automation systems: design and implementation*. 1st ed. Butterworth-Heinemann, 2015. 668 pp.
- [9] C. Dey and S. K. Sen. *Industrial automation technologies*. 1st ed. Boca Raton, FL: CRC Press, 2020. 287 pp.
- [10] A. Alghamdi and M. Niazi. “Toward Successful Secure Software Deployment: An Empirical Study”. In: *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. EASE '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 487–492. DOI: 10.1145/3593434.3593966.
- [11] S. Sharma and S. K. Pandey. “Integrating AI Techniques In SDLC: Design Phase Perspective”. In: *Proceedings of the Third International Symposium on Women in Computing and Informatics*. WCI '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 383–387. DOI: 10.1145/2791405.2791418.

- [12] P. S. Chatterjee and H. K. Mittal. “Enhancing Operational Efficiency through the Integration of CI/CD and DevOps in Software Deployment”. In: *2024 Sixth International Conference on Computational Intelligence and Communication Technologies (CCICT)*. 2024, pp. 173–182. DOI: 10.1109/CCICT62777.2024.00038.
- [13] M. Krief. *Learning DevOps : the complete guide to accelerate collaboration with Jenkins, Kubernetes, Terraform and Azure DevOps*. 1st edition. Birmingham, England: Packt, 2019. 504 pp.
- [14] Atlassian. *What Is DevOps?* URL: <https://www.atlassian.com/devops> (visited on 12/04/2025).
- [15] Atlassian. *Getting Git right*. URL: <https://www.atlassian.com/git> (visited on 12/04/2025).
- [16] C. Brindescu et al. “How do centralized and distributed version control systems impact software changes?” In: *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014, pp. 322–333. ISBN: 1450327567.
- [17] G. Hyun et al. “The Impact of an Automation System Built with Jenkins on the Efficiency of Container-Based System Deployment”. In: *Sensors* 24 (2024). DOI: 10.3390/s24186002.
- [18] M. Labouardy. *Pipeline as code : continuous delivery with Jenkins, Kubernetes, and Terraform*. Shelter Island, New York: Manning Publications Co., 2021. 528 pp.
- [19] Codefresh. *What Is Jenkins and How Does it Work? Intro and Tutorial*. URL: <https://codefresh.io/learn/jenkins/> (visited on 01/22/2026).
- [20] CD Foundation The Linux Foundation. *Glossary*. URL: <https://www.jenkins.io/doc/book/glossary/> (visited on 01/22/2026).
- [21] CloudBees. *Automate with Jenkinsfile*. URL: <https://docs.cloudbees.com/docs/cloudbees-ci/latest/pipelines/pipeline-as-code> (visited on 01/22/2026).
- [22] CD Foundation The Linux Foundation. *Pipeline*. URL: <https://www.jenkins.io/doc/book/pipeline/> (visited on 01/22/2026).
- [23] P. King. “A history of the Groovy programming language”. In: *Proc. ACM Program. Lang.* 4.76 (2020), pp. 1–53. DOI: 10.1145/3386326.
- [24] CloudBees. *Automate with Jenkinsfile*. URL: <https://docs.cloudbees.com/docs/cloudbees-ci/latest/automating-with-jenkinsfile/> (visited on 01/22/2026).
- [25] CD Foundation The Linux Foundation. *Using a Jenkinsfile*. URL: <https://www.jenkins.io/doc/book/pipeline/jenkinsfile/> (visited on 01/22/2026).
- [26] CD Foundation The Linux Foundation. *Defining execution environments*. URL: <https://www.jenkins.io/doc/pipeline/tour/agents/> (visited on 01/22/2026).
- [27] CD Foundation The Linux Foundation. *Pipeline Syntax*. URL: <https://www.jenkins.io/doc/book/pipeline/syntax/> (visited on 01/22/2026).
- [28] CD Foundation The Linux Foundation. *Pipeline: Input Step*. URL: <https://www.jenkins.io/doc/pipeline/steps/pipeline-input-step/> (visited on 01/22/2026).

- [29] M. Schneider, B. Götz, and T. Bauernhansl. "A Framework for Software Deployment in Manufacturing Environments". In: *Procedia CIRP* 130 (2024), pp. 1043–1048.
- [30] M. Schneider et al. "Software deployment in manufacturing environments: A requirements analysis". In: *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2022, pp. 1–8. DOI: 10.1109/ETFA52439.2022.9921690.
- [31] N. Kiefl et al. "Real-Time Capable Architecture for Software-Defined Manufacturing". In: *Advances in Automotive Production Technology - Towards Software-Defined Manufacturing and Resilient Supply Chains*. Switzerland: Springer International Publishing AG, 2023.
- [32] A. Dubey. "Evaluating software engineering methods in the context of automation applications". In: *2011 9th IEEE International Conference on Industrial Informatics*. 2011, pp. 585–590. DOI: 10.1109/INDIN.2011.6034944.
- [33] SG Systems. *Factory Acceptance Testing (FAT)*. Oct. 2025. URL: <https://sgsystemsglobal.com/glossary/factory-acceptance-testing-fat/> (visited on 02/25/2026).
- [34] M.D. Li, A. F. Nicolescu, and C. Pupz. "VIRTUAL COMMISSIONING OF MANUFACTURING SYSTEMS. A REVIEW". In: *Proceedings in Manufacturing Systems* 19 (2024), pp. 91–95. ISSN: 2067-9238.
- [35] Inc. The MathWorks. *What Is Hardware-in-the-Loop (HIL)?* URL: <https://www.mathworks.com/discovery/hardware-in-the-loop-hil.html> (visited on 01/29/2026).
- [36] B. Schofield, E. Blanco, and J. Borrego. "Continuous Integration for PLC-based Control System Development". In: *JACoW ICALEPCS2021* (2022), pp. 478–483. DOI: 10.18429/JACoW-ICALEPCS2021-TUPV035.
- [37] N. Ericsson et al. "Challenges from research to deployment of industrial distributed control systems". In: *2016 IEEE 14th International Conference on Industrial Informatics (INDIN)*. 2016, pp. 68–73. DOI: 10.1109/INDIN.2016.7819136.
- [38] T. Blüher et al. "DEVOPS FOR MANUFACTURING SYSTEMS: SPEEDING UP SOFTWARE DEVELOPMENT". In: *Proceedings of the Design Society* 3 (2023), pp. 1475–1484. DOI: 10.1017/pds.2023.148.
- [39] S. K. Nalluri and V. Teja Bathini. "Integrating Jenkins and AWS in Siemens Opcenter MES: Bridging the digital divide through modern development and deployment pipelines". In: *World Journal of Advanced Engineering Technology and Sciences* 15 (2025), pp. 2064–2074. DOI: 10.30574/wjaets.2025.15.2.0767.
- [40] Siemens. *Totally Integrated Automation Portal - Always ready for tomorrow*. URL: <https://www.siemens.com/global/en/products/automation/industry-software/automation-software/tia-portal.html> (visited on 11/15/2025).
- [41] Siemens. *TIA Portal Add-Ins*. Nov. 19, 2025. URL: <https://support.industry.siemens.com/cs/document/109773999/tia-portal-add-ins?dti=0&lc=en-FI> (visited on 11/21/2025).

- [42] Rockwell Automation. *FactoryTalk AssetCentre Securing Assets to Your Most Important Systems*. URL: <https://www.rockwellautomation.com/en-us/products/software/factorytalk/maintenancesuite/assetcentre.html> (visited on 11/15/2025).
- [43] Rockwell Automation. *FactoryTalk AssetCentre Digital Brochure*. Apr. 9, 2019. URL: https://literature.rockwellautomation.com/idc/groups/literature/documents/pp/ftalk-pp001_-en-p.pdf (visited on 11/15/2025).
- [44] Microsoft. *Azure DevOps*. URL: <https://azure.microsoft.com/en-us/products/devops> (visited on 11/15/2025).
- [45] GitLab. *Accelerate delivery with CI/CD automation*. URL: <https://about.gitlab.com/solutions/continuous-integration/> (visited on 11/15/2025).
- [46] Red Hat. *How Ansible works*. URL: <https://www.redhat.com/en/ansible-collaborative/how-ansible-works> (visited on 11/15/2025).
- [47] Ansible project contributors. *Introduction to Ansible*. Nov. 20, 2019. URL: https://docs.ansible.com/projects/ansible/latest/getting_started/introduction.html (visited on 11/21/2025).
- [48] JFrog. *8 Reasons for DevOps to use a Binary Repository Manager*. URL: <https://jfrog.com/whitepaper/devops-8-reasons-for-devops-to-use-a-binary-repository-manager/> (visited on 11/21/2025).
- [49] JFrog. *Manage AI and Software Artifacts at Scale*. URL: <https://jfrog.com/artifactory/> (visited on 11/21/2025).
- [50] J. J. Huang et al. "An Automated Deployment Scheme with Script-Based Development for Cloud Manufacturing Platforms". In: *Intelligent Information and Database Systems*. Vol. 10751. Lecture Notes in Computer Science. Switzerland: Springer International Publishing AG, 2018, pp. 489–499. ISBN: 9783319754161.
- [51] C. Siebra et al. "Empowering Continuous Delivery in Software Development: The DevOps Strategy". In: *Software Technologies*. Vol. 1077. Communications in Computer and Information Science. Switzerland: Springer International Publishing AG, 2019, pp. 247–265. ISBN: 9783030291563.
- [52] O. Oluwagbemi et al. "An Analysis of Scripting Languages for Research in Applied Computing". In: *2013 IEEE 16th International Conference on Computational Science and Engineering*. 2013, pp. 1174–1180. DOI: 10.1109/CSE.2013.174.
- [53] S. Schlund et al. "Robotic Process Automation in Industrial Engineering: Challenges and Future Perspectives". In: *Advances in Manufacturing, Production Management and Process Control*. Vol. 274. Lecture Notes in Networks and Systems. Springer International Publishing AG, 2021, pp. 320–327. ISBN: 9783030804619.
- [54] R. Syed et al. "Robotic Process Automation: Contemporary themes and challenges". In: *Computers in Industry* 115 (2020). DOI: 10.1016/j.compind.2019.103162.
- [55] Robot Framework ry. *Robot Framework*. URL: <https://robotframework.org/> (visited on 12/01/2025).

- [56] Robot Framework Foundation. *Robot Framework User Guide Version 7.3.2*. URL: <https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html> (visited on 12/01/2025).
- [57] Robot Framework Foundation. *1.1 Purpose / Use Cases*. URL: <https://robotframework.org/robotframework-RFCP-syllabus/docs/chapter-01/purpose> (visited on 01/21/2026).
- [58] Robot Framework Foundation. *1.2 Architecture of Robot Framework*. URL: <https://robotframework.org/robotframework-RFCP-syllabus/docs/chapter-01/architecture> (visited on 01/21/2026).
- [59] Robot Framework Foundation. *1.3 Basic Syntax & Structure*. URL: <https://robotframework.org/robotframework-RFCP-syllabus/docs/chapter-01/syntax> (visited on 01/21/2026).
- [60] Robot Framework Guides Built with Docusaurus. *Project Structure*. URL: https://docs.robotframework.org/docs/examples/project_structure (visited on 01/21/2026).
- [61] Alena Galysheva. *Introduction to Test Automation with Robot Framework*. Mar. 31, 2024. URL: <https://wimma-capstone.pages.labranet.jamk.fi/support-material/4.%20TEST/3.%20Tools%20for%20testing/Tool%20-%20Robot%20Framework/introduction-to-rf/> (visited on 01/21/2026).
- [62] Robot Framework Foundation. *2.1 Suite File & Tree Structure*. URL: <https://robotframework.org/robotframework-RFCP-syllabus/docs/chapter-02/suitefile> (visited on 01/21/2026).
- [63] *Robot Framework documentation*. URL: <https://robotframework.org/robotframework/#standard-libraries> (visited on 01/21/2026).
- [64] Robot Framework Foundation. *2.4 Keyword Imports*. URL: https://robotframework.org/robotframework-RFCP-syllabus/docs/chapter-02/keyword_imports (visited on 01/21/2026).
- [65] Robot Framework community. *SeleniumLibrary*. URL: <https://robotframework.org/SeleniumLibrary/SeleniumLibrary.html> (visited on 01/16/2026).
- [66] Robot Framework community. *SeleniumLibrary*. URL: <https://github.com/robotframework/SeleniumLibrary> (visited on 01/16/2026).