Informatics

Dynamic Deployment of Fault Detection Models

A Use Case of the Asset Administration Shell

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Dominik Mailer, BSc
Matrikelnummer 01634043

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Mitwirkung: Univ.Ass. DDipl.-Ing. Dr.techn. Gernot Steindl

Wien, 17. Juli 2023

Dominik Mailer   Wolfgang Kastner

Informatics

Dynamic Deployment of Fault Detection Models

A Use Case of the Asset Administration Shell

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Dominik Mailer, BSc
Registration Number 01634043

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Assistance: Univ.Ass. DDipl.-Ing. Dr.techn. Gernot Steindl

Vienna, 17th July, 2023

Dominik Mailer Wolfgang Kastner

Erklärung zur Verfassung der Arbeit

Dominik Mailer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 17. Juli 2023

Dominik Mailer

Acknowledgements

First of all, I would like to thank my advisors Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner and Univ.Ass. DDipl.-Ing. Dr.techn. Gernot Steindl for supervising this thesis and for their valuable and constructive feedback. I would especially like to thank Gernot Steindl for the input and results of discussions with him on various aspects of this work, which were very helpful in writing this thesis and contributed significantly to its completion.

I would also like to thank my family for their moral support during my studies. In particular, I would like to thank my parents and my girlfriend for their motivating words and their encouragement to complete this work. In addition, I want to thank my girlfriend for her support in proofreading this thesis.

Kurzfassung

Industrie 4.0 kann als Paradigmenwechsel im Bereich der industriellen Produktion gesehen werden, der zur Digitalisierung von Fertigungsprozessen führt. Zu diesem Zweck bietet das Konzept des digitalen Zwillings einen strukturierten Ansatz für den Austausch und die Abbildung von Daten in diesem Kontext. Hierzu zählen auch Sensordaten, die mittels maschinellen Lernens (ML) beispielsweise zur Erkennung von Fehlern in Produktionsprozessen genutzt werden können. ML-Modelle müssen jedoch bei Bedarf aktualisiert und anschließend produktiv eingesetzt werden. Allerdings mit der Einschränkung, dass Produktionssysteme zu diesem Zweck nicht einfach angehalten werden können. Deshalb müssen diese Aktualisierungen so durchgeführt werden, dass der Wechsel zwischen den Modellen ohne Unterbrechung der Produktionsprozesse erfolgen kann.

Diese Arbeit zeigt einen Ansatz zur Integration eines ML-Modells zur Fehlererkennung in eine Verwaltungsschale (AAS), die eine standardisierte Implementierung eines digitalen Zwillings ist. Der Schwerpunkt liegt dabei auf der kontinuierlichen Bereitstellung aktualisierter Versionen dieses Modells während des Betriebs, ohne dass es zu Ausfallzeiten in den Produktionsprozessen kommt. Die wichtigsten Ergebnisse sind die Identifizierung von Anforderungen an eine dynamische Modellbereitstellungs- und Wechselstrategie auf der Grundlage einer Literaturrecherche und ein ML-Modell-Integrationskonzept für die AAS und eine prototypische Implementierung mit Eclipse BaSyx, um die identifizierten Bereitstellungsstrategien sowohl qualitativ als auch quantitativ bewerten zu können.

Zu diesem Zweck wurde ein generisches Teilmodell basierend auf dem AAS-Metamodell entwickelt, das verschiedene Arten von ML-Modellen sowie Anwendungsszenarien unterstützen kann. Darüber hinaus wurden drei Bereitstellungsstrategien ermittelt, die einen Wechsel zwischen Modellversionen ohne Ausfallzeiten ermöglichen. Bei diesen Strategien, die im Prototyp implementiert wurden, handelt es sich um zwei Blue-Green Deployments und ein Rolling-Deployment-Konzept. Die entwickelten Konzepte und Ansätze wurden anhand des Prototyps veranschaulicht, um die Einsetzbarkeit und Eignung des Ansatzes zu verdeutlichen. Die Ergebnisse unterstreichen, dass es keine Einheitslösung gibt und dass die am besten geeignete Bereitstellungsstrategie vom jeweiligen Anwendungsfall abhängt. Darüber hinaus sind alle drei beschriebenen Bereitstellungsstrategien dem Basisansatz, einem manuellen Bereitstellungsansatz unter Verwendung des Recreate Deployment Patterns, überlegen. Sie übertreffen diesen sowohl bei der kriterienbasierten Bewertung als auch beim Vergleich der Ausführungszeiten.

Abstract

Industry 4.0 can be seen as the new paradigm shift in the field of industrial production, leading to the digitalization of manufacturing processes. To this end, the concept of digital twins provides a structured approach for exchanging and representing data in this environment. This includes sensor data that can be used by techniques such as machine learning (ML) to detect faults in production processes, among other things. However, ML models need to be updated or retrained and deployed as needed but with the limitation that production systems cannot simply be stopped for this purpose. Therefore, these updates must be performed in such a way that switching between models can be accomplished without interrupting production processes.

This thesis describes an approach for integrating a ML model for fault detection into an Asset Administration Shell (AAS), which is a standardized implementation of a digital twin, focusing on the seamless deployment of updated versions of this model during operation without causing downtime in production processes. The main contributions of this thesis are therefore the identification of requirements for a dynamic model deployment and switching strategy based on a systematic literature review, a ML model integration concept for the AAS, and a prototype implementation using Eclipse BaSyx to perform both a qualitative and quantitative evaluation of the identified deployment strategies.

To this end, a generic submodel was developed that conforms to the AAS metamodel definition and can support different types of ML models and application scenarios. In addition, three deployment strategies were identified that allow switching between model versions without downtime. These strategies described in detail and implemented in the prototype are two blue-green deployments and one rolling deployment approach. The evolved concept and approaches were demonstrated by means of a prototype to illustrate the applicability and suitability of these artifacts. The findings emphasize that there is no one-size-fits-all solution and that the most appropriate deployment strategy depends on the specific use case. In addition, all three deployment strategies described are superior to the baseline approach, a manual deployment approach using the recreate deployment pattern, as they outperform it in both criteria-based evaluation and comparison of execution times based on the prototype.

Contents

CHAPTER 1
Introduction

1.1 Motivation and Problem Statement

Industry 4.0 can be seen as the new paradigm shift in the field of industrial production. The utilization of information and communication technology to create modular and efficient manufacturing systems [LFK+14] and the intelligent networking of machines and processes are fundamental aspects of Industry 4.0 [Pla23]. To generate added value, the sensor data, which is initially only available locally, must be transferred from machines and production facilities to other components. This end-to-end digitalization of manufacturing processes leads inevitably to cross-layer interactions and thus requires a structured approach for exchanging and representing data. The concept of the digital twin addresses these requirements for structured cross-layer communication within Industry 4.0 [SKA20].

A digital twin is the virtual representation of an asset that covers its whole lifecycle and provides access to the asset's data and services. These assets are physical entities of the real world, such as a product to be produced or a machine. Digital twins aggregate (real-time) data that is generated in the physical world. This data can be used for simulations, machine learning applications, decision-making support and for providing insights that may in turn affect the entities in the physical world as updated configurations in order to optimize a whole production plant [SKA20][Arm20].

One standardized implementation of digital twins is the Asset Administration Shell (AAS)[1]. The AAS is defined as a technology-independent metamodel that enables the modeling of assets as their digital representation. It encapsulates the information and functionalities of the assets, such as their properties, measurement data, real-time data, and capabilities [NOP22]. Subsequently, the sensor data encapsulated in AAS models can be used to detect faults in production processes to reduce the costs of monitoring

[1] `https://industrialdigitaltwin.org/en/`

and maintaining production equipment, since manual fault detection is complicated, inefficient, and lacks real-time capability [HDZ+20].

Improvements in sensor, storage and communication technology are enabling the transformation towards smart factories. Due to technological advancements, projects are emerging that leverage the Big Data collected and apply machine learning techniques to improve fault detection in production environments. Proposed fault detection approaches are based, for example, on Bayesian networks, artificial neural networks, deep neural networks, support vector machines, linear classifiers with linear discriminant analysis, or on the hidden Markov model as an extension of Markov chains [ABP17], [HDZ+20]. Other proposed techniques for fault detection models are ensemble learning [LDKC19] or deep transfer learning [XSLZ19].

However, machine learning models need to be updated or retrained when their performance degrades. Reasons for performance degradation include concept drift and model aging. This means that the learned feature distribution shifts over time due to changes in the data generating processes, resulting in misclassified error states [ZEK21]. In the context of production environments, machine components age over time, are maintained or replaced. Therefore, all these aspects influence the fault patterns to be identified. Thus, the stronger the model drifts, the more likely it is that incorrect error classifications will occur [LDKC19]. To continuously ensure a given classification accuracy, fault detection models must be retrained and deployed as needed. However, production systems cannot simply be stopped when a newly trained model needs to be deployed somewhere in the system as this would incur high costs for the corresponding processes. Therefore, the updates must be performed in such a way that the switch from the old to the new version of a model can be accomplished in a limited period of time without interrupting the production process [KBA+21].

Although there exists substantial research on various fault detection approaches for different areas in production [ZXW+14], [ABP17], [LSA+19], [XSLZ19], [HDZ+20], [SbAT20], [SLJK21], and some research on concept drift detection and machine learning model update strategies [LDKC19], [ZEK21], only a limited number of papers addresses deployment approaches for machine learning models [Ben20], [GPR+21]. To the best of our knowledge, there is currently no research on combining AAS concepts with dynamic deployment strategies for machine learning models and switching between different model version that aim to achieve zero downtime for production systems. In addition, there is little research on topics related to the AAS and, in particular, on the incorporation of machine learning models into AAS submodels. The latter can also be of interest to machine manufacturers, since fault detection models incorporated into the AAS can be handed over to customers directly in the AAS, along with other machine properties and documentation. Thus, they do not need any further information about the internal fault detection process.

1.2 Aim of the Work

A significant amount of research has been conducted to develop different approaches of high performing fault detection models in the production environment. Some papers also address concept drift detection and appropriate update strategies to maintain the desired level of quality. However, structured approaches for deploying new or retrained machine learning models in a continuously operating production environment are rarely mentioned. It can be assumed that the further development of the concept of digital twins and in particular of the AAS will also expand the application possibilities and increase interoperability. This also requires the ability to dynamically deploy machine learning models in production environments while striving for zero downtime. Therefore, the overall goal of this thesis is to develop a concept for the integration of data-driven fault detection models into the AAS in order to subsequently enable the exchange of these models during runtime with minimal downtime.

Based on the above objectives, the following research questions can be formulated:

- **RQ1:** Which deployment strategies for Machine Learning (ML) models and dynamic update methods for software components are mentioned in existing literature or have been applied in production environments that could be transferred to the AAS?
- **RQ2:** Given the current state of the AAS, what is a feasible approach to integrate a fault detection model into the AAS considering that the models should be called directly from the edge nodes that contain the real-time data?
- **RQ3:** What is an appropriate strategy to deploy a fault detection model in a production environment and switch from the previous to the new model without stopping production processes?

RQ1 sets the focus for the literature review to rigorously determinate the current state of the art on these research topics and, based on that, begin to develop an integration approach. An important aspect of conceptualizing the integration into an AAS is that it should be possible to run the fault detection model on a node close to the devices that actually emit the real-time data. Therefore, *RQ2* restricts the possibilities to define a viable integration approach. Subsequently, requirements must be defined for the dynamic deployment of fault detection models in production environments and for the non-disruptive switch between the previous and the updated version of the models. Based on these requirements, possible deployment and model switching strategies derived from such approaches in other areas will be evaluated. Again, these requirements limit the options that can be accepted as an appropriate strategy in terms of *RQ3*.

Accordingly, this thesis aims to fill the gap between the development of fault detection models, including their need to account for model drift, and the deployment strategies of such models, considering the possibility of integration into an AAS. The following paragraphs summarize the expected results:

- **Identification of Requirements for Model Deployment and Switching** The first step is to define requirements for deploying new versions of fault detection models and switching from the old to the new version of these models (e.g., time constraints depending on cycle times or management overhead). These requirements include also the aspect of the model's integrability into an AAS that takes into account the deployment on edge nodes, implementation effort and complexity, and the constraints imposed by this requirement on the dynamic deployment strategy.

- **Model Integration Concept for the AAS** The fault detection model should be integrated into an AAS. Thus, the task is to identify how such a model can be integrated into the concept of the AAS using the existing metamodel elements. A key requirement in this regard is that the fault detection model should be deployed directly on a node at the edge of the network, i.e., in close proximity to the assets that generate the data.

- **Identification of Deployment and Switching Strategy** A further outcome of this work is the conception of strategies for deploying an updated version of a fault detection model and switching from the outdated version to the new version of the model without stopping or interrupting the production process.

- **Evaluation of the Deployment Strategies and AAS Integration Concept** In order to evaluate the identified deployment and switching strategies for a fault detection model, as well as the approach for integrating such a model into an AAS, a prototype is implemented that demonstrates how such an architecture looks like and how fast a model switch can be performed compared to the baseline method of stopping the system, replacing the model, and restarting the system again. For the implementation of this proof of concept, the Eclipse BaSyx Framework[2] will be used as it is quite an extensive implementation of the AAS.

1.3 Structure

The following section provides a brief overview of the structure of this thesis. In Chapter 2, the methodologies used are described, especially for the SLR and the evaluation. The background is elaborated in Chapter 3 covering essential topics. Based on the performed SLR, the related work is discussed in Chapter 4. A list of relevant criteria and requirements, both for the integration concept and the dynamic model deployment and switching strategies is defined in Chapter 5. In Chapter 6, the approach for integrating a fault detection model into an AAS is described, while Chapter 7 presents relevant model deployment and switching strategies. The evaluation of the prototype and the results of the technical experiments are presented in Chapter 8 and are discussed in detail in Chapter 9. Finally, in Chapter 10 the insights and findings gained are summarized and ideas for future work are pointed out.

[2]https://www.eclipse.org/basyx/

CHAPTER 2

Methodology

This chapter describes the methodologies used in this thesis. The methodological approach followed is the Design Science Framework by Hevner et al. [HMPR04]. Within this framework there is first a Systematic Literature Review based on Kitchenham et al. [KC07] conducted that is reduced to the main parts. For the evaluation, a qualitative evaluation of the concept is done based on the defined requirements. Additionally, a prototype is implemented that proves that the concept actually works and also demonstrates the utility and suitability of this approach. In the following, the respective methods used within the Design Science Framework are described in detail.

2.1 Systematic Literature Review

The first step is to perform a literature review in order to identify the current state of deploying ML models. Furthermore, the current research on dynamic update approaches of software components in general will be investigated. In order to dynamically update implemented ML models in a continuously operating production environment, with the aim of having minimal downtime during this process, it will be necessary to link and combine the findings on deployment strategies and upgrading approaches for ML models with the approaches on dynamic update methods for software components.

In order to systematically analyze the existing literature and provide an overview of the topics mentioned a Systematic Literature Review (SLR) based on Kitchenham et al. [KC07] is conducted. However, since it would otherwise be too much overhead for this thesis, only certain steps of this SLR approach are selected and applied accordingly. The relevant steps of the SLR that are followed in this thesis are:

1. Formulation of the research questions with regard to the identification of the related work

2. Selection of appropriate search keywords
3. Definition of inclusion and exclusion criteria
4. Selection and reading of papers
5. Descriptive synthesis of the extracted information
6. Summarizing the findings

2.1.1 Research Questions for the SLR

The formulation of research questions is an essential step in a SLR process. In order to provide a solid foundation of the existing research to build upon and fill the gap in the existing literature, the following research questions for the SLR can be formulated:

- **RQ1a:** Which deployment strategies for ML models have been published by other researchers or are suggested by practitioners and ML experts?
- **RQ1b:** Which dynamic update methods for software components in the production environment are mentioned in existing literature and which software frameworks do they use?
- **RQ1c:** Which requirements are mentioned in the literature that are relevant for dynamic deployments of software components in production environments?

It can be seen by the abbreviations of these questions that they should be understood as part of *RQ1*, as *RQ1* covers the frame of the literature review to rigorously determine the current state of research on these topics.

2.1.2 General Inclusion and Exclusion Criteria

In the following, general criteria are defined that determine whether a paper remains in the SLR process or is excluded thereof.

Inclusion Criteria In order to be evaluated in detail, a paper must

- be written in English or German.
- have a title that implies relevance to the research question.
- have an abstract that implies relevance to the research question.
- be fully accessible.

Exclusion Criteria Papers are explicitly excluded from the SLR process

- if they have not been peer reviewed yet, for example, if they have only been published on *arXiv*[1].
- if the title or abstract does not indicate answering the research question in a useful way.

Further inclusion and exclusion criteria, more specific to the particular question, are defined in the respective sections.

Most of the scientific literature eligible for answering the research questions can be found in major online libraries and publishing sites. Therefore, the following publication portals were used as sources for the SLR:

- IEEE Xplore[2]
- ACM Digital Library[3]
- Scopus[4]
- Springer Link[5]

2.1.3 Foundation for the Requirements Definition

In addition to provide a sound foundation on these topics, the SLR also contributes towards defining criteria and requirements for the artifact to be developed. Based on the obtained findings, relevant aspects for the integration of an ML model for fault detection as well as for the deployment of such models with a special focus on dynamic updates are derived, which subsequently are formulated as criteria and requirements that the artifact should meet.

These criteria and requirements involve aspects that are essential for switching between ML models in the context of production environments, such as striving for zero downtime. In addition, requirements for the integration of such models into an AAS will be determined. A key necessity in this context is the requirement to deploy the model on edge nodes. These evaluation criteria are mainly qualitative ones.

2.2 Evaluation

The created artifact is evaluated in two ways. On the one hand, a criteria-based evaluation according to Cronholm and Goldkuhl [CG03] is conducted. The integration concept

[1]https://arxiv.org/
[2]https://ieeexplore.ieee.org/
[3]https://dl.acm.org/
[4]https://www.scopus.com/
[5]https://link.springer.com/

and the identified approaches for deploying and switching between the model versions are evaluated according to the criteria defined at the beginning. On the other hand, a prototype evaluation is carried out following Peffers et al. [PRTV12]. In addition, the prototype is also used for a quantitative evaluation of the identified deployment strategies by measuring the duration of the steps required to deploy a new model. This will also enable a comparison of the different strategies. Furthermore, by comparing it to a baseline approach, it is possible to assess how well the identified deployment strategies solve the motivated problem.

2.2.1 Criteria-Based Evaluation

This type of evaluation uses a checklist approach to evaluate the artifact first. In general, criteria are more universally applicable because they are not derived from a specific organizational context. Thus, criteria-based evaluation is ideally deductive [CG03].

After the concept for integrating a fault detection model into an AAS is completed and the strategies for deploying new model versions without any interruption are identified, they are evaluated against the list of defined criteria and requirements of the respective categories. On the one hand, this is done for the integration concept, and on the other hand, all identified deployment strategies are evaluated individually to enable a subsequent comparison. The result of this evaluation step is compared in a table. The rows contain the criteria and requirements, while the columns contain the integration concept and the deployment strategies. The degree of fulfillment is evaluated in three grades: (+) completely fulfilled, (○) partially fulfilled, and (–) not fulfilled.

2.2.2 Evaluation based on the Implemented Prototype

As part of the prototypical evaluation, the integration concept will be implemented together with the identified deployment strategies using the BaSyx platform in combination with Docker for containerization as well as other required software components. The implemented prototype should prove that the developed concept actually works and also demonstrate the utility and suitability of the approach in a rather abstract and synthetic environment. In addition, technical experiments can be conducted based on the prototype to evaluate the technical performance of the identified deployment strategies [PRTV12]. This can be compared to the baseline approach, which is a recreate deployment pattern.

The outcome of this evaluation is twofold: an executable implementation of the artifact that satisfies the predefined criteria and requirements, and technical experiment runs that allow measuring the performance of the deployment strategies in terms of the execution time of the individual steps required to prepare a new model version in an environment and eventually to switch between model versions. Tables for comparison and graphical representations of the measured values are used for a clear presentation of the latter results. There, the mean values as well as the standard deviations of the execution times of the test runs, grouped by steps, are presented.

2.2.3 Technical Details for Prototype Evaluation

This section deals with the implementation details of the prototype for the conceptual design of an AAS integration approach and dynamic deployment method for ML models for a Packed-Bed Regenerator.

Design Decisions and Selection of Software Tools and Components

For the implementation of the prototype, in particular the server-side implementation of the AAS, the Eclipse BaSyx™ Framework was chosen. At the moment, Eclipse BaSyx supports four programming languages. According to the project documentation, only the Java and C# SDKs provide the full functionality of BaSyx and are appropriate for the implementation of applications or services[6]. Because of that fact, but also due to the circumstance that many examples for different use cases are available on the Eclipse project page as well as on GitHub and on the Internet, the SDK for Java was chosen.

For containerization Docker and Kubernetes were chosen, as the were mentioned in the literature, but also because both are quite easy to install, set up, and get started with. Furthermore, ready to use components for an AAS Server and Registry are provided as Docker Containers. However, a self-hosted AAS Server and Registry were used for the evaluation of the prototype, as this was required for pushing operations and custom code into the AAS as well as for debugging purposes.

Used Software Versions and Hardware Specifications

Table 2.1 summarizes the used versions of the software components utilized in the course of implementing and evaluating the prototype as well as the hardware specifications of the system on which the evaluation took place.

joblib vs. ONNX vs. Other Serialization Formats For the persistence of the trained fault detection ML models, *joblib* serialization is used. *joblib* is based on pickle serialization, but improves upon it by being more efficient with respect to objects that internally contain large numpy arrays, which is often the case with fitted scikit-learn estimators. However, with *joblib* it is only possible to serialize on disk, and there are security and maintainability limitations as well [Sci23]. But these limitations are not an issue for the implemented prototype and, therefore, *joblib* is completely sufficient for this use case.

Other possible formats or libraries for persisting ML models are, for example, *skops*, which is not based on *pickle* and allows specifying trusted types, or other formats aimed at interoperability. These include ONNX or PMML, for instance. However, since this use case does not involve switching between different machine learning frameworks, switching between different computing architectures, or emphasizing readability through the use of an XML structure [Sci23], *joblib* is completely sufficient from this point of view as well.

[6]`https://wiki.eclipse.org/BaSyx_/_Overview`; last accessed: 2023-02-03

Component	**Version**
Windows	Windows 10 Pro - Version 22H2
Java	OpenJDK 16.0.1
Python	3.9
Eclipse BaSyx™	1.1.0
Docker Engine	20.10.22
Kubernetes	1.25.4
Traefik	2.9.1
Eclipse Mosquitto™	2.0.15
Processor	Intel® Core™ i5-4690K CPU @ 3.50GHz
System Type	64-bit operating system, x64 based processor
RAM	16 GB

Table 2.1: Used Software Component Versions and Hardware Specifications

Under the aspect of achieving the lowest possible level of complexity, a vote can be cast for *joblib* too.

2.2.4 Evaluation Approach of the Technical Experiments

This section explains the details of the approach in terms of the structure and procedure of the technical experiments to measure execution times and evaluate the identified deployment strategies.

Since no other infrastructure for testing was available and one machine was sufficient for the purpose of implementing the prototype, only this machine was used with the specifications listed in Table 2.1. This also ensured that identical conditions with regard to hardware and software specifications prevailed for all test runs of the deployment strategies. In addition, network-dependent influencing factors could be excluded. This means that Docker with all required containers, the AAS server along with the HTTP servers, MongoDB, and the evaluation program were running on the same machine.

Preparation of the Fault Detection ML Models In order to show that ML models can be integrated in this approach, but more importantly can be dynamically exchanged, several ML models had to be created. For this purpose, a total of seven models were trained on the same dataset of simulated sensor values. However, they were all of a different type of ML model, but still belonged to the overall class of classification models. The purpose of this was to demonstrate the suitability of this approach for different types of ML models, as long as they meet the predefined requirements. Regardless of the actual type of the model, these requirements are: (1) (De-)Serializability, (2) Containerizability, and (3) Inference via an API.

Since it was not the intention to create a high performing and accurate model for fault detection, especially since the database would not allow this anyway, simple models were deliberately chosen and no hyperparameter tuning was done. All ML algorithms used are from the *scikit-learn* library for Python. The following list summarizes the model types that were created for the prototype and thus also used for the test runs:

- Support Vector Machine Classifier (two models using different kernels)
- Decision Tree Classifier
- Random Forest Classifier (two models with different depth)
- KNeighbors Classifier
- Linear Discriminant Analysis

Evaluation Program The evaluation program is a script written in Python that executes a configurable number of test runs. In addition, a parameter can be used to define which deployment strategy is currently being evaluated. This has an effect on when which tests are executed regarding the active and standby environment. This evaluation program also logs all operations that are called in the AAS.

However, the actual time measurement is located on the AAS side. More precisely, in the individual implementations of the operations in BaSyx. This is done primarily for two reasons: First, this is the only way to measure time in sub-steps within an operation (e.g., sub-steps that are necessary to prepare the model in an environment). Secondly, this leaves out a layer that could cause delays, which have nothing to do with the actual deployment, namely, the network layer between the evaluation program and the fault detection submodel. If everything runs on one machine, as in this case, this factor may be negligible. However, if it runs on different nodes, latencies could influence the measured time and thus distort the result.

Test Run A test run is defined as a sequence of steps that automatically invoke the defined operations in the fault detection submodel. To this end, a test run uploads each of the seven previously trained and serialized models to the AAS, prepares the environment with this model and finally performs the switch between the models. After preparing the standby environment, it checks whether both the active and standby environments are still accessible.

Execution of Test Runs

The test runs for time measurement as part of the technical experiment are executed separately for each deployment strategy. This means for a test run, that the scenario is prepared specifically for a deployment and switching strategy (e.g., preparing the required Docker containers, adjusting the evaluation program, loading the customized AAS with the corresponding addresses and ports in the properties, etc.) and the operations are invoked automatically via the evaluation program.

For each deployment strategy identified, including the baseline approach with the recreate deployment pattern, two times ten test runs are performed. After ten test runs are performed, everything is reset and ten test runs are executed again. Therefore, in total 140 times (i.e., $2 \times 10 \times 7$) per deployment strategy a model is loaded into the AAS, the environment is prepared and the dynamic change is accomplished. If an error occurs unexpectedly during one of the called operations, the entire attempt for this one model is not considered and an additional loop run is made in order to always reach the total of 140.

In order not to fill up the AAS and thus also the stored models in the archive in the MongoDB unnecessarily in a short time during the test runs, the models of the previous test run (if already present) are deleted after each test run.

CHAPTER 3

Background

The following chapter explains the fundamental concepts, approaches and technologies that are used or serve as a basis in this thesis. First, an overview of Industry 4.0 and digital twins is presented, followed by a more detailed elaboration on the AAS as a standardized digital twin implementation that is employed in this thesis. Subsequently, the potential of ML for detecting faults is motivated and the challenge of concept drift is explained. Finally, a brief introduction on the basic characteristics of the Packed-Bed Regenerator is provided.

3.1 Industry 4.0 and Digital Twins

Industry 4.0, an ex-ante defined term for the fourth industrial revolution, is a disruptive paradigm shift in the manufacturing industry. The term is particularly well established in German-speaking countries and combines the two development directions of application-pull (e.g., short time-to-market, individualization, flexibility, decentralization, etc.) and technology-push (e.g., further mechanization and automation, digitalization, miniaturization) [LFK$^+$14]. Industry 4.0 aims to achieve an end-to-end digitalization of production processes and ensures greater flexibility and changeability in manufacturing, thereby reducing the costs of individualized products that are only produced in small batches and making their manufacture economically viable [SKA20]. With this paradigm, a shift from product to service orientation is expected and the emergence of new companies is anticipated [LFK$^+$14].

Numerous sensor data are already collected by machines, but they are only available locally and subsequent data processing is considerably hampered by the separate layers of the automation pyramid and their respective protocols. This is why Industry 4.0 opts for a peer-to-peer architecture to promote cross-layer communication. For example, an ERP system can directly receive sensor data to track the quality of a production process, skipping all the intermediate stages that would be required in the traditional automation pyramid. Further examples of use cases that take advantage of the described flexibility of Industry 4.0 are the significant reduction of the integration time for new devices into a manufacturing process and the dynamic change of manufactured products with the corresponding process adjustments - and all this with low effort and without downtime [SKA20]. In this way, scenarios are conceivable in which products control their own production processes [LFK+14].

Digital Twin To enable this cross-layer interaction, a structured approach for data representation and exchange is essential. Digital twins are the key to this requirement. They encapsulate data from the physical world and allow them to be used, for example, for tests or simulations in the digital world, and can in turn affect the physical asset based on the knowledge gained in the digital twin [SKA20].

One definition of digital twins by the Industrial Internet Consortium [MvSB+20] is that *“a digital twin is a formal digital representation of some asset, process or system that captures attributes and behaviors of that entity suitable for communication, storage, interpretation or processing within a certain context.”* Armstrong [Arm20] highlights a dynamic aspect and brings the lifecycle of an asset into play as a digital twin *“spans its lifecycle, is updated from real-time data, and uses simulation, machine learning and reasoning to help decision-making”* [Arm20].

Thelen et al. [TZF+22] have conducted a comprehensive literature review on the development and trends of the digital twin. The first appearance of the concept “digital twin” was mentioned already in the 1960s in NASA’s Apollo 13 program. Due to its increasing prevalence in various fields, many domain-specific definitions of digital twins have been proposed [TZF+22]. VanDerHorn and Mahadevan [VM21], based on their review of several definitions, presented a consolidated and generalized definition of digital twins as *”a virtual representation of a physical system (and its associated environment and processes) that is updated through the exchange of information between the physical and virtual systems”* [VM21].

Based on the above definition, three essential components defining a digital twin are: (1) a physical reality, (2) a virtual representation, and (3) connections for information exchange between the physical and virtual world [VM21]. Kritzinger et al. [KKT+18] have distinguished the terms digital model, digital shadow, and digital twin. The main difference between these three concepts is the way data flows between the physical and digital asset. In a digital model, the data flow in both directions is optional and mostly manual. While for a digital shadow the data flow is unidirectional from the physical to the digital asset with an optional data flow back, a digital twin requires a bidirectional data flow preferably automated [KKT+18].

3.2 Asset Administration Shell

This section discusses some general aspects of the AAS and provides a brief introduction to the main concepts and elements of the AAS metamodel that are used in this thesis.

The AAS is one standardized embodiment of a digital twin and its standardization is currently being carried out through a collaboration between Plattform Industrie 4.0[1] and other notable associations like the Industrial Digital Twin Association (IDTA)[2] and ZVEI[3]. Their published specification *Details of the Asset Administration Shell* in the version 3.0RC02 [Pla22] serves as basis for the following elaboration on the metamodel elements.

The general structure of an AAS is specified by a metamodel that is independent of any specific technology, but is supported by several technology-specific serialization mappings, including XML, JSON, or OPC UA. Therefore, an AAS enables interoperability between solutions from different vendors [NOP22]. Besides its technology- and device-independence it is furthermore a machine-readable description of an asset that provides access to all its properties and functionalities [Sch22a].

AASs are a key concept in Industry 4.0, enabling information hiding and providing an abstraction layer for technical or logical components. Therefore, AASs can be defined for physical as well as for non-physical entities, for example, for devices, products, but also for processes [Sch22a]. The content within an AAS, such as properties, measurement and real-time data, or capabilities, is structured in submodels [NOP22].

At the moment, three types of AASs can be distinguished regarding their interaction pattern [Sch22a], [NOP22]:

- **Type 1 - passive**: These are basically serialized files, for example in XML or JSON format, that contain static information and can be distributed as files. The metamodel specification [Pla22] primarily deals with this type of AAS and introduces the AASX package format.
- **Type 2 - reactive**: These AASs are available as runtime instances and are hosted on servers. They can contain static data as well as provide dynamically changing data. Therefore, AASs of type 2 can interact with other components. This type is mainly covered in the second part of the AAS documentation, which deals with API definitions for accessing the information contained in AASs.
- **Type 3 - proactive**: In addition to the characteristics of type 2 AASs, they contain active behavior, which means that they can communicate directly with other AASs independently of applications (peer-to-peer interaction).

AASX Package Explorer One standardized format for exchanging an AAS is the Asset Administration Shell Package (AASX) format that is derived from Open Package Conventions standards. This generic package file format contains the structure along with the data and supplementary files, such as manuals or CAD files [Pla22]. Based on

[1]https://www.plattform-i40.de/IP/Navigation/EN/Home/home.html
[2]https://industrialdigitaltwin.org/en/
[3]https://www.zvei.org/

this exchange format, Hoffmeister et al. have published an open source tool[4] that allows to graphically create new AASs or edit and view existing ones saved in the AASX format. It is an desktop application written in C# that is quite intuitive to use and provides some support by indicating potential violations of the standard through hints.

Identifiers for AASs In general, each element within an AAS requires an ID. Some of them have to be globally unique, while for others a short ID (a local identifier) is sufficient, since these elements are unique in combination with the ID of the parent elements. This IDs are needed, for example, to address specific elements in an AAS via the API [Pla22].

3.2.1 Submodel

Submodels serve to model specific aspects of an asset that is represented by the AAS and encapsulate their properties and features. They are one of the most important structuring elements within an AAS in terms of technical functionality and digital representation [Sch22a]. Standardized versions of submodels are called submodel templates.

A submodel is composed by several submodel elements, which can be of different types and are used to describe and distinguish asset characteristics. To ensure a well-defined meaning, each submodel element requires a semantic definition, which can be specified either directly via an external reference, such as ECLASS, or indirectly by providing a concept description [Pla22]. A submodel element is per se an abstract element and can therefore not be instantiated, which is why concrete element types are required. The currently existing submodel element types are quite manifold and can be clustered into certain groups. The most important ones in the context of this thesis are: DataElement (with concrete elements like Property, Blob, or ReferenceElement), Capability, Submodel Element Collection (SMC), Operation, and Event.

Submodel elements can in turn contain other submodel elements, creating an internal hierarchy and providing a way to organize elements. For this purpose, a SMC is the element of choice, since it is a collection of submodel elements and may allow duplicates [CS20b]. These collections are primarily used for cases that require a fixed set of properties with unique names [Pla22].

The *Property* submodel element can store a single value of a simple data type, such as string, date, integer, a floating point number, and so on. The data element *Blob* stores the content of a file directly in its value property as Binary Large Object (BLOB). In addition, the corresponding MIME type of the content must be specified. In contrast, the *File* submodel element stores only the file path. Another data element is the *ReferenceElement*, which defines a logical reference to another element that can be either in the same AAS or in any other AAS. It can also refer to external objects if they have a global unique ID [Pla22]. In the following sections, some important submodel elements are explained in more detail.

[4]https://github.com/admin-shell-io/aasx-package-explorer

3.2.2 Capability, Operation, Skill

A capability is an implementation-independent definition of an asset's potential to realize a desired impact in the physical or virtual environment [Pla22]. In addition, they should be independent of a particular asset and also of the AAS itself [BBB+20]. Typically, a capability is mapped to one or more skills which represent it either as a property or an operation. However, in more complex scenarios, it can be also mapped as a collection or an entire submodel [Pla22]. Skills are therefore defined on an asset-specific basis. Possible realizations of capabilities include state variables, trigger variables, operations, function blocks, and semantic protocols, ranging from simple possibilities to highly complex representations [BBB+20].

Defining capabilities and the corresponding operations is essential to perform capability-based engineering and operation. This means, for example, that the actual sequence of production processes is not determined in advance, but only after a capability check has evaluated the potential options and the feasibility check determines the option that can actually be used. For this purpose, it is necessary to provide the description of the capabilities in a machine-readable format using a formal semantic description [BBB+20].

Operations, as one possible implementation of skills, can be used to perform client/server interaction, for example. Function blocks, in contrast, are modeled in the AAS as a submodel. This submodel contains data elements for input and output parameters, the internal states, as well as operations to start, stop, or interrupt a function. Thus, the AAS essentially defines the information that can be used, but does not implement the actual functionality. This can be provided by additional files that are added as files or BLOBs [BBB+20].

An operation as an AAS metamodel concept defines the behavior of a component as a procedure. Several input and output variables as well as combined input/output variables can be defined, which are all of type *OperationVariable* and can have a submodel element of arbitrary type as a value. Such variables describe a provided parameter or the result of an operation [Pla22]. Operations can essentially be invoked by a simple method call. In an AAS, the defined generic and abstract capabilities are linked with *RelationshipElements* to the particular skills that actually provide that functionality [BBB+20].

3.2.3 Events

Events in the context of the AAS are very versatile and serve various purposes. These events are of different types and can represent input or output events with respect to the referred element. Input events are provided by an external massaging infrastructure and can be processed by the affected component. Whereas output events are broadcast to other AASs or external systems via an external massaging infrastructure. One possible use case for events is when a supplier wants to inform a customer that the firmware of an asset has been updated (forward events). Reverse events, on the other hand, are used when a supplier monitors the status of a device to detect possible incidents.

Events can also be used to track changes to components, which in turn can be collected centrally [Pla22].

In the AAS metamodel specification [Pla22], a list of event types has been defined, including structural changes, value changes of submodel elements, execution of operations, and so forth. In addition, custom event types can also be defined. While a general format of events has to be followed, the actual payload is fully customizable. The *BasicEventElement* inherits from the abstract *EventElement* and has several attributes defined. These include a reference to the observed element which thus defines the scope of the event, the direction, the state (i.e., on or off), the message subject, the broker, and additional timing information [Pla22].

Besides these aspects of events in the AAS metamodel specification [Pla22], there is very little documentation on the use of events in real use cases. Furthermore, the support for events in the AASX Package Explorer and BaSyx does not appear to be very extensive at this time either.

3.2.4 Eclipse BaSyx™

Eclipse BaSyx™ is an open source framework provided under the Eclipse Public License 2.0 (EPL 2.0). It provides a platform for the development of Industry 4.0 applications by facilitating the integration of heterogeneous devices based on a standardized data and information model to enable cross-technology communication. To this end, the BaSyx platform provides ready-to-use components (e.g., AAS server and registry) as well as SDKs for Java, .NET, C++, and Python that enable the development of individual components and applications, but also the integration of devices [Sch21].

BaSyx is a concrete implementation of the AAS metamodel, which enables the connection of all relevant components and ensures the required end-to-end digitalization. It has to be noted that BaSyx currently supports only version 2 of the AAS metamodel. In general, the defined AASs with their respective submodels and all elements contained therein are hosted on AAS servers and can be located via registries. On the one hand, these servers provide persistence via a backend storage functionality and, on the other hand, make the contained information available via a standardized interface. Therefore, a central component for communication between devices and systems is the Virtual Automation Bus (VAB) [Sch21]. A standardized API[5] is available to allow external applications and systems to easily access an AAS, including its submodels and elements, primarily through CRUD operations (create, read, update, and delete), but also to invoke operations.

The VAB enables end-to-end communication and reduces the implementation overhead for supporting different protocols, since VAB primitives serve as common base. In total, five primitives are defined for the VAB: (1) create, (2) delete, (3) retrieve, (4) update, and (5) invoke. Custom VAB implementations need to implement and map these five primitives in order to interact with VAB elements, like AAS objects and submodels, for example [Sch20].

[5]`https://app.swaggerhub.com/organizations/BaSyx`

3.3 Machine Learning based Fault Detection in Production Environments

In this section, the fundamentals of performing fault detection based on ML models with focus on production environments are presented. However, only a brief overview will be given, since this is a very broad field of research.

Fault detection, as used in this context, belongs to the category of fault diagnostics approaches mentioned in the literature. In particular, it aims to detect the presence of damage or defect that represents the current state of the system. In contrast, fault prognostics aims to assess the future health state of the system. Another approach to be distinguished in this area is predictive maintenance. This is a proactive approach in the area of plant maintenance that is based on the recognition of patterns in sensor data and attempts to predict when a machine component might fail in order to plan maintenance work in good time [TZF+22].

All of these tasks are value-adding processes and are necessary in the manufacturing environment because unexpected failures can cause high maintenance and downtime costs, reduce customer satisfaction, and even lead to injuries or fatalities [TZF+22]. In addition, most of the time during downtime is spent on locating the fault rather than fixing the actual problem [ABP17].

Further distinctions in the broader context can be made between reactive maintenance, where maintenance work is performed only after a fault has occurred, and preventive maintenance, where maintenance work is performed at regular intervals, which in most cases is more frequent than necessary [TZF+22].

ML based Fault Diagnostics The sensor data encapsulated in digital twins such as the AAS can be used to automatically detect faults in production processes to reduce the costs of monitoring and maintaining production equipment, as manual fault detection is complicated, inefficient, and lacks real-time capability since components are often inaccessible and the process requires human intervention [HDZ+20]. For this purpose, ML models are used, for example, to classify the health of a system, to identify the type of fault, or as a predictive tool to quantify the damage in a system. Both classical ML approaches and deep learning methods can be used to accomplish this tasks [TZF+22].

In a classical ML approach, features must first be extracted from the collected and preprocessed sensor data before training the model as a classification or regression task can be performed. However, these approaches also have some downsides, such as domain knowledge is required to extract features, and selecting the right features for a satisfactory overall performance is an iterative, tedious, and time-consuming process. Deep learning methods eliminate this task by automatically extracting features from large amounts of data. In many applications, deep learning approaches have performed better than conventional ML approaches for fault diagnostics [TZF+22].

Different approaches have been proposed in the literature for a wide variety of use cases, as they have different advantages and disadvantages. Approaches based on Bayesian networks are more intuitively understandable because they are white-box models, and support uncertainty modeling and hierarchical structures well. Artificial neural networks (ANN) are well suited for modeling nonlinear complex problems, but are more computationally intensive. Support vector machines are good for modeling both linear and nonlinear relationships and are faster than ANNs, but are more difficult to tune and incorporate domain knowledge. Hidden Markov models as a probabilistic model are computationally intensive for training, but are excellent for modeling processes with unobservable states [ABP17]. Also, linear classifiers with linear discriminant analysis can be used for fault detection [HDZ+20].

Further proposed approaches for fault detection in manufacturing include, among others, a combination of an adapted restricted Boltzmann machine for feature extraction and a deep neural network [HDZ+20], an ensemble learning algorithm based on a set of offline classifiers [LDKC19] as well as based on an ExtraTree classifier [SLJK21], a deep transfer learning method using a deep neural network as a basis [XSLZ19], a support vector data description method [ZXW+14], an online reduced rank kernel principal component analysis [LSA+19], or a reduced kernel partial least squares regression method [SbAT20].

Another aspect that should not be ignored for this topic is the fact that the quantity and quality of the data is essential for the performance of the models. Especially data representing incorrect states are more difficult and less frequent to acquire than those for correct states. This leads to an imbalance of the data and to a poor accuracy in terms of fault diagnostics [TZF+22]. Solutions to this problems include oversampling techniques applied to the minority class, such as the Synthetic Minority Oversampling Technique (SMOTE), or applying undersampling techniques to the majority class, which have the disadvantage of losing potentially important information [LDKC19]. Further approaches in the field of digital twins are based on physics and involve the generation of synthetic data (i.e., erroneous data is generated from a physical simulation and used to augment the training data set), physics-informed architecture or loss function design (inclusion of knowledge about the physical facts), or fault diagnostics based on physics-based modeling (for use cases where fault data is extremely rare) [TZF+22].

3.4 Problem of Data and Concept Drift

Since the ML models contained in a digital twin are often used in constantly changing environments, a situation arises where the distribution of the training data initially used is far off the data obtained live from the system (production data). Therefore, such a situation is also called “out-of-distribution”, since the assumption that the distribution of the training data is also representative for the production data is not true. Reasons for these changed conditions are, for example, operational changes due to the addition or replacement of sensors, adaptation of manufacturing processes, changes in the measurement method, and so on [TZF+22].

All of these aspects cause the underlying distribution to shift over time, and therefore the legacy mapping no longer fits the newly collected data [ZEK21]. This shift introduces a discrepancy between the data distribution of the training datasets and the observed data distribution. If these deviations are not integrated into the model, the performance of the ML model may deviate significantly. This degradation, also known as model decay, can be divided into two types: Data drift and concept drift [TZF+22].

Data Drift This drift type refers to a scenario in which the statistical characteristics of the production data differ significantly from those of the training data. Although the relationship between X and Y remains the same, X follows two distinct distributions for the training and production datasets. Ultimately, this leads to erroneous predictions of the model for the production dataset. Mathematically, data drift can be defined as $p^{train}(X) \neq p^{prod}(X)$ [TZF+22]. X, in this context, refers to the explanatory variables (features), while Y denotes the dependent variables.

Concept Drift This type of drift describes the effect that the underlying relationship between X and Y changes. Mathematically, concept drift can be defined as $p^{train}(Y|X) \neq p^{prod}(Y|X)$ [TZF+22]. In this context, a concept refers to the combined distribution of feature vectors and dependent variables that may change in response to replacement or aging of machine components, for example, resulting in a new joint distribution [LDKC19]. In this case, the model is also no longer reliable to a certain extent and will make incorrect predictions.

In practice, this leads to some challenges. To ensure consistent and satisfactory performance, the ML model needs to be updated when drift is detected. For example, by using a drift indicator, which is a quantitative application-specific metric. Several approaches have been proposed in the literature to address this problem, such as evaluating the divergence between training and production data, monitoring the variation in the quality of predictions, or calculating the mean error between predictions and actual measurements [TZF+22]. Which approach is most appropriate also depends on the ML model used.

Zufle et al. [ZEK21] have identified two groups of approaches to meet this challenge. The first is incremental model learning, which is well suited for rule-based models, support vector machines, tree-based ensembles, and neural networks, since only a part of the underlying concept such as rules, support vectors, trees, or weights need to be updated. The second group requires complete retraining. However, this requires a strategy on when to trigger the retraining. This can be done either at regular intervals, based on quality thresholds, or based on statistics of deviations [ZEK21]. In contrast, Lin et al. [LDKC19] proposed an ensemble learning-based approach that supports offline classifiers to deal with concept drift and imbalanced data.

3.5 Packed-Bed Regenerator

In this section, an overview on the Packed-Bed Regenerator is given. The Packed-Bed Regenerator is an illustrative industrial use case that initially served as a motivating example, helping to condense the first idea into the identified problem and the formulation of the problem statement. Additionally, it serves, in a simplified and scaled-down version, as an application example for the prototype that will be implemented in this thesis in order to evaluate the developed integration concept and the identified deployment strategies. To this end, this section summarizes the key points of the Packed-Bed Regenerator concept.

The diploma thesis by Michalka [Mic18] investigated the transient behavior of a Packed-Bed Regenerator and determined parameters such as storable energy, power and energy density for comparison purposes. Due to the need to be able to store energy, since supply and demand do not always coincide, energy storage systems play a major role in ensuring a continuous supply of energy. The heating and cooling sector represents the largest end-use energy consumption in Europe accounting for 50 percent. Sensible thermal energy storage systems, including the Packed-Bed Regenerator, are cost-efficient due to the use of water, stones, or other solid materials as fillings and, moreover, they do not contain toxic materials. The properties of these fillings are a significant performance factor for such regenerators. Their field of application are short to medium-term high-temperature heat storage and are employed in industrial furnaces or power plants [Mic18].

Figure 3.1 provides an overview of the rough structure of a Packed-Bed Regenerator based on a schematic sketch. This shows a charging process in which hot air is directed into the container from above in order to heat the bulk material. Technical details have been omitted intentionally, as the focus is only on the basic understanding of the concept and the most essential properties of such a thermal energy storage systems.

In the middle of Figure 3.1 is the container with the storage mass illustrated, which is filled with gravel or other bulk material and serves as a heat storage unit and is composed of four modules. The conical shape of the container can be explained by physical effects. Without this shape, the expansion during heating of the container would cause the filling to settle and then lead to stresses during the cooling phase [Mic18]. In addition to the central bulk material container, there exist also other essential components that were not included in the figure [Mic18]:

- a connection to technical processes that emit the heat used to heat the storage mass or, in the case of a test rig, the air heating coil that heats the ambient air sucked in to the desired temperature, and a fan to generate an air mass flow
- isolation to reduce heat loss to a minimum
- bulk material, which serves as a storage mass for thermal energy
- pipelines, valves and flaps to control the air flow and the operating mode

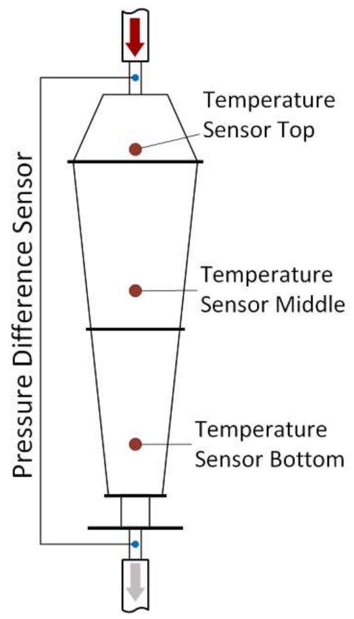

Temperature Sensor Top
Temperature Sensor Middle
Temperature Sensor Bottom
Pressure Difference Sensor

Figure 3.1: Simplified Schematic Sketch of a Packed-Bed Regenerator based on Michalka [Mic18]

The air for charging and discharging is passed through the central bulk material container via pipelines. Basically, two operating modes can be distinguished [Mic18]:

- **Charging**: The hot air, which originates as residual heat from technical processes, is directed into the container from above, transferring heat to the bulk material in the container. The loading times depend on the air mass flow as well as on the defined air outlet temperature. Alternatively, the container can also be loaded with hot air from the bottom.
- **Discharging**: The sucked-in, cold ambient air is directed into the container from the bottom. The thermal energy stored in the bulk material is then continuously released to the air flowing through.

3.5.1 Sensors and Measurement Technology

In the Packed-Bed Regenerator test rig described by Michalka [Mic18], primarily temperature sensors, pressure sensors and a mass flow meter are installed. The temperature and pressure sensors are also depicted in Figure 3.1. These sensors can play an essential role in fault detection using ML models.

Temperature Sensors To enable an accurate measurement of the temperature distribution within the heat storage unit, a total of 18 temperature sensors are installed in the test rig, one at the inlet, one at the outlet, and four in each of the four modules placed on a plane with equal distances between them and are aligned towards the center of the container [Mic18]. These sensors are represented as red points in Figure 3.1, simplified for only three modules and only one temperature sensor each.

Pressure Sensors Multiple types of pressure sensors are used in the Packed-Bed Regenerator test rig. Downstream of the fan is, for example, a relative pressure sensor or a differential pressure measurement is performed as a redundant measurement to determine the air mass flow. In order to determine the pressure loss within the filling, which is caused by the internal friction of the air as well as by friction between air particles and the solid material, a differential pressure measurement is carried out. For this purpose, sensors are installed both at the top and bottom to determine this pressure difference [Mic18]. The sensors for this differential pressure measurement are represented in Figure 3.1 by the two blue dots.

3.5.2 (ML) Models for the Packed-Bed Regenerator

Halmschlager et al. [HKBH19] investigated the application of a recurrent neural network approach for modeling the complex non-linear behavior of a Packed-Bed Regenerator in order to perform time series predictions. The objective of the model was to predict the outlet temperature of the Packed-Bed Regenerator for a given number of time steps. The basis for this data-driven approach was simulated training data that took into account bottom and top air temperatures, mass flow, temperatures at the four levels in the modules, and fill level, i.e., the total amount of thermal energy stored [HKBH19].

In another work, Halmschlager et al. [HMH21] investigated a mechanistic grey-box model for the Packed-Bed Regenerator, which can be considered a hybrid model combining a physical and a data-driven model. The model is based on physical information about the Packed-Bed Regenerator, which is used to define physical relationships described as equations, and enriched with experimental data to optimize specific parameters. The objective of this approach is the same as with the previous model: to predict the outlet temperature. In total, the researchers defined three mechanistic grey-box models, each based on different assumptions and thus having different equations. The results of the mechanistic grey-box model, specifically one of its advanced versions, exhibit high accuracy as well as robustness and reliability, thus surpassing the model based on a neural network approach as well as a physical model [HMH21].

These research papers demonstrate the relevance of using ML models and various other forms of modeling for a Packed-Bed Regenerator in scientific contexts. By utilizing, for example, temperature sensor data and other relevant information from the Packed-Bed Regenerator, it is possible to accurately predict certain output variables.

CHAPTER 4

Related Work

The following chapter addresses existing work on deployment strategies for ML models and dynamic update methods for software components. To this end, the methodological approach for conducting a SLR is defined in Section 2.1. These SLRs are performed for both topics and the results are summarized accordingly. Finally, integration approaches for ML models into an AAS are introduced and compared. The following sections address the research questions defined in Section 2.1.1 and add further, more specific criteria regarding the SLR process.

4.1 Deployment Strategies for Machine Learning Models

In addition to the general inclusion and exclusion criteria, the following inclusion criteria are added:

- The paper must be published between 2012 and 2022.
- The paper should describe how ML models can be deployed, either based on a specific use case or at a more general level.

Therefore, papers that focus solely on the implementation of a particular ML model for a given problem and its superiority over other solutions are excluded from the SLR process.

In order to get a broad overview on the topic of ML model deployment strategies in a first step, an overview search for scientific papers on Semantic Scholar[1] and Google Scholar[2] was conducted. The search terms used in Semantic Scholar and Google Scholar were:

[1]https://www.semanticscholar.org/
[2]https://scholar.google.com/

```
(deployment of) OR (deployment strategies for) machine learning
models OR machine learning model deployment in production
```

However, too many papers were obtained with this search term, making it impossible to process them systematically. Therefore, in order to refine and extend the search term, the papers on the first 10 result pages were reviewed and evaluated in terms of relevance to the research questions. Both literature search engines ordered their results according to relevance. In total, 21 papers could be identified that potentially can contribute to answering the research question based on their title and abstract. To get also more practice related literature on this topic, blog posts on deploying ML models were searched and evaluated.

In addition to the papers, some blog posts obtained through a Google search on the same keywords as used for the papers were analyzed as well. In this context, specific terms for deployment strategies were used repeatedly, which are not exclusively used for the deployment of ML models but are also very common there. Among others, the following were mentioned [Bar22], [Cen20], [Kha21], [Mal21]: *Recreate deployment*, *Canary deployment*, *Blue-Green deployment*, *Shadow deployment*, *A/B testing*, or *Rolling deployment/ramped deployment*.

Using a recreate deployment strategy, the system is first stopped, the new version is then deployed, and then the system is finally restarted. This approach is very easy to perform, but results in downtime and has a negative impact on users experience. In a canary deployment scenario, a new version of a model is deployed, and initially only a small portion of the user traffic is redirected to the new version. This fraction is then increased gradually. In this way, the new model is tested against the current operational model. With this approach, deployments without downtime are possible. Furthermore, there is the possibility for a rollback if necessary [Bar22]. In a blue-green deployment scenario, also known as red-black deployment, a new version of a model is deployed in a separate environment while the current model is operational in another environment. After the new model is tested, it can be put into service, and all traffic is routed to the new model. The other environment can be used for possible rollbacks or as a staging area for the next version of the model [Cen20].

In a shadow deployment scenario, the new model version is deployed alongside the current model and receives the same requests as the live model, but the results are not published to the users as they are only used for evaluation purposes [Bar22]. A/B testing is a strategy that compares at least two versions of a model to determine which of them performs better. For this purpose, the user group is divided into subgroups and each of them is assigned a model version [Kha21]. Rolling deployments can be used if there are multiple instances of an application. During an upgrade step, only some of the instances are upgraded to the new version while the others are still running the previous version. This procedure is then repeated until all the instances are upgraded [Cen20].

Based on these findings, the search string was then modified to obtain a manageable set of results. To this end, the search term was iteratively refined and tested on IEEE

Xplore. The reasons for using IEEE Xplore were that the initial overview search already identified many articles on this topic published by IEEE and also the advanced search interface from IEEE Xplore is very intuitive to use. Table 4.1 summarizes the iteration steps and the number of results obtained. The displayed search string in the table has been slightly adjusted for better readability compared to the syntactically correct one used for the search on IEEE Xplore. The *metadata* data field used in the first iteration includes among others the abstract, index terms, and document and publication title.

One insight that was recognized right at the beginning was that search terms that contain ML in the text or in the abstract and were not combined with other, more restrictive terms led to too many results, almost all of which were not relevant. Based on the findings from iteration 1, the data field for the search terms was changed to just the title of the document to limit the number of results. For step 2, the search term was extended to include at least one of the names of the different deployment strategies in order to find these strategies also in the scientific literature. However, this extension led to only one result. Therefore, the search terms for the title were expanded to include terms that were regularly found in the overview search and also aimed at answering the research questions. Step 4 of the definition of the search term thus leads to 14 results. For the fifth iteration, the search term was slightly tweaked. In the overview search, some frameworks ending in *flow* were mentioned in the papers. Therefore, this part was added to the search term for the title. To exclude papers from the hit list that merely indicate the use of ML for solving their research problem in their work, the title filter string was extended by explicitly excluding the phrase "using machine learning" from the title. Furthermore, *model scoring*, *ML platform*, and *MLOps platform* were added to the full-text search field, as these terms also appeared a few times in the initial search. Furthermore, instead of *deployment* only *deploy** was taken in the title filter, so that for example also *deploying* is found.

The final search string that was also used to query the other three publication databases is therefore as follows:

```
Title:("deploy*" OR "deployment strategy" OR "rollout" OR release
OR "lifecycle" OR "continuous integration" OR "CI" OR "continuous
delivery" OR "CD" OR "architecture*" OR "*flow" NOT "using machine
learning") AND ("AI" OR "ML" OR "machine learning" OR "artificial
intelligence")
AND
Text:  "canary" OR "blue-green" OR "red-black" OR "shadow deployment"
OR "rolling deployment" OR "ramped deployment" OR "recreate deployment"
OR "model scoring" OR "machine learning platform?"  OR "ML platform?"
OR "MLOps platform?"
```

Remarks on the Search Results of the Publication Databases Scopus does not support a full text search functionality. Therefore, the search term for the full text part of the query has been added to the combined search data field for title, abstract and

Iteration	Search Term	# of Results
1	**Metadata:**(“deployment” OR “deployment strategy” OR “rollout” OR “release”) AND (“AI” OR “ML” OR “machine learning” OR “artificial intelligence”) AND “model”	4623
2	**Title:**(“deployment” OR “deployment strategy” OR “rollout” OR release) AND (“AI” OR “ML” OR “machine learning” OR “artificial intelligence”) AND “model”	12
3	**Title:**(“deployment” OR “deployment strategy” OR “rollout” OR release) AND (“AI” OR “ML” OR “machine learning” OR “artificial intelligence”) AND “model” AND **Text:** “canary” OR “blue-green” OR “red-black” OR “shadow deployment” OR “rolling deployment” OR “ramped deployment” OR “recreate deployment”	1
4	**Title:**(“deployment” OR “deployment strategy” OR “rollout” OR release OR “lifecycle” OR “continuous integration” OR “CI” OR “continuous delivery” OR “CD” OR “architecture*”) AND (“AI” OR “ML” OR “machine learning” OR “artificial intelligence”) AND **Text:** “canary” OR “blue-green” OR “red-black” OR “shadow deployment” OR “rolling deployment” OR “ramped deployment” OR “recreate deployment”	14
5	**Title:**(“deploy*” OR “deployment strategy” OR “rollout” OR release OR “lifecycle” OR “continuous integration” OR “CI” OR “continuous delivery” OR “CD” OR “architecture*” OR “*flow” NOT “using machine learning”) AND (“AI” OR “ML” OR “machine learning” OR “artificial intelligence”) AND **Text:** “canary” OR “blue-green” OR “red-black” OR “shadow deployment” OR “rolling deployment” OR “ramped deployment” OR “recreate deployment” OR “model scoring” OR “machine learning platform?” OR “ML platform?” OR “MLOps platform?”	16

Table 4.1: Evolution of Search Terms and Number of Results Obtained for Deployment Strategies for Machine Learning Models, performed on 25.06.2022 on IEEE Xplore

Source	**# of Results**	**# of Selected**	**# of Relevant**
IEEE Xplore	16	12	11
ACM Digital Library	4	2	1
Scopus	2	0	0
Total	22	14	12

Table 4.2: Final Results Based on the Defined Search Term for Deployment Strategies for Machine Learning Models, performed on 25.06.2022

keywords. The search and advanced search functions at Springer Link also differ from those at IEEE Xplore and ACM Digital Library. Springer Link does not offer a combined search for title and full text or abstract data fields. Therefore, one approach was to combine the search terms for title and full text with AND to create a search term that was used only for a full text search. However, this search produced 9625 results, which was impossible to handle in the course of this work. Also, the title search alone could not be used with the search terms previously defined and used with the other publication databases. Therefore, the Springer Link publication database is not considered for this part of the SLR.

Table 4.2 summarizes the final result in terms of relevant papers found in a quantitative way. In total, three publication databases were considered and a total of 22 papers was found that matched the defined search term and was further evaluated. The number of selected papers indicates the number of papers for which both the title and abstract seem promising to contribute to answer the defined research questions. On IEEE Xplore, 16 entries were found, of which 12 had a promising title and abstract, and 11 were finally classified as relevant. In the ACM Digital Library, four entries were found, of which two were read thoroughly, and one is relevant according to the defined criteria. Both papers found on Scopus are not relevant for answering the defined research questions at all.

4.1.1 Summarizing the Findings

The following section summarizes the key statements of the relevant papers after a thorough reading of those selected papers. Thus, all 12 summarized papers contribute to answering the defined research questions and are therefore considered as relevant to the research.

Maskey et al. [MMH$^+$19] mention a difference between developing a model and deploying it productively to make predictions. They examine this topic from the perspective of the Earth science domain and propose a inherently iterative ML lifecycle focused on their community. An essential aspect of deploying ML models is testing and ensuring the performance of the model in production as different data is processed in production environments. A/B testing is therefore a method to compare the performance of different deployed models. The authors mention as a challenge in deploying ML models the regular iterations of models and keeping track of their changes. They suggested the adoption of mechanisms similar to those used in software development and operations (DevOps) [MMH+19]. This challenge was later addressed by other researchers.

Hummer et al. [HMR+19] propose with ModelOps a cloud-based framework and platform for Artificial Intelligence (AI) application lifecycle management, which aims to extend the principles of traditional DevOps pipelines and combine these with AI model lifecycles, though they differ in some aspects. Nevertheless, deploying models also poses some risks. Techniques such as canary releases, A/B testing, user feedback, and drift detection can be used to address these risks. ModelOps provides the ability to seamlessly shift between execution environments, as the generic, platform-independent pipeline generates execution environment-specific artifacts using additional information. In addition, the authors mention the importance of closed feedback loops to retrain models based on specific events [HMR+19].

Garg et al. [GPR+21] investigated approaches on applying concepts from DevOps to the world of machine learning model deployment (MLOps) using existing advances such as containerization and model management tools that support the model lifecycle. Containerization allows to decouple the execution environment and to ensure reproducibility. Together with an orchestration framework, containers can be easily deployed, managed, and automated. Thus, Docker and Kubernetes are often used in practice for such use cases. Furthermore, the authors differentiate between three levels of MLOps depending on their level of automation - ranging from manual and script-driven processes to fully automated Continuous Integration and Continuous Delivery (CI/CD) pipelines. In recent years, a number of MLOps platforms have emerged that include a set of tools and frameworks that facilitate the process of automated development of machine learning models. However, they are often not yet mature and are still under development [GPR+21].

Gisselaire et al. [GCGb+19] defined an approach to asses deployment architectures for intelligent Cyber-Physical Systems (iCPSs) based on cost and security aspects. iCPSs are Cyber-Physical Systems (CPSs) that embed also ML technologies. For their approach, they reduced each iCPS to four components: (1) Training Data-Sets, (2) Model Learning, (3) Predictive Model, and (4) Decision Taking. Each of these components can be deployed either on the edge or in the cloud, resulting in a total of 16 deployment configurations. For example, developing and deploying ML models in the cloud is much more cost effective than on the edge because the cloud infrastructure already provides much more functionality and resources are provided on-demand. On the other hand, some security risks are of greater concern when data is sent from the edge to the cloud [GCGb+19].

In their paper, Peticolas et al. [PKT19] describe a cloud and edge spanning ML and data science framework, named Mímir, for the Industrial Internet of Things (IIoT) and how to deploy ML applications in this industrial setting. In this context, time series data is continuously streamed to the cloud and stored there to perform analysis and train and validate ML models based on this data. To deploy the model, they create ML model containers that contain all the necessary packages and dependencies so that they can eventually deploy the model to the cloud or alternatively to the edge. This approach allows easy scaling for additional edge nodes, as well as easy replacement of model containers with new model versions [PKT19]. However, the authors do not explicitly mention a specific strategy for updating the model containers.

Wöstmann et al. [WST$^+$20] propose a reference architecture for ML in the process industry. The proposed architecture follows the hierarchical structure that is defined in industry standards and, furthermore, deals with the integration of software components at different levels of the automation pyramid. The edge devices, an optional component of the architecture that enables decentralized execution of scoring processes, act as kind of buffer in case of network issues to prevent data loss, but they can also provide real-time streaming data or send them as “batch”-format. Furthermore, model scoring directly on the edge enables low-cost and near-real-time model execution. In addition, model update capabilities and storage strategies are crucial for stable productive use. The authors also suggest the use of ML platforms for model management and deployment, especially when larger amounts of data are processed and more complex models and pipelines are required. For performance and stability reasons as well as to ensure short response times and resource efficiency, the proposed architecture utilizes the concept of a separate deployment environment. Thus, the proposed architecture provides the flexibility to deploy models on-premise, in the cloud, or also on fog nodes. Moreover, the paper mentions the issues of concept drift and its implications and therefore the need for permanent monitoring and evaluation of model performances [WST$^+$20].

Giannopoulos et al. [GSK$^+$22] propose a general-purpose workflow for developing, deploying, evaluating, and retraining ML models in the area of Open Radio Access Networks (O-RANs). A O-RAN is an approach for future wireless networks that incorporate transparent, open, and programmable communications. Within such O-RANs there are different application scenarios for ML possible that have different timing constraints (from real-time to near-real-time to non-real-time applications). The described workflow focuses on a near-real-time controller as the main intelligence actor. After the model is constructed and trained, it is containerized using standardized tools and subsequently stored in a model catalog. The trained model is passed to the near-real-time controller, which performs near-real-time inference. If the need for a model update is discovered during the model evaluation, the model can be either retrained or there is already a better performing model in the model catalog. In this case, the better model needs to be deployed to the controller again [GSK$^+$22]. Lee et al. [LJSY21] also propose a workflow architecture for ML models in an O-RAN and their integration into a RAN Intelligent Controller (RIC) platform. For training and deploying the created model, they use ML pipeline automation techniques to satisfy MLOps Level 1. For the implementation, they used the MLOps platform Kubeflow because it supports both end-to-end lifecycle management of the model and automation of the training pipelines. The trained and packaged models are then deployed on inference services to process requests. After a required retraining of the model, a new version thereof is created, and the inference host can then update the model with the latest version. The deployment strategy used to deliver those model updates is a canary rollout (a small percentage of the requests is routed to the new model version, while the remainder is still routed to the old one) [LJSY21].

Muthusamy et al. [MSI18] propose a data-driven approach to utilize AI to analyze the performance of AI methods in business processes. They suggest to deploy models as microservices that have three well-defined interfaces for scoring, retrieving feedback, and notification of new model versions. The functionalities can be accessed via an API. For the deployment of new model versions strategies such as canary deployments and A/B testing can be used. The use of a new model version always involves a certain risk, even if the model has been sufficiently tested, because the overall impact on the business application is difficult to predict. One way to mitigate the risk is to use a model proxy that forwards the calls to the respective model microservices and logs both the inputs and outputs. Based on these logs, analyses can be performed, whose results in turn can affect routing. Also, fallback models can then be used if the performance of a model is not as expected or does not meet certain thresholds [MSI18].

A similar approach is taken by Argesanu and Andreescu [AA21] for the design of a end-to-end ML platform that is mainly focused on batch inference. The architectural design consists of three main layers: (1) Orchestration Layer, (2) Model Serving, and (3) Data Access. While the Model Serving tier hosts all the models, the Orchestration tier handles among other things the prediction requests. The orchestration layer also records prediction requests, errors, and results to provide statistics and insights for various stakeholders, but also to monitor model performance over time. Additional emphasis is also placed on containerization to ensure that the characteristics of the production environment are the same as for model training. Also, model versioning ensures that all trained models are ready for deployment. These two aspects facilitate model deployment and help mitigate the risks associated with deploying ML applications [AA21].

Warnett and Zdun [WZ22] defined a decision model for architectural designs based on a qualitative study of the technical challenges that practitioners face when introducing ML models into production. One decision point for deploying such a model is the level of automation to be selected, ranging from no automated deployment to semi-automation by deploying pre-prepared pipelines to full CI/CD pipeline automation. In addition, it is important to define the events that trigger these pipelines. Among the tasks that can be automated in building and deploying pipelines are containerization and testing using canary deployment or A/B testing. Furthermore, there are different approaches regarding which and how many model versions should be used in production. This decision is in turn influenced by various factors such as the need for safe model transitions or rolling upgrades. Both require limited downtime, which is only possible if there is more than one model version available. Therefore, a strategy with N versions in production is recommended, as this meets both of the above requirements and enables smooth model version exchanges without downtime. One question that came up repeatedly and that would help with decision making if answered right at the beginning is whether or not to use MLOps [WZ22].

Openja et al. [OMK$^+$22] conducted an experimental study by analyzing 406 open-source

Deployment Concept	[AA21]	[GPR$^+$21]	[GSK$^+$22]	[GCGb$^+$19]	[HMR$^+$19]	[LJSY21]	[MMH$^+$19]	[MSI18]	[OMK$^+$22]	[PKT19]	[WZ22]	[WST$^+$20]
MLOps platf. - CI/CD		X	X			X	X		X		X	X
Containerization	X	X	X			X			X	X	X	X
A/B Testing					X		X	X			X	
Canary Deployment					X	X		X			X	
Rolling Deployment											X	
Web Service - API				X				X	X			
New Framework	X				X					X		

Table 4.3: Mapping of Papers to Deployment Concepts

ML software projects to understand how Docker is used in the field of ML deployment processes. They found that most use cases for using Docker as deployment tool are application systems that contain the ML model along with the rest of the code to perform end-user tasks, followed by MLOps and ML toolkits. The intended uses of Docker are many, including model and data management, cloud-based deployment, interactive deployment, CI/CD, or build and packaging. However, Docker is in general used for deploying ML models mainly for portability reasons [OMK$^+$22].

Table 4.3 provides a list of the main approaches and strategies for deploying ML models and the corresponding papers in which they are stated. Most of the analyzed papers focus their deployment strategies and concepts on MLOps, mostly in combination with containerization techniques. Moreover, most of these papers provide only a brief description of ML model deployment and focus rather on the entire development and ML model lifecycle. Other authors, however, proposed their own framework for the development and deployment of ML models, as they encountered some limitations within the existing frameworks. Some papers also discuss possible testing and upgrade strategies for such models, especially if multiple model versions are in service.

Figure 4.1 summarizes the results of this part of the SLR in a quantitative manner. The vertical axis of this bar chart shows the deployment concepts mentioned in the papers, while the horizontal axis shows the number of papers that mention the respective concept in their paper. It is important to note that multiple concepts may be mentioned in a paper, so the total number of these occurrences does not add up to the 12 relevant papers identified during the SLR.

MLOps platforms, one of the concepts for deploying ML models listed in Table 4.3, are platforms such as Kubeflow, Amazon SageMaker, and MLflow [GPR$^+$21]. The concept of containerization comprises tools such as Docker and the orchestration tool Kubernetes.

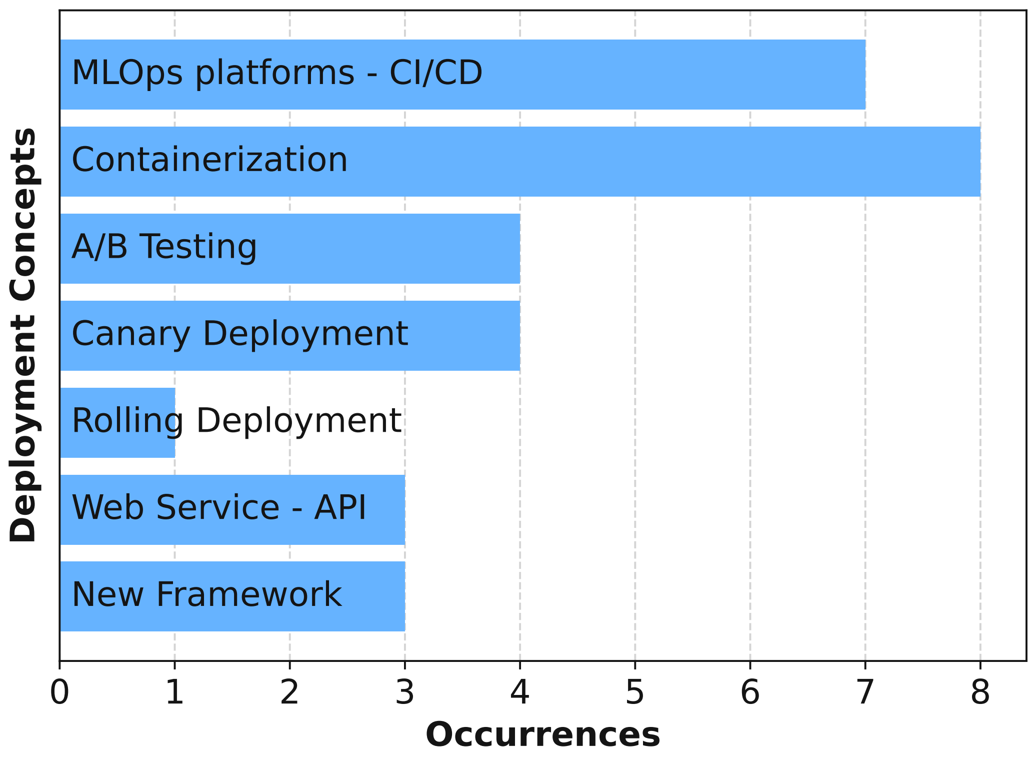


Figure 4.1: Occurrences of Deployment Concepts in Papers

Complementary Paper to the SLR Results One paper that was found in the initial overview search but did not match the search terms in the SLR was published by Benton [Ben20]. Because of the title *"Machine Learning Systems and Intelligent Applications"*, the search criteria did not fit, as adding *Systems* or *Applications* to the search string would have yielded far too many results. Nevertheless, this paper contains some interesting aspects that can help to answer the research questions.

Benton [Ben20] relates the technical challenges of complex application developers to the field of machine learning systems and introduces the concept of intelligent applications. One important challenge for ML is the reproducibility of pipelines. But the way ML applications fail is also different compared to classical software, because ML models can fail but still provide predictions even though they are wrong. Thus, intelligent applications are different to classical applications as ML models provide an essential functionality and therefore the development process differs. Also, their way of deployment is different. To simplify the development and deployment of intelligent applications, he suggests the use of a microservice architecture. Thus, Kubernetes and containerization are proposed as the basis for this architecture. This approach allows declarative deployments and enables continuous integration, continuous deployment, and even a more sophisticated blue-green deployment strategy. In this context, a blue-green deployment is described as an approach where one part of the requests is forwarded to the new model while other requests are still forwarded to the old model. The proportion of requests for the old model gradually changes more and more towards the new model. This also enables deployments without downtime [Ben20]. In the rest of the literature, however, a blue-green deployment is understood as a complete switch from the old to the new model version from one moment to the next. The approach described in this paper is more commonly understood in the literature as canary deployment.

4.1.2 Answer to Research Question RQ1a

As it can be seen in Table 4.2, 22 papers were initially found in accordance with the search terms. After reviewing the title and abstract, 14 papers were selected for thorough reading. Finally, 12 papers were considered as relevant to answer the research question *RQ1a*.

RQ1a: Which deployment strategies for ML models have been published by other researchers or are suggested by practitioners and ML experts? The assignment of deployment concepts in Table 4.3 already helps to answer this question. Most of the work analyzed focuses on ML model deployment on MLOps platforms and ML frameworks, both commercial and open-source. This includes also the automation of building, packaging, and testing pipelines up to full automation using CI/CD pipelines. Likewise, containerization techniques, such as the use of Docker and Kubernetes, are common in the papers and are used, for example, for packaging the models or for easy deployment to different locations, such as edge nodes or the cloud. These tools are also used for quick and easy model upgrades. Especially for cases where multiple models are in operation or to test the performance of different models against production data, different deployment and testing strategies are mentioned. Canary deployment and A/B testing are most frequently mentioned, followed by rolling deployment strategies. These strategies can also be used for model upgrades, especially with a focus on zero downtime and safe rollouts as well as the ability for an easy rollback to the old version if the performance of a model is not as good as expected. In contrast, accessing the model for model scoring via an API was mentioned rather seldom. However, accessing the model via an API can be done either directly via a web service or also via an API in combination with a containerized model.

In some papers, expert interviews were conducted with domain experts, while in others authors from companies in the field of ML or the respective companies where the use cases or implementations were tested contributed to these publications. Based on their analyses of codebases and repositories of real-world projects, the work of Openja et al. [OMK+22] also showed that Docker plays an important role in practice at various stages of model development and deployment, as well as for model integration into applications.

4.2 Dynamic Updates of Software Components

In order to link the findings on deployment strategies and upgrading approaches for ML models to the topic of dynamic update methods for software components in production environments in general, a SLR based on the same methodological approach as defined in Section 2.1 is also performed for the latter topic. In addition, relevant aspects with regard to these methods in relation to production environments should be identified as well.

Additional to the general inclusion and exclusion criteria in Section 2.1.2, the following inclusion criteria are added:

- The paper must be published between 2007 and 2022.
- The paper should describe how software components like Programmable Logic Controllers (PLCs) can be dynamically updated in production environments such as real-time systems or CPSs. In addition, these dynamic update strategies also include approaches for disruption-free software updates and update strategies for time-critical applications. Moreover, software frameworks used for this purpose are also of interest.

The time period to be considered for the publications was extended from the last 10 years to the last 15 years compared to the SLR on deployment methods for ML models, since quite a few articles to be noted on this topic were also published around 2010.

As for the first part of the SLR, an overview search on this topic was initially conducted on Semantic Scholar and Google Scholar. The search terms used in these search engines were:

```
dynamic software updating real time systems OR disruption-free
software updates in production environments
```

As one might have already expected, too many papers were found to be able to evaluate all of them systematically. In this context, both literature search engines ordered their results by relevance, taking into account results up to the 10th page for further analysis. Based on these search terms, 16 papers were identified that provided a rough overview of the topic and on the basis of these findings the search term could be improved. As before, IEEE Xplore advanced search was used to test and refine the search terms.

Table 4.4 summarizes the iteration steps and the number of results obtained for this part of the SLR. Again, the displayed search string in the table has been slightly adjusted for better readability.

Although the metadata data field was used in the first iteration, including among other things the abstract, index terms, and the title of the document, only six results were found using these search terms. Therefore, additional search terms were added in the second iteration. One of the main goals of dynamic software updates is to not disrupt

Iteration	Search Term	# of Results
1	**Metadata:**(“dynamic software updat*” OR “DSU” OR “dynamic update method?”) AND (“real time system?” OR “automation system?”	6
2	**Metadata:**(“dynamic software updat*” OR “DSU” OR “dynamic update method?” OR “disruption-free” OR “non-disruptive” OR “zero-downtime” OR “no-downtime”) AND (“real time system?” OR “automation system?” OR “container environment?” OR “Virtual PLC?”	21
3	**Full Text:**(“dynamic software updat*” OR “DSU” OR “dynamic update method?” OR “disruption-free” OR “non-disruptive” OR “zero-downtime” OR “no-downtime”) AND (“real time system?” OR “automation system?” OR “container environment?” OR “Virtual PLC?”	124
4	**Title:**(“dynamic updat*” OR “dynamic software updat*” OR “DSU” OR “dynamic update method?” OR “disruption-free” OR “non-disruptive”) AND (“real time system?” OR “automation system?” OR “container environment?” OR “Virtual PLC?” OR “non-disruptive” OR “zero-downtime”	4
5	**Title:**(“dynamic updat*” OR “dynamic software updat*” OR “DSU” OR “dynamic update method?” OR “release” OR “Docker” OR “Kubernetes” OR “container-based” OR “containerization”) AND (“real time system?” OR “automation system?” OR “time-sensitive application?” OR “container environment?” OR “Virtual PLC?” OR “high availability” OR “disruption-free” OR “non-disruptive” OR “zero-downtime” OR “no-downtime”)	11

Table 4.4: Evolution of Search Terms and Number of Results Obtained for Dynamic Updates of Software Components, performed on 13.07.2022 on IEEE Xplore

running operational programs. Therefore, terms such as *disruption-free* or *zero-downtime* were added, as well as other system-related keywords that appeared in some of the articles found in the overview search. For the third iteration, a full-text search was conducted and a total of 124 papers were found, but most of them did not indicate relevance to the research questions solely based on their titles. Accordingly, for the fourth iteration, the data field for the search was changed to search only within the title, as this seemed to be a good indicator for relevance. In addition, some terms were rearranged within the search clause to provide more appropriate combinations of search terms. For the last iteration, the search string was extended again and some terms were rearranged to find more relevant papers. Thus, terms like *Docker*, *Kubernetes* or *time-sensitive application* were added, as these terms also appeared sometimes in the papers found in the overview search. In contrast to the first SLR for deployment methods, searching the full-text fields often resulted in a large number of non-relevant papers, so for this SLR a full-text search with specific terms was omitted.

Therefore, the final search string used to query the publication databases is as follows:

```
Title:("dynamic updat*" OR "dynamic software updat*" OR "DSU"
OR "dynamic update method?"  OR "release" OR "Docker" OR
"Kubernetes" OR "container-based" OR "containerization") AND
("real time system?"  OR "automation system?"  OR "time-sensitive
application?"  OR "container environment?"  OR "Virtual PLC?" OR
"high availability" OR "disruption-free" OR "non-disruptive" OR
"zero-downtime" OR "no-downtime")
```

Remarks on the Search Results of the Publication Databases The search functionality on Springer Link is somewhat different from that of the other publication databases, as in the former the title-only search with the defined search term had not worked properly and thus no result rows were obtained. Therefore, the search term was used for the full-text search, as this returned results. However, this search resulted in 1601 hits. Since some of the search terms often led to articles that were not relevant, the final search string for the search on Springer Link was reduced to some extent. In addition, entire books were excluded from the results and only the specific articles where the corresponding search terms occurred were considered. The search string used for Springer Link is therefore:

```
Text:("dynamic updat*" OR "dynamic software updat*" OR "dynamic
update method?")  AND ("real time system?"  OR "automation
system?"  OR "zero downtime" OR "no downtime" OR "container
environment?"  OR "Virtual PLC?") NOT "signal processing"
```

Thus, compared to the original search string, those search terms were omitted that mainly led to non-relevant results. In addition, the term *signal processing* was excluded, as this also led to non-relevant hits. Under these constraints, the number could be reduced to a manageable amount of 33 papers.

Table 4.5 summarizes the final result in terms of relevant papers found in a quantitative

Source	# of Results	# of Selected	# of Relevant
IEEE Xplore	11	6	6
ACM Digital Library	2	2	2
Scopus	20	3	2
Springer Link	33	7	5
Total	66	18	15

Table 4.5: Final Results Based on the Defined Search Term for Deployment Strategies for Machine Learning Models, performed on 13.07.2022

way. A total of four publication databases were considered and altogether 66 papers were found that matched the defined search terms and were further evaluated. The number of selected papers indicates the number of papers for which both the title and abstract appear promising to contribute to answering the defined research questions. Duplicates were also removed in this step. It is important to note that many duplicates were found in the Scopus database that were already included in the search results of those publication databases in which they were originally published. Therefore, papers published by IEEE count as selected directly for IEEE. The same policy applies to ACM and Springer Link. On IEEE Xplore, 11 entries were found, of which six had a promising title and abstract, and all of which were also considered relevant. In the ACM Digital Library two papers were found, both of which were selected and eventually classified as relevant. Through the Scopus search, 20 papers were originally found, some of them were duplicates or did not contribute to answering the research question based on title and abstract. Of the selected three papers, two could be finally classified as relevant. Of the 33 papers originally found on Springer Link, the analysis of title and abstract already excluded a large number of papers, since the full-text search also found many papers that contained these words but covered a completely different topic. Finally, five of the seven selected papers were classified as relevant.

4.2.1 Summarizing the Findings

In the following section, the key messages of the relevant papers are summarized after a thorough reading of the selected papers. Therefore, all 15 papers summarized contribute to answering the defined research questions and are therefore considered relevant to the field of research.

Wahler et al. [WRO09] propose an approach to dynamically update components in an embedded systems without violating real-time constraints. First, they had to determine the times at which an update is actually possible, since real-time systems always have to meet certain deadlines. Additionally, the update must be completed within a deterministic amount of time. In their approach, the updates are performed in the period after the execution of the code and the beginning of a new cycle. Within this period, the state transfer of the internal information from the old to the new component must also take place. Basically, their framework consists of components and channels that interconnect the former, and a component manager that coordinates operations and updates. During a time-critical update phase, the component manager gives the old component the signal to terminate. Then, the latter hands over the channels to the component manager and terminates itself. Finally, the new component takes over the connections from the component manager and starts operating. The state transfer between the old and new components is done via a shared memory and must be completed in the same cycle in which the update occurs, including possible data type conversions [WRO09].

Yi [Yi19] mentions that there is a trend of transitioning from currently often single-purpose embedded real-time systems to open platforms that allow the integration of additional software components to improve or customize functionality or protect against security threats. Therefore, he suggests that such systems need to be upgraded in a component-wise, incremental manner, without the need to redesign, update, or verify the entire system. Such real-time systems require a deterministic input-output and timing behavior. Although simulation tools can be used to test and verify the semantic behavior of entire systems, in order to obtain reliable and predictable systems, the extension of such systems is complicated because new components must fit exactly into the time schedules. Yi [Yi19] therefore proposes a design architecture, consisting of three layers (function layer, software layer, and hardware layer), to address the challenges of building updatable real-time systems, such as ensuring the verifiability of updates while considering runtime and resource efficiency. New or updated components must meet the defined functional and non-functional requirements [Yi19].

Seifzadeh et al. [SKKM09] propose an approach to dynamically update tasks without making assumptions about task properties such as execution times, as these may differ between the original and updated versions of a task. They defined a task that is executed periodically in each hyper-period and is responsible for checking for new task updates and replacing the outdated task with the new task if it is still schedulable, otherwise the update request is rejected. In this context, a schedulability or acceptance test ensures that a set of tasks can be executed without missing deadlines, which is an essential requirement in real-time systems. For the next job of this task in the following hyper-period, the updated version will then be executed. An essential metric for the update task in real-time environments, but one that is difficult to calculate due to the I/O operations performed, is its Worst Case Execution Time (WCET) [SKKM09].

In the literature, such an approach is referred to as Dynamic Software Updating (DSU). Such DSU techniques aim at updating or modifying computer programs during runtime without the need to shut down and restart the system, but by performing code and state transformations from the currently running program to the new program. Beyond that, redundant hardware is also not required [MAAJ19].

Ribeiro and Baunach [RB17] identified approaches for remote updates and proposed a concept for applying remote updates to dynamically composed real-time systems, focusing on schedulability analysis during dynamic updates to ensure real-time capability throughout the update process and beyond. Their proposed update protocol consists of steps that can be performed either on the server side, which transfers an executable module to the target device, or on the device itself, as well as hybrids that result in a trade-off between overhead incurred on the server or device side. An essential element for performing dynamic updates is metadata, which creates this overhead. However, this metadata is required to perform compatibility checks defined by pluggability (i.e., checking whether new modules fit into the existing system and references can be resolved) and interoperability (i.e., execution behavior), as well as for linking and relocation. Therefore, an approach in which the entire overhead is shifted to the resource-constrained target devices is rather inconceivable for real-time systems [RB17].

Mugarza et al. [MAAJ19] propose a mixed-criticality software architecture based on the Cetratus runtime framework for implementing a high-availability smart energy application. Cetratus enables DSUs for safe live updates of software components for industrial control systems. Before a new update goes into production, the modified component is tested and validated in a quarantine environment. If the auditor confirms the safety and correctness of the update, it is put into production. In the proposed architecture, the components defined as upgradable have two containers associated with them, which provide isolated execution environments in both spatial and temporal domains. These containers can be used to dynamically update one container without affecting the other container in order to perform a safe upgrade strategy with quarantine-mode and final switch to the new version [MAAJ19].

Pina et al. [PAHC19] propose an approach to perform reliable low-latency updates for stateful services that combines DSU with multi-version execution (MVE) to solve the problem that DSU techniques alone cannot guarantee continuous availability, since the updated code itself may contain new bugs or state transformations may be incorrect. So, they called it MVEDSUA. A MVE system runs multiple instances of a program and gives each instance the same input and then compares the output to check if they are the same. In their approach, the current program can be forked into the leader and a follower, while both receive the same input and thus can maintain the same state. The follower is then updated and the results of the two are compared for equality. If there are discrepancies between the outputs, then this indicates an error in the new version or during the updating process, and the update can subsequently be reverted. Otherwise, the updated program (the follower) is promoted to the leader and the leader is then the follower as standby backup that can be terminated at the end if the update is considered successful overall. To deal with expected differences, programmers can define such deviations using domain-specific languages provided by MVE systems. In MVEDSUA this even works with new commands in the new program version [PAHC19].

Sollfrank et al. [SLVH19] investigated the impact of time delays on time-sensitive applications due to application virtualization using Docker containers in embedded systems. Application virtualization requires additional software components and incurs additional processing overhead. However, in CPSs and Cyber-Physical Production Systems (CPPSs) it enables platform-independent development and deployment of applications and supports security and scalability. For their research, they studied the impact on time-sensitive applications in distributed networked control systems. To do this, they specified three criteria for evaluation: (1) round trip times, (2) processing time delays, and (3) suitability for networked control systems. Their experiments showed that applications in Docker containers, given the highest real-time priority, can be used for applications with real-time requirements. The additional time delay when using Docker for round-trip time was 43 μs, and the added processing delay was not significantly higher with 5 μs [SLVH19].

In a subsequent work, Sollfrank et al. [SLDVH21] extended their previous research and used similar evaluation criteria: (1) round trip times, (2) CPU time delays, and (3) time distributions and outlier detection to meet safety requirements. They conducted a similar experiment, but with the new approach of observing 10,000 request/response patterns, they wanted to gain more detailed insight into the impact on performance and prove the suitability of using containerized applications for real-time tasks. Their results showed again that Docker-based applications with the highest real-time priority are well suited for soft real-time requirements in industrial automation [SLDVH21].

Abdollahi et al. [VSTK19] evaluated the possibility of using Kubernetes to orchestrate containerized stateful microservices and subsequently proposed a solution to improve the availability of these microservices by enriching Kubernetes with an additional state controller. The state controller initially assigns active and standby labels to pods after they are deployed. If a pod providing a service fails, this is detected by the state controller, which can then redirect to a standby pod that can resume providing the service. Due to state replication, the standby pod is always aware of the state of the active pod, as data from the active pod is also sent to the standby pod via a state replication service. Their experiments have shown that Kubernetes can be used in combination with an additional state controller to deploy stateful microservices in a resilient manner by reducing the recovery time of such microservices [VSTK19].

Silva et al. [SGB16] performed a theoretical study and evaluated strategies for the deployment of updated services to mitigate or even eliminate service unavailability as this leads to lower customer satisfaction or financial issues. They address blue-green deployments and canary releases, both of which basically have two separate environments, one of them is upgraded, and differ primarily in the strategy for switching requests between them. In addition, these two strategies can be combined with virtualization techniques or lightweight virtualization techniques such as containerization. In their study, they compared blue-green deployments and canary releases strategies, each combined with virtualization and lightweight virtualization, in terms of their advantages and disadvantages regarding costs, complexity, performance, effort, and impact on users. Their results showed that none of these combinations is clearly better than the others, but in their implementation they used a blue-green deployment strategy in combination with virtualization because lightweight virtualization was associated with high complexity since this technology was newer at that time [SGB16].

A blue-green deployment strategy for industrial control application updates at runtime using microservices is proposed by Koziolek et al. [KBA$^+$21]. They developed a dynamic update mechanism for virtual PLCs that cyclically execute control algorithms in production processes and are deployed as Kubernetes microservices. A recent trend is to shift from dedicated embedded controllers to more flexible container environments in order to reduce costs and simplify application management. A crucial point in this context is the updating of such stateful virtual PLCs as the updated version must have the same internal state information as the outdated one. Their approach allowed them to complete a full update of a PLC within the cycle slack time (i.e., cycle time minus actual execution time), including the transfer of up to 100,000 state variables, in less than 15 ms, so that it could be performed between two cycle executions and without interrupting production processes. Basically, they started a second container with the updated PLC logic, then transferred the state variables and finally switched to the output of the updated controller after validating both PLC behaviors [KBA$^+$21].

Ahmadighohandizi and Systä [AS18] propose a platform for the development and deployment of Internet of Things (IoT) applications based on DevOps principles. They also discussed different deployment strategies for updated versions of applications in the context of IoT devices. In their proof-of-concept, they implemented a simple blue-green deployment strategy for their applications. However, they also mention that for mass deployment to a larger number of IoT devices in production, a canary deployment strategy is the better choice, as the new version can be tested on a small set of devices first and in case of defects, the other devices are not affected [AS18].

Boyer et al. [BEdPT18] propose an architecture-based approach for automated microservice updates in the domain of Platform-as-a-Service sites. For this purpose, update strategies are defined as sequences of elementary changes (transitions) on an architectural model of a microservice application, where the model describes in principle how microservices are configured and deployed. Applying updates to a model first has some advantages, such as ease of use, previewing changes, or better control of the update process. Therefore, microservices are updated by simply specifying the desired target architecture and the respective update strategy as transitions. Possible upgrade strategies include a blue-green deployment strategy that aims for zero-downtime and uses an intermediate architecture to evaluate the new version before putting it into production. Other strategies implemented and evaluated by the authors included canary deployment and variations thereof, clean redeploy, or straight (without any intermediate architecture). Since update strategies are defined as transitions, the current and target architectures are always compared and updates are thus idempotent and can be restarted in case of errors during a previous run. In addition, rollbacks are also possible, i.e. it is only necessary to set the original architecture as the target architecture and perform the update process again [BEdPT18].

Képes et al. [KLZ20] propose an approach for modeling and executing software updates in a situation-aware manner in order to perform the updates at the right time. This means that the context of the system in terms of time, application state, environment or people can be observed and the WCET of an update is also taken into account. The first step of the proposed method deals with modeling of an update, such as the components and relationships required to achieve the desired target deployment model. In the second

Dynamic Update Method	[VSTK19]	[AS18]	[BEdPT18]	[KLZ20]	[KBA+21]	[MAAJ19]	[NNP+20]	[PAHC19]	[RB17]	[SKKM09]	[SGB16]	[SLVH19]	[SLDVH21]	[WRO09]	[Yi19]
DSU					X	X		X	X	X				X	
Containerization/Partitioning	X				X	X					X	X	X		
Blue-Green Deployment		X	X		X						X				
Canary Deployment		X	X								X				
Microservices	X		X		X										
State Transfer	X				X			X						X	
Situation-Aware Updating				X					X						
Modeling of Update Strategies			X	X											
New Design Architecture		X	X	X			X	X							X

Table 4.6: Mapping of Papers to Dynamic Update Methods

step, the model is annotated with information about the situations in which the update can occur and the WCET. During the third step, the imperative deployment model is generated, which contains the operations to achieve the target deployment model and takes into account the annotations. If all situational conditions are met and there is enough time to safely perform the update and also apply possible compensatory measures (rollback), the update is performed [KLZ20].

Naseer et al. [NNP+20] propose a framework used at Facebook to enable disruption-free global releases. Their approach aims to avoid any limitations for users and their user experience, to have no downtime, and to maintain cluster capacity, as well as the robustness of the infrastructure. Their framework uses three update mechanisms to meet the requirements of different tiers of their architecture: (1) socket takeover, (2) partial post-reply, and (3) downstream connection reuse. Compared to a traditional upgrade approach, where updates are rolled out in batches and the restarting instances enter the draining mode and thus do not accept new connections until the end of that period, their approach achieved reduced completion times and an improved cluster capacity [NNP+20].

Table 4.6 provides a list of the main approaches, concepts and strategies to perform dynamic updates of software components and the corresponding papers in which they are stated. DSU as an update method is mentioned quite frequently in the analyzed literature, as it is a very broad term for updating or modifying systems at runtime without stopping and restarting the system. For stateful applications, this dynamic updating also requires a method for a state transfer between the old and updated version of an service or application. Very often, dynamic update methods also use containerization and orchestration techniques and deploy applications as microservices. Some authors also developed an entire framework or architecture to implement such update strategies. Moreover, some papers discuss the switch between the legacy and updated versions, as well as their advantages and disadvantages. A few papers also chose a model-based approach to address this subject.

Table 4.7 provides a list of aspects mentioned by researchers in their work that are relevant for evaluating dynamic updating methods and for implementing such strategies in practice. In all the papers examined, it is pointed out that the updating method used

Relevant Aspects for DSU	[VSTK19]	[AS18]	[BEdPT18]	[KLZ20]	[KBA+21]	[MAAJ19]	[NNP+20]	[PAHC19]	[RB17]	[SKKM09]	[SGB16]	[SLVH19]	[SLDVH21]	[WRO09]	[Yi19]
Functional / Nonfunctional Correctness									X						X
Timing Constraints	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Additional Delays												X	X		
Cycle Times					X									X	
Fallback Strategy / Backwards Compatibility		X		X	X			X		X					
Quality Assurance		X				X		X	X						
Infrastructure Requirements														X	
Complexity									X		X				
Situation Awareness				X											
Loose Coupling/Portability									X						X

Table 4.7: Relevant Aspects Mentioned in Literature for Dynamic Updates

must comply with certain time constraints. This includes the requirement that there must be zero-downtime, the WCET of update procedures, the completion of updates in a fixed time slot, or the availability of services in general. Therefore, cycle times and the execution of updates within these cycles are a requirement especially for real-time systems. Furthermore, depending on the specific update strategy, the quality assurance of the updates installed and the possibility of going back to the previous version in the event of errors are properties that are regularly mentioned. Other aspects are not mentioned as frequently because they are more specific to a particular research topic in the broader field of dynamic updating methods.

Description of the Aspects Relevant for DSU

The following paragraphs briefly describe the aspects listed in Table 4.7.

Functional and Nonfunctional Correctness: Functional Correctness comprises the need that new components do not interfere with the existing system. Nonfunctional Correctness describes the requirement that the platform has still enough resources to not be overloaded and to meet timing requirements. Real-time systems thus have a deterministic input-output and timing behavior [Yi19].

Timing Constraints: In many real-time environments, stopping a program for update reasons is unacceptable because they are mission-critical and/or safety-related software systems or the interruption incurs significant costs or leads to prohibitive losses [MAAJ19], [SKKM09]. Taking into account the WCET of update processes also goes in this direction, since in systems where users or processes in general should not be interrupted, there is only a limited time available to complete the update [KLZ20]. In addition, time constraints also include the aspect of service availability. This means that highly available systems should be accessible at least 99.999% of the time [VSTK19]. Another aspect of time constraints is disruption to web application users when they lose their connection and then experience a degradation in quality of experience [NNP$^+$20]

Additional Delays: Using additional software components such as Docker for virtualization introduces an additional processing overhead by adding another layer to the software stack [SLVH19]. This must be taken into account because it could possibly violate the real-time requirements of some systems, since response or execution times can no longer be met.

Cycle Times: These are the time periods that the jobs have available to complete their task before the next cycle begins. Typical cycle times for controllers are between 10 and 1000 ms [KBA+21].

Fallback Strategy/Backwards Compatibility: This refers to the ability to revert to the old version quickly, or even within the same update cycle, if the new version does not meet the expected or defined requirements.

Quality Assurance: This measure ensures that a new version of a software component is tested and validated in a secure environment before the update is released into production. Only if the required quality criteria are met, the update will be accepted.

Infrastructure Requirements: Sometimes it is required that a certain software or commercial, unmodified operating system is used, for example, to fulfill support conditions or for other business reasons [WRO09].

Complexity: This criterion describes how complicated or time-consuming it is to implement the deployment strategy or how much know-how is required. It also takes into account how long the technology has been around, as newer ones are often more complex due to less information and guidance being available [SGB16].

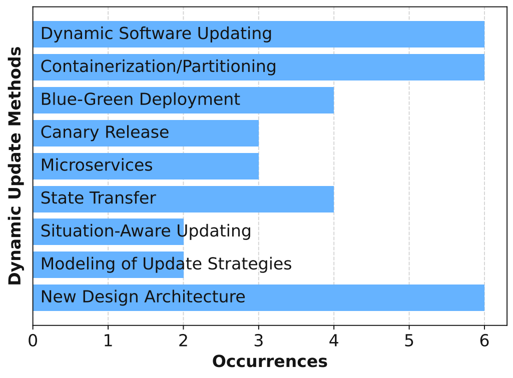
Situation Awareness: This means that the context of a system or application in the course of an update should be observable and taken into account by the updating process. This includes factors such as time, application state, environment, and people involved [KLZ20].

Loose Coupling / Portability: Components should be completely isolated so that updating one does not require a rebuilt of the entire system. In addition, the use of special features of compilers or architectures and the modification of standards should be avoided [RB17].

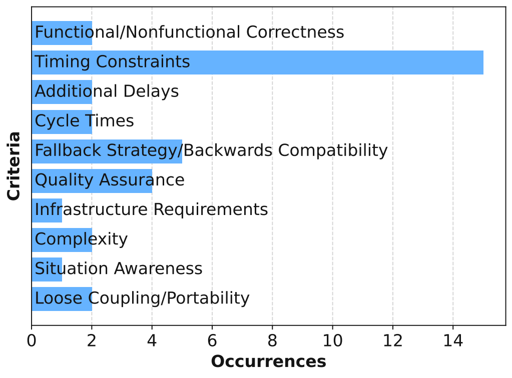
Figure 4.2a quantitatively summarizes the dynamic update methods and concepts for achieving such on-the-fly updates mentioned above. The vertical axis of this bar chart lists the methods and concepts, while the horizontal axis indicates the number of mentions in the relevant papers. As with the chart for the first part of the SLR, a paper can be assigned to several methods or concepts. In Figure 4.2b, the aspects that are either required for the evaluation of dynamic updating methods in practice or are an essential part thereof are summarized. Again, multiple assignments are possible.

4.2.2 Answer to Research Question RQ1b and RQ1c

As shown in Table 4.5, 66 papers were initially found that matched the search terms. After reviewing the title and abstract, 18 papers were selected for thorough reading.



(a) Mentioned Update Methods in Papers



(b) Mentioned Relevant Aspects in Papers

Figure 4.2: Occurrences of (a) Update Methods and (b) Dynamic Update Aspects

Finally, 15 papers were considered as relevant to answer the research questions *RQ1b* and *RQ1c*. The following sections therefore address these questions.

RQ1b: Which dynamic update methods for software components in the production environment are mentioned in existing literature and which software frameworks do they use? Because dynamic updating of software components implies that updates are performed "on-the-fly" without disrupting running operational programs or users and to avoid halting and restarting a systems, a widely used approach is DSU. DSU is a fairly broad term for updating or modifying systems at runtime. Depending on the specific use case, this method can have different characteristics, but the underlying idea remains the same. For example, if a stateful service in an embedded real-time system needs to be updated, the reliable transfer of internal state information from the old component to the new version of the component is also required, including possible changes in data types, commands, etc. For stateless applications, on the other hand, switching between old and new versions is much easier. Furthermore, a DSU strategy for containerized microservices has different technical and operational requirements than for an embedded real-time system that runs directly on the hardware and thus relies heavily on operations and functionalities provided by the operating system and the programming language.

In recent years, virtualization and containerization technologies are also used in the field of dynamic software updates. Docker and Kubernetes are therefore widely used software frameworks for deploying various applications, for example, as microservices, and updating them by deploying a container that hosts the new version and reconfiguring the dispatcher to serve the user or other applications with the output of the updated service. In addition, state transfers can also be an issue in virtualized environments. Furthermore, especially in the context of containerized applications or the partitioning of applications into staging and production areas, or when an application is initially validated in a quarantine environment, a switchover strategy from the old to the new version is essential. Thus, blue-green deployments and canary releases are regularly mentioned in the literature in order to perform such switches, with the strategy chosen depending on the anticipated objectives.

Some authors have also proposed their own framework or architecture for implementing dynamic update strategies. Other, less frequently mentioned approaches to address challenges in dynamic updates include situational awareness of systems and their respective update processes and modeling of update strategies to assess possible impacts on the system already in the architectural model.

RQ1c: Which requirements are mentioned in the literature that are relevant for dynamic deployments in production environments? Since dynamic update approaches are often used for embedded real-time systems, time-sensitive applications, or systems that require high-availability overall, time constraints are the most important requirement mentioned in the literature. This includes, for example, the demand for zero-downtime during the update process and no interruption of any running operational routines or users, as well as ensuring that upgrades can be completed in the cycle slack time, including state transfers and taking into account the WCET. Two other requirements that were mentioned quite frequently are a possible fallback or rollback strategy and the necessity to approve a version in a safe environment first. The former refers to the ability to quickly or even immediately revert to the original replaced version in case an error occurs during the update process or the updated version of an application does not perform as expected. The latter aims to avoid such situations in the first place, as new applications are first validated in a safe or isolated environment before going into production.

Adding an additional software layer by introducing virtualization or containerization techniques can also lead to additional delays and processing overhead, which could affect the real-time requirements of some systems. Therefore, this should also be considered when designing dynamic update strategies. Further aspects that are not as frequently mentioned in the literature include the consideration of infrastructure requirements, complexity of technologies and their implementation, and the importance of systems being aware of their environment and context when performing updates. For some authors, loose coupling of components and portability is also important, as this allows for component-wise adaptability. This facilitates updates of isolated components without having to rebuild or validate the entire system.

Based on these requirements, the criteria for evaluating the identified approaches for a model switching strategy for fault detection models in the production domain are derived in Chapter 5.

4.3 Integration of Models into an Asset Administration Shell

An essential part during the development of a concept for the dynamic exchange of ML models in production environments is the possibility of integrating these models into an AAS. However, for the integration of ML models into an AAS there is no SLR conducted, since, to the best of our knowledge, there exists very little literature regarding this topic so far. Nevertheless, there is a metamodel description for the AAS as well as documentations for AAS implementations that can be used to evaluate the current state of the literature in terms of integrating models into the AAS. In the following, these key aspects mentioned in the literature are summarized, while in Section 3.2 the general aspects of the AAS are discussed.

Rauh et al. [RGB+22] propose a management approach for AI artifacts and integrated their lifecycle into the digital twin domain. For this purpose, they defined a semantic specification model, based on the AAS metamodel, to describe AI models and their associated dataset and learning algorithm entities. Their defined AI assets can be divided into three groups of entities: (1) model, (2) dataset, and (3) learning algorithm. To this end, they use AAS submodels to describe general but also specific properties. For example, all entities contain an identification, contact information and an appendix property. Entity-specific properties include, for example, references to the dataset and learning algorithm, training phase information such as hyperparameters, validation information, and properties describing the deployment and operation. The implemented proof of concept consists of an automatic generation of AI AAS instances using the Eclipse BaSyx Framework and incorporating metadata from the MLOps pipelines. In this context, the trained AI model is exported in the ONNX format and is tightly linked to the metadata to finally obtain a self-describing model. As a downside to their solution, they pointed out that there is currently no direct integration for an AAS into AI frameworks, but they see their approach as a starting point for a standardization process using already supported protocols [RGB+22].

A logical, generic, and technology-independent model for predictive maintenance in the context of a smart factory is proposed by Cavalieri and Salafia [CS20a], which uses the AAS as a standardized abstraction layer to overcome the technological heterogeneity of assets in manufacturing environments. For this purpose, they defined the required general functions for a predictive maintenance solution as so called logical blocks. In this context, these blocks are modular but cooperating elements that group functionalities, such as data acquisition, data manipulation, or the predictive model, and abstract the actual necessary operations. These logical blocks are then implemented in submodels in their AAS, with the exception of the prediction model and maintenance actions, as these were considered too computationally intensive to be implemented on AAS-enabled devices. However, they also point out that this is not a limitation, since for certain solutions the functionalities of the predictive model can also be implemented in a submodel. In their case study they demonstrated this approach based on an AAS-based predictive maintenance model for 100 milling machines, which also leverages the concept of logical blocks. The majority of the logical blocks are implemented as submodels on edge devices, while the predictive model and maintenance decision making reside in the cloud [CS20a].

In a further paper, Cavalieri and Salafia [CS20b] proposed a submodel template to describe an IEC 61131-3 program and its relationships with PLCs and the respective production facility using the AAS metamodel. The IEC 61131-3 standard defines the syntax and semantics for PLC programming languages. To this end, the authors defined the elements of such an IEC 61131-3 configuration, such as resources, program organization unit, tasks, and so on, using elements from the AAS metamodel. Therefore, an AAS is required for the PLC where the program is executed. This AAS contains the previously defined submodels and enriches it with references to the real I/O connections provided by the PLC. After all, this approach supports tasks in the lifecycle of a production system and facilitates, for example, the test operation of a plant, maintenance during runtime, reconfiguration processes of plants, but also documentation in general [CS20b].

Göllner et al. [GPS21] propose an automated and generic generation process for dynamic simulation models based on the AAS as standardized information model containing the necessary machine topology. The generation process uses the Functional Mockup Interface (FMI)-standard and consists of a two-step procedure that allows different simulation tools to be linked. The FMI-standard is used to represent a dynamic simulation model as a black box, but where the interfaces and parameters are visible and the latter can be adjusted accordingly. Within an AAS, the information is encapsulated in submodels that can contain properties or also files. Accordingly, specific submodel templates are defined as standards in order to enable a uniform provision of information by different companies. Furthermore, a simulation model can also be integrated into a machine's AAS submodel by referencing the simulation model file and additionally providing the input and output interfaces [GPS21].

Juhlin et al. [JKS$^+$22] describe a cloud-enabled simulation platform for drive-motor-load simulations with a focus on composite assets – that are systems that consist of multiple assets – and therefore require a systematic integration and synchronization of the individual properties. Thus, they use the concepts of the AAS and Functional Mockup Units (FMUs) to address existing challenges in composite system engineering, such as the lack of interoperability at the system level, both data and component-wise, software deployment and administrative overhead, or security concerns during information exchange. The former allows for automatic data exchange while the latter enables interoperable simulations. To enable a software independent integration model, AAS submodels were used, which are linked to each other, but also refer to proprietary data models. The simulation models can be included in AAS submodels as FMU model files or by using AutomationML. In addition, simulation results and parameter files can also be added to the submodels. The basic structure of the proposed simulation platform is based on a cloud solution using containerization and runs the actual simulation. Based on a mapping, the implemented Drive-Motor-Load Integration Model links asset models from different manufacturer tools as well as from different engineering phases via their inputs and outputs. After the integration model is set up and linked, the simulation models and data can be exchanged. The load simulation model can be developed in any tool and then loaded into the simulation platform as FMU. With the parameters and settings contained in the submodel, the relevant information can be linked and subsequently provide the simulation with the required parameters. Moreover, this architecture and the information it contains make it possible to perform advanced tasks such as predictive maintenance or asset recommendations [JKS+22].

The IDTA[3] is an association founded by the two major industry associations in this field (ZVEI[4] and VDMA[5]) and several companies active in the field of Industry 4.0. Its aim is to accelerate digitization in industry by simplifying and standardizing the individual developments of the industrial digital twin, using the AAS with the idea of establishing it as an international standard, as the world moves towards more interoperability and standards [Bö21]. On its homepage[6], the IDTA summarizes use cases from the industrial practice as well as submodel templates that describe content-related or functional aspects of assets [IDT23]. However, most of the accessible use cases do not deal with the integration of external software components into an AAS. In addition, a majority of the listed submodels have a status of “In Review” or “In Development” and primarily address data encapsulation and data sharing issues.

One submodel template called *Provision of Simulation Models*[7] addresses the interoperable provision of simulation models by providing them as ASCII or binary files to be used with a simulation software. In addition, they provide information about the type, use, and application areas of the model. Moreover, the assets to be used for such a simulation model must also have their own AAS [IDT23].

There are four more submodel templates listed on the IDTA homepage that have promising names and descriptions: (1) *Artificial Intelligence Dataset*, (2) *Artificial Intelligence Deployment*, (3) *Artificial Intelligence Model Nameplate*, and (4) *Predictive Maintenance*. However, all four of these templates are in the “In Development” state and there is no information about these submodel templates other than a brief description. The first template covers the unique identification and explanation of datasets used to train and subsequently instantiate AI models. In addition, other metadata descriptions such as data source, dataset properties, and references to the origin should be captured in a submodel. The second submodel template is intended to encapsulate the necessary information associated to the operation of AI models such as runtime and dependencies, but also deployment requirements, which in turn should facilitate automatic and dynamic deployments of AI models. The AI Model Nameplate is supposed to provide a unique identification and explanation of AI models. This includes, for example, the type of

[3] https://industrialdigitaltwin.org/en/
[4] https://www.zvei.org/
[5] https://www.vdma.org/
[6] https://industrialdigitaltwin.org/en/content-hub/submodels
[7] https://industrialdigitaltwin.org/en/wp-content/uploads/sites/2/2023/01/IDTA-02005-1-0_Submodel_ProvisionOfSimulationModels.pdf

model, the class of the training algorithm, information about the input and output data, but also the possibility to include the artifact as a binary file. The fourth submodel template shall make it possible to describe the predictive maintenance process along with its sub-processes in a structured form based on industry standards [IDT23].

4.3.1 Summary

To conclude this section on integration approaches for external software components into an AAS, the most important aspects of the previously discussed literature are summarized.

The AAS metamodel itself offers basically two ways to include external data, i.e., files or binary data, namely via the submodel elements *File* and *Blob*. This is also true for the papers mentioned here. Basically, if at all, relevant software components such as AI models or simulation models are packaged in such submodel elements and additionally supplemented with all necessary information, resulting in a comprehensive representation of such components as a digital twin.

For example, Rauh et al. [RGB$^+$22] complemented in their AI AAS the binary model in ONNX format with textual metadata to obtain a self-description of these AI models. Also, Göllner et al. [GPS21] followed a similar approach. They integrated a FMU model file and supplemented this simulation model with references to the digital representation of the physical components that provide the required inputs and receive the outputs of the performed simulation [GPS21]. In addition, Juhlin et al. [JKS$^+$22] also mention that simulation models can be integrated into an AAS submodel if it is exported in the appropriate format.

Cavalieri and Salafia [CS20a] said that it is possible to incorporate a predictive maintenance model into a submodel, but did not do so in their work because of the excessive computational complexity for the device used. In their other work, Cavalieri and Salafia [CS20b] modeled IEC 61131-3 elements with AAS metamodel elements, but did not deal with the integration of whole software components.

The simulation model submodel template on the IDTA website is essentially the equivalent of the core statement about integrating simulation models into an AAS mentioned above. Moreover, the four submodel templates referred to, for which no further details are yet specified, provide a strong indication that it is also possible to integrate AI models into AAS submodels [IDT23].

CHAPTER 5

Criteria and Requirements Definition

The aim of this chapter is to define a list of relevant criteria and requirements, both for the integration concept and for the model deployment and switching strategies. The subsequent Chapter 6 introduces a detailed approach for integrating fault detection models into an AAS. Afterwards, in Chapter 7, an exploration of model deployment and switching strategies designed to minimize downtime during model switching is presented.

The defined list of criteria and requirements is to a large extent based on the insights gained from the literature review in Chapter 4, especially on the relevant aspects identified for ML model deployments and dynamic update approaches for software components. The remaining criteria are derived from the properties of the AAS metamodel as well as the characteristics of the BaSyx implementation of the AAS. The criteria and requirements relevant for this use case can be categorized into three groups: (1) Deployment Concepts, (2) Dynamic Updates of Software Components, and (3) AAS Related Aspects. The details of each group are elaborated in the following sections.

5.1 Deployment Concepts

The following list of criteria and requirements primarily originates from the findings of the SLR on deployment strategies for ML models from Section 4.1. Specific solutions identified by the SLR on this topic are directly mapped to the requirements for dynamic updates in Section 5.2 to avoid duplicates and overlaps. Hence, the integration approach of ML models into an AAS and especially the deployment of these models should consider the following points:

- **R-5.1.a - Containerization/Partitioning** The concept of using containerization with products such as Docker in combination with orchestration tools like

Kubernetes was mentioned quite frequently for deployment concepts as well as dynamic updates. This enables, among other things, microservice deployments of applications or their publication as a web service and the provision of their functionalities via an API. In addition, this concept has the advantage of providing pre-configured images that can be easily deployed on a variety of systems running different environments.

- **R-5.1.b - Microservices, Web Services, API** Both microservices and web services are frequently discussed (both as deployment concepts and for dynamic updates) as software architectures and methods for providing application access to users. Microservices provide a way to structure and deploy different types of applications, including ML models. Typically, microservices are combined with containerization, which requires a well-defined API to provide access to the model. Web services can be used to implement a microservice architecture by exposing the service through a web interface, such as REST APIs.

- **R-5.1.c - Deployment Strategy** Different deployment strategies are mentioned in the literature for deploying ML models and providing them as services and updating them depending on different requirements. Basically, two strategies can be considered for this use case: (1) Blue-Green Deployment and (2) Rolling Deployment.

  Blue-Green Deployment: With this strategy, the model can be prepared and tested in a separate standby environment before switching it to the active model to serve all incoming requests. Furthermore, this strategy allows for zero downtime deployments, but also enables a rollback to the previous model version in case of an unexpected condition regarding the model's performance.

  Rolling Deployment: This deployment strategy can be used when there are multiple instances running and during an update only a part of them is updated while the others can serve incoming requests. In this way, deployment without downtime is possible.

  Therefore, the characteristics of the mentioned deployment strategies should already be taken into account during the development of the integration concept, as they are well-defined ways for deploying different types of applications.

5.2 Dynamic Updates

The following list addresses the aspects that are primarily related to the findings discussed in Section 4.2 about dynamic update methods for software components. Therefore, the identified deployment strategies, but also possible implications in this regard in the integration concept, should take these points into account:

- **R-5.2.a - Dynamic Software Updating (DSU)** This is a broader term for updating or modifying systems during runtime without having to stop and restart the system. Therefore, a key requirement for dynamic updating in this case is to aim for zero downtime when switching between two model versions used for fault detection. Accordingly, timing constraints can also be considered as a sub-criterion of DSU.

Timing Constraints This term, as defined in Section 4.2, includes several requirements related to timing, such as downtime, service availability, or connection losses that result in degradation of the quality of user experience. In this approach, zero downtime and high service availability are the main criteria that should be met. Specifically, the goal is to have no downtime for ML inference, ensuring very high service availability and excellent user experience.

- **R-5.2.b - Fallback Strategy/Backwards Compatibility** A frequently mentioned point in the literature on dynamic updates is the need to quickly roll back to a previous version, in case the new version of the model does not meet the specified requirements or causes an error condition. In addition, backward compatibility should be ensured so that the introduction of a new version of the model does not cause any changes, that would make older versions unusable. One way to accomplish this is by following a blue-green deployment strategy.

- **R-5.2.c - Quality Assurance** This requirement ensures that a new version of a ML model can first be tested and validated in a safe environment before it is released for production. Only if the required quality criteria are met, the switch between the models will actually be performed. Thus, a standby environment should be available to prepare and test the new model in advance. This can be achieved, for example, with a blue-green deployment strategy or by means of A/B testing.

  A/B Testing: This is one way to ensure the quality of new models by comparing at least two model versions in terms of their performance by splitting the incoming requests among the models to be tested.

- **R-5.2.d - Infrastructure Requirements** It is important to consider that the identified deployment strategies must take into account the requirements of other software components that are essential for the definition and implementation of fault detection models within the defined integration concept. As such, these components can be considered as essential infrastructure and cannot be neglected. Therefore, the requirements imposed by the BaSyx SDK and the AAS metamodel should be met. Some of these requirements are defined in the following list in the upcoming section.

5.3 Desired Features of the ML Model Integration Approach

In addition to the requirements that emerged from the literature research, the following list defines AAS-specific requirements that arise either from the AAS metamodel or the BaSyx SDK, along with requirements that are intended primarily for usability and practical applicability. Therefore, the integration approach of ML models into an AAS should provide or enable at least the following points:

- **R-5.3.a - Strive for a Generic Integration Approach**: The integration concept to be developed should be as generic as possible in a way that it should be as independent as possible from the actual implementation. This implies that it should not matter, for example, which type of ML model is used, in which programming language it is implemented or, in principle, how it is hosted or made available. The concept should only define a few fixed aspects, such as interfaces or endpoints. In addition, it should not just be designed for a specific use case. For example, the number or data type of sensor values used should not play a role, nor where the data must be actually read from the AAS.

- **R-5.3.b - Independence from the Deployment Strategy Used**: The fault detection submodel should be designed in a way that it is not tailored to a specific deployment strategy. This means that no submodel elements should be added, which limit this integration concept to a single deployment strategy or are designated for a specific purpose (e.g., by their name). This should allow that the concept does not only work for one deployment strategy, but for several. After all, not for every use case the same deployment strategy is the best one.

- **R-5.3.c - Fault Detection Submodel should form a Logical Unit**: The goal is that the submodel, which represents the fault detection, encapsulates all the necessary properties collected as submodel elements within one submodel. This is necessary to ensure that this submodel can be hosted as a logical unit on any server. Based on that, one further aspect can be derived as a requirement: The fault detection submodel should be able to be hosted as a self-contained unit on a separate server (edge node) in order to be as close as possible to the nodes that provide the data in order to keep latency as low as possible.

- **R-5.3.d - Possibility to Archive Models**: It should be possible to keep old models that were used for fault detection earlier even after their use in the AAS. This facilitates a possible rollback to a previous version if problems occur during the operation of a new version, but also to keep a transparent history of the used models or to compare their properties between the versions.

- **R-5.3.e - Quality Criteria based Decision Making**: It should be possible to switch from an older to a newer version of an ML model based on objective quality criteria. These can be performance metrics depending on the model type. The latest values of these metrics should be persisted in a submodel in the AAS.

5.4 Deliberately Excluded Criteria

Besides the aspects mentioned above that should be fulfilled by the developed integration concept and the identified deployment strategies based on it, there are also some requirements mentioned in the literature that have been deliberately excluded for various reasons, which are explained in the following list.

- **Cycle Times** Although cycle times are a central element in real-time systems, there is no PLC in service in this use case and also in the prototype based on it, and furthermore, no hard real-time requirements are applicable. Thus, cycle times do not play a role for this application.
- **Additional Delays** Additional software components such as Docker for containerization and virtualization add processing overhead and may violate real-time constraints as a result. However, since this use case does not require hard real-time requirements, but only soft real-time requirements, this limitation can be neglected.
- **State Transfer** The ML models used in this work do not have an internal state or state variables that must first be transferred to the new version, which is why just considerations for stateless deployments are relevant.
- **Canary Deployment** This deployment strategy can be used to compare different models in terms of their performance. Additionally, this strategy also allows a rollback if an undesired condition occurs, as well as zero downtime deployments. In the case of fault detection, this type of testing and comparison of models and their performance is not reasonable because live requests trigger an event in the case of a detected fault and might subsequently initiate further implications in a productive system. This can lead to an undesirable state since the suggestions for possible faults come from two different systems. Therefore, other testing strategies are better suited to ensure model quality.
- **Complexity** In Section 4.2, this term was defined as a factor that takes into account the time required and the level of expertise needed, but also the maturity level of the technology used for implementing the deployment strategy. In this approach, primarily state-of-the-art components (such as Docker for containerization) should be used and, in general, the level of complexity should be as low as possible but as high as required for the implementation according to the other aspects. However, this is a criterion whose fulfillment/non-fulfillment is very difficult to determine objectively and was therefore taken into account but not included in the above list.

CHAPTER 6

Conceptualization of an AAS Integration Approach

The objective of this chapter is to present a comprehensive approach for integrating fault detection models into an AAS. Based on the criteria and requirements defined above and in particular taking into account the existing AAS metamodel as defined in the specification *Details of the Asset Administration Shell* in the version 3.0RC02 [Pla22], the resulting concept for integrating ML models into an AAS is discussed below. The technical possibilities and limitations of the BaSyx SDK also play an essential role in the development of the integration concept as well as the findings and restrictions resulting from the identified deployment strategies for ML models, which are described in detail in Chapter 7.

6.1 Eclipse BaSyx™ – Particularities and Restrictions

The following sections elaborate on some special features and limitations that the usage of the BaSyx framework impose on the development of the integration concept of ML models into an AAS submodel.

6.1.1 AAS Operations in BaSyx

Through operations in BaSyx it is possible to push runtime behavior to a server and subsequently perform this functionality on the server. This functionality is added during the implementation phase of the AAS using a BaSyx SDK (currently existing for Java, .NET, C++, and Python). In the further course, only the Java side is considered, since this is relevant for the implementation of the prototype (reasons for that are provided in Section 2.2.3 on the methodology of the evaluation).

Considering now just the Java-based SDK, the functionality that can be embedded into a BaSyx operation is quite manifold. The essential condition is that it must be an invokable object set to the operation element and correspond to one of the four types:

- `Function<Object[], Object>`
- `Runnable`
- `Supplier<Object>`
- `Consumer<Object[]>`

As all the defined operations for this concept have both input and output variables, the set invokables are of type `Function<Object[], Object>`. This approach works well in principle as long as these operations are not hosted directly on an AAS server, but on a separate server that hosts only one or more submodels, e.g. on the `BaSyxHTTPServer`. If, on the other hand, the operations are hosted directly on the AAS server (within the identical submodel as before), then this will lead to an error when invoking such an operation. The reason for this behavior is the current implementation of the invocation process within the AAS server. Technically it is possible to provide operations also on an AAS server. However, for security reasons the invocation is not supported at the current state[1].

Operation Delegation One possibility to overcome this issue is to use the operation delegation approach. With this approach it is possible to redirect the invocation of an operation to another server. The called operation can be hosted on an AAS server, but the operation that performs the actual functionality must be published on a separate server. The operation to which the delegation is made can also be implemented in another programming language. Furthermore, there is no need to implement the entire submodel interface [Sch22c].

To define an operation using the delegation approach, it must have neither input nor output variables. In addition, the operation must be declared as a delegation operation by means of a qualifier. The qualifier must have the type `"invocationDelegation"` and the URL of the operation to be invoked must be set as the value of this qualifier. However, when calling the delegation operation, the in and out parameters must be provided. These are automatically passed on to the actual operation. The client must therefore know which In/Out/InOut variables are expected by the actual operation [Sch22c].

6.1.2 Persistency Backend in BaSyx

Within BaSyx, the hosted AAS with all contained submodels and the respective submodel elements can be persisted using the storage backend of the AAS server component, which uses MongoDB as data store. However, this must be configured explicitly, otherwise the *InMemory* backend is used by default [Sch22d].

[1]`https://github.com/eclipse-basyx/basyx-java-components/issues/217`

In addition to text or numeric based submodel elements, it would also be possible to store Java bytecode in a MongoDB collection. However, this would introduce potential problems and is therefore currently not supported by the BaSyx framework[2]. Another point to note in this context is that the storage backend capability is only available for the AAS server, but not for other servers that only host submodels, such as the `BaSyxHTTPServer`. This implies that any elements contained in the same submodel as operations cannot be stored in MongoDB through the standard BaSyx functionality. This is because it cannot be hosted on an AAS server with a MongoDB backend due to the restricted functionality of operations within an AAS server, and on the other hand, if hosted on a separate server due to the lack of an storage backed available for this type of server.

6.1.3 Eventing in BaSyx

The BaSyx framework supports the publication of MQTT events and uses a hierarchical topic structure for this purpose. Eventing exists for both the AAS registry and the various server components provided by BaSyx. In principle, events are published for the creation, modification and deletion of an AAS, submodels or submodel elements or their corresponding values. Since this behavior is impractical for certain submodel element types, for example BLOBs, this can be disabled for defined submodel elements using a qualifier with the type *emptyValueUpdateEvent* and the value `true` [Sch22b]. Furthermore, it is possible to activate the publishing of events only for selected components (submodels, submodel elements, etc.) by means of whitelisting and to deactivate it for the remaining components by default. However, this configuration option exists only for an AAS server [Fis21].

Other configuration options, such as defining custom topics or triggering the publication of messages based on certain conditions, do not work at the current stage of the BaSyx framework. For example, if you only want to publish an event via MQTT in case a potential error has been detected (i.e., if the property *FaultDetected* is set to the value `true`), then this cannot be achieved with the existing functionality of BaSyx.

6.1.4 Implications for the Concept

Based on these findings, there are basically two possible ways to map the necessary functionality by means of operations in the AAS:

1. **Cloud-Edge-Deployment**: This approach implies that there exists at least two types of servers, an AAS server and one or more separate servers that just host one or more submodels. For the integration of ML models this would mean that the AAS is hosted on the AAS server along with the submodels, which contain mainly static information that needs to be persisted in the data store. In contrast, those submodels that contain operations or constantly changing sensor values are

[2]see footnote 1

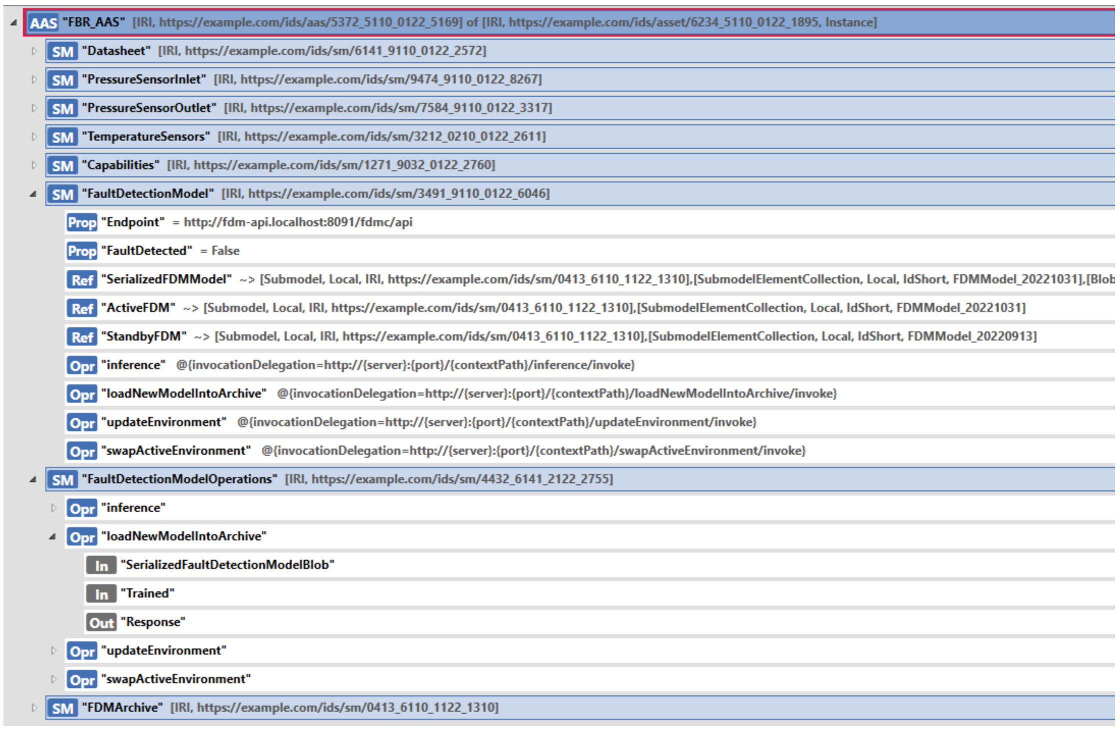
hosted on a separate HTTP server, which hosts the submodels contained in servlets to provide access to their properties. In such a scenario, the AAS server is also referred to as "Cloud Server", while the second server is called "Edge Server", as it is also described on the BaSyx project site[3].

2. **Using Operation Delegation**: This approach means that the submodel covering the elements related to fault detection is split into two separate submodels. The first submodel, which we call *FaultDetectionModel*, contains all the submodel elements required to integrate an ML model, provide access to it, and perform the switch between model versions, as well as the operations that delegate the call. These operations have no operation variables defined, but do have the required qualifier for delegation. The second submodel, *FaultDetectionModelOperations*, contains only the operations with all required operation variables and functionality. The first submodel is hosted on the AAS server along with the AAS itself, while the second submodel must be pushed to an HTTP server in order to invoke the implemented functionality. However, this approach would clearly conflict with the requirement *R-5.3.c* by breaking logical unity and increasing management and maintenance overhead. Figure 6.1 depicts the splitting of the fault detection submodel into two separate ones that will have to be hosted on different servers. On the other hand, this approach would provide the automatic functionality of persisting the submodel elements, such as the references for the active and standby environments or the endpoint in MongoDB using the BaSyx native functionality.

Custom MongoDB Client Since the operation delegation approach, as mentioned earlier, contradicts the requirement *R-5.3.c* and would also double the number of operations modeled, the cloud-edge-deployment approach is chosen for the ML model integration approach. This implies that the *FaultDetectionModel* submodel runs on its own HTTP server and, therefore, the elements within this submodel are not automatically persisted to the MongoDB data storage. This comprises elements like the reference element values to the active and standby environment as well as to the serialized ML model in the archive. Besides that, there is also the requirement that the parameters needed during the implementation of each operation in BaSyx as well as those needed for the ML model deployment must also be persistently stored somewhere.

For this reason, a custom MongoDB client was implemented. This client allows to store the properties contained in the *FaultDetectionModel* submodel (*SerializedFDModel*, *ActiveFDM*, *StandbyFDM*) but also the parameters used in the implementation of the ML model deployment operations in a dedicated collection in MongoDB, such as the currently used Docker image version, the currently actively used container, and so on. This ensures that basically all properties are persistently stored in a MongoDB collection and can be read from it accordingly, like when the submodel is hosted on an AAS server with a MongoDB backend. Thereby, it is guaranteed that after a restart of the AAS server or a

[3]https://wiki.eclipse.org/BaSyx_/_Scenarios_/_Cloud_Edge_Deployment



```
AAS "FBR_AAS" [IRI, https://example.com/ids/aas/5372_5110_0122_5169] of [IRI, https://example.com/ids/asset/6234_5110_0122_1895, Instance]
  SM "Datasheet" [IRI, https://example.com/ids/sm/6141_9110_0122_2572]
  SM "PressureSensorInlet" [IRI, https://example.com/ids/sm/9474_9110_0122_8267]
  SM "PressureSensorOutlet" [IRI, https://example.com/ids/sm/7584_9110_0122_3317]
  SM "TemperatureSensors" [IRI, https://example.com/ids/sm/3212_0210_0122_2611]
  SM "Capabilities" [IRI, https://example.com/ids/sm/1271_9032_0122_2760]
  SM "FaultDetectionModel" [IRI, https://example.com/ids/sm/3491_9110_0122_6046]
    Prop "Endpoint" = http://fdm-api.localhost:8091/fdmc/api
    Prop "FaultDetected" = False
    Ref "SerializedFDMModel" ~> [Submodel, Local, IRI, https://example.com/ids/sm/0413_6110_1122_1310],[SubmodelElementCollection, Local, IdShort, FDMModel_20221031],[Blob
    Ref "ActiveFDM" ~> [Submodel, Local, IRI, https://example.com/ids/sm/0413_6110_1122_1310],[SubmodelElementCollection, Local, IdShort, FDMModel_20221031]
    Ref "StandbyFDM" ~> [Submodel, Local, IRI, https://example.com/ids/sm/0413_6110_1122_1310],[SubmodelElementCollection, Local, IdShort, FDMModel_20220913]
    Opr "inference" @{invocationDelegation=http://{server}:{port}/{contextPath}/inference/invoke}
    Opr "loadNewModelIntoArchive" @{invocationDelegation=http://{server}:{port}/{contextPath}/loadNewModelIntoArchive/invoke}
    Opr "updateEnvironment" @{invocationDelegation=http://{server}:{port}/{contextPath}/updateEnvironment/invoke}
    Opr "swapActiveEnvironment" @{invocationDelegation=http://{server}:{port}/{contextPath}/swapActiveEnvironment/invoke}
  SM "FaultDetectionModelOperations" [IRI, https://example.com/ids/sm/4432_6141_2122_2755]
    Opr "inference"
    Opr "loadNewModelIntoArchive"
      In "SerializedFaultDetectionModelBlob"
      In "Trained"
      Out "Response"
    Opr "updateEnvironment"
    Opr "swapActiveEnvironment"
  SM "FDMArchive" [IRI, https://example.com/ids/sm/0413_6110_1122_1310]
```

Figure 6.1: Potential AAS Design using the Operation Delegation Approach

server on which a submodel is hosted, the state can be restored and all parameters have the value they had before the restart.

Custom MQTT Client Since the existing configuration and customization options of BaSyx regarding MQTT eventing are not sufficient to publish selected MQTT messages with a specific topic whenever a potential error has been detected, a separate MQTT client is used for the realization, which fulfills exactly these requirements. Using Java, this is relatively easy using the `MqttClient` from the package `org.eclipse.paho.client.mqttv3`. This client is also used internally by BaSyx for its MQTT functionality.

To enable eventing (with BaSyx or the custom MQTT client) an MQTT broker must be accessible to which the client can connect and where the events with the corresponding topics are published. For this purpose, a property named *MQTTBrokerEndpoint* was defined in the AAS in the *Datasheet* submodel where the current URL of the MQTT broker is stored. The reason why it is not in the *FaultDetectionModel* submodel but in the "general" *Datasheet* submodel is simply because the MQTT broker cannot be used exclusively for publishing fault detection events but also for other messages, such as if BaSyx's MQTT messages are also used when creating, modifying, or deleting elements.

The topic specified for this purpose is given as *AAS/FaultDetection/FAULT*, and the corresponding message is structured as *({AAS Identifier},{FaultDetectionModel Submodel Identifier},FaultDetected)*. Both the topic and the message can be easily customized by using the custom MQTT client.

The hierarchical topic structure for the message in case of a fault was defined solely in the implementation of the AAS in BaSyx. One could also include this as a property in the *FaultDetectionModel* submodel and always retrieve the current value from the property when publishing a new message. However, since this is not directly related to a dynamic deployment approach for ML models and the topic of MQTT and publishing messages is only marginally touched upon anyway, the decision was made to define the topic just in the implementation in BaSyx.

6.2 UML Diagram of the AAS Integration Concept

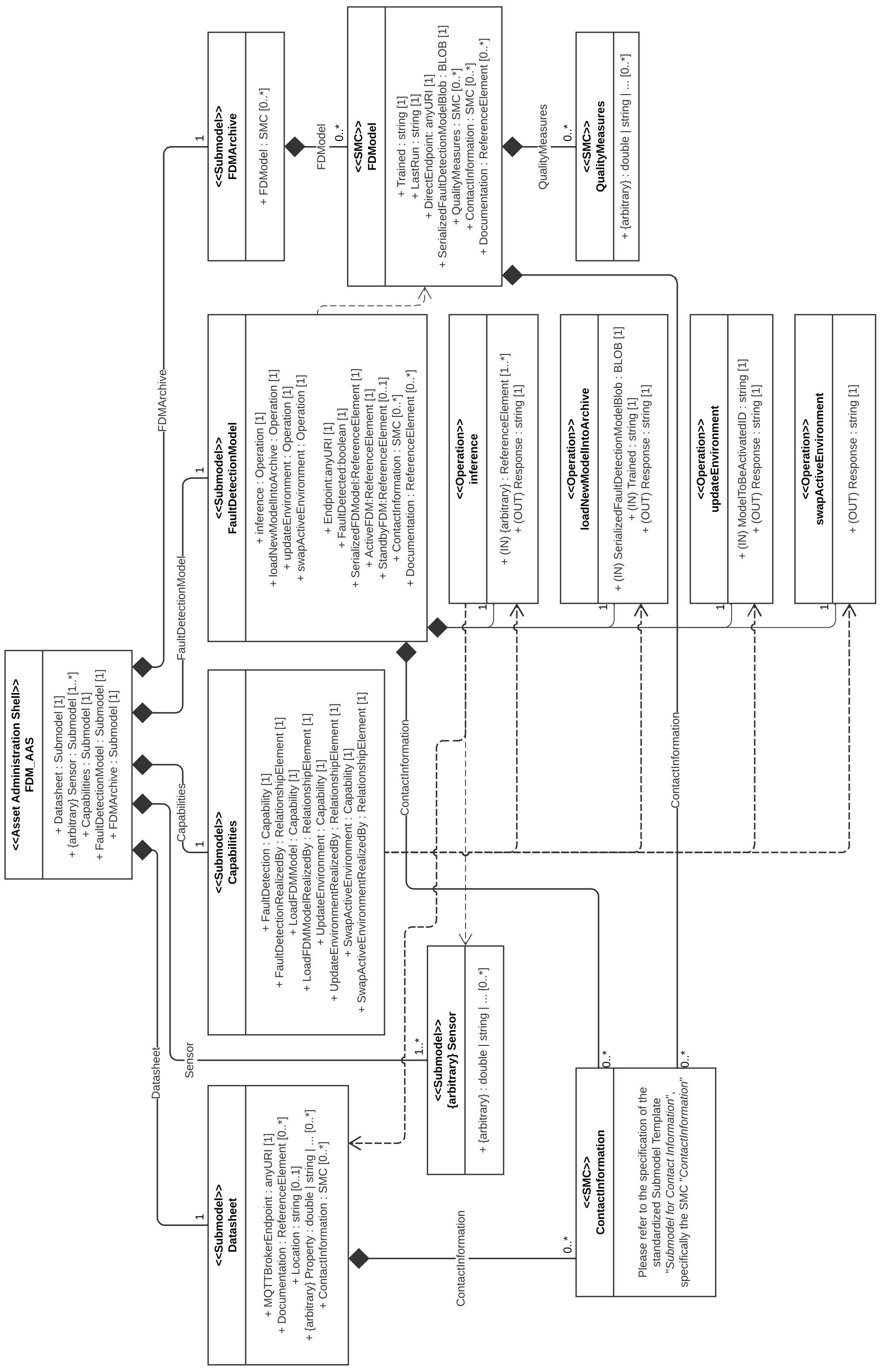
The following section describes in detail the integration concept for ML models into an AAS, in particular for a fault detection model, and is structured as follows: First, an overview of the integration concept is provided. Subsequently, the details of the submodels are presented and in particular the *FaultDetectionModel* and the *FDMArchive* submodel are discussed in detail.

For the concept to be developed, it has been attempted to make it as abstract and generic as possible, so that it can be used for different (industrial) applications. This refers, for instance, to the fact that it is suitable for different machine types, number and type of sensors, but also as independent as possible of specific ML model types. UML class diagrams are used for the graphical representation of the concept using AAS metamodel elements. This diagram type is also used by the IDTA for the documentation of their published submodel templates.

6.2.1 Overview of the AAS Integration Concept

In this section, an overview of the integration concept for a ML model into an AAS with the goal of performing fault detection is presented. Figure 6.2 shows the defined submodels with their respective submodel elements such as properties, operations, reference elements, SMC, and so on. In addition, the dependencies and interrelationships between the different submodel elements of the various submodels are depicted.

The UML diagram in Figure 6.2 shows that an AAS for a ML fault detection approach consists of at least five submodels, of which each serves a different purpose. The composition link between the AAS and the submodels, but also between the submodels and further operation elements or SMCs have specified not only the name to which they refer, but also a multiplicity. This multiplicity is also specified directly at the element in the respective container and determines whether this element is mandatory or optional in this context, but also if this element can occur more than once, such as sensor submodels or quality measures for ML models. The dependencies (dashed lines with arrows) between submodels and operation elements or SMCs represent the demand



<<Asset Administration Shell>> FDM_AAS
+ Datasheet : Submodel [1]
+ {arbitrary} Sensor : Submodel [1..*]
+ Capabilities : Submodel [1]
+ FaultDetectionModel : Submodel [1]
+ FDMArchive : Submodel [1]
<<Submodel>> FDMArchive
+ FDModel : SMC [0..*]
<<SMC>> FDModel
+ Trained : string [1]
+ LastRun : string [1]
+ DirectEndpoint: anyURI [1]
+ SerializedFaultDetectionModelBlob : BLOB [1]
+ QualityMeasures : SMC [0..*]
+ ContactInformation : SMC [0..*]
+ Documentation : ReferenceElement [0..*]
<<SMC>> QualityMeasures
+ {arbitrary} : double | string | ... [0..*]
<<Submodel>> FaultDetectionModel
+ inference : Operation [1]
+ loadNewModelIntoArchive : Operation [1]
+ updateEnvironment : Operation [1]
+ swapActiveEnvironment : Operation [1]
+ Endpoint:anyURI [1]
+ FaultDetected:boolean [1]
+ SerializedFDModel:ReferenceElement [1]
+ ActiveFDM:ReferenceElement [1]
+ StandbyFDM:ReferenceElement [0..1]
+ ContactInformation : SMC [0..*]
+ Documentation : ReferenceElement [0..*]
<<Operation>> inference
+ (IN) {arbitrary} : ReferenceElement [1..*]
+ (OUT) Response : string [1]
<<Operation>> loadNewModelIntoArchive
+ (IN) SerializedFaultDetectionModelBlob : BLOB [1]
+ (IN) Trained : string [1]
+ (OUT) Response : string [1]
<<Operation>> updateEnvironment
+ (IN) ModelToBeActivatedID : string [1]
+ (OUT) Response : string [1]
<<Operation>> swapActiveEnvironment
+ (OUT) Response : string [1]
<<Submodel>> Capabilities
+ FaultDetection : Capability [1]
+ FaultDetectionRealizedBy : RelationshipElement [1]
+ LoadFDMModel : Capability [1]
+ LoadFDMModelRealizedBy : RelationshipElement [1]
+ UpdateEnvironment : Capability [1]
+ UpdateEnvironmentRealizedBy : RelationshipElement [1]
+ SwapActiveEnvironment : Capability [1]
+ SwapActiveEnvironmentRealizedBy : RelationshipElement [1]
<<Submodel>> {arbitrary} Sensor
+ {arbitrary} : double | string | ... [0..*]
<<Submodel>> Datasheet
+ MQTTBrokerEndpoint : anyURI [1]
+ Documentation : ReferenceElement [0..*]
+ Location : string [0..1]
+ {arbitrary} Property : double | string | ... [0..*]
+ ContactInformation : SMC [0..*]
<<SMC>> ContactInformation
Please refer to the specification of the standardized Submodel Template "Submodel for Contact Information", specifically the SMC "ContactInformation"
FDMArchive
FaultDetectionModel
Capabilities
Sensor
Datasheet
FDModel
QualityMeasures
ContactInformation
1
1..*
0..*

Figure 6.2: UML Diagram of the AAS Integration Concept - An Overview

for information, such as sensor values, or because the reference elements refer to elements in the other submodels, as it is the case with the active standby fault detection model.

The following list provides the necessary details on the five submodels required to model the digital representation of a fault detection approach:

- **Datasheet**: The *Datasheet* submodel, which was defined in the course of the concept development, contains primarily general information of the machine for which the fault detection is performed and properties that are not intended to be used solely by the fault detection model. Therefore, this submodel contains:
  - **MQTTBrokerEndpoint**: This mandatory property of the type *anyURI* contains the URL of the MQTT broker that is used for publishing MQTT events in case of an identified fault. However, this endpoint can also be used for other purposes of publishing MQTT events, such as for simple threshold monitoring of sensor values.
  - **Documentation**: This property of type *ReferenceElement* is optional and can be used to link general information about the machine, such as sketches, plans, maintenance documents, manuals, etc., to the AAS. Within the *ReferenceElement*, one or more references can be defined to refer to a variety of elements within an AAS but also to files and URLs that do not necessarily have to be included in any AAS.
  - **{arbitrary} Properties**: The *Datasheet* submodel can also contain any number of further properties that are required to describe the machine or are helpful to the overall topic of fault detection in this AAS. These properties can have basically any data type that is provided by the AAS metamodel. In addition, these properties can also be used for the fault detection itself as they may contain values that are relevant for determining potential faults. Obviously, this depends very much on the intended use case, but such properties include for instance the date of the last maintenance or the number of loading cycles since the last cleaning.
  - **ContactInformation**: The *ContactInformation* submodel is already a submodel template standardized and published by the IDTA. This submodel template by Bayha et al. [BBB+20] called *"Submodel for Contact Information"* and in particular the SMC named *"ContactInformation"* describes in detail all possible fields in order to clearly reach the right contact person for the service avoiding any ambiguity.

    For this scenario, the optional *ContactInformation* submodel is used not only in the *Datasheet* submodel but also in two further submodels. It is used to provide the possibility to store contact data of responsible persons for a module, be it for the machine and general information or, for example, for the fault detection model.

- **{arbitrary} Sensor**: Within an AAS for fault detection, any number of sensor submodels can be defined. The sensor submodel can be used to model any physical sensor that provides values that are used for the fault detection inference. Moreover, the submodel element contained in a sensor submodel, which actually stores the measured value, can be basically of any data type that is supported by the AAS metamodel, like integer, floating point, or string data types. The name of the encapsulated variable is not of importance. Of course, multiple submodel elements can be defined for sensor values within a sensor submodel. During operation, the sensor values are usually updated periodically with the latest values. How this can happen is beyond the scope of this paper. In this context, it is merely assumed that this happens periodically.

- **Capabilities**: This submodel contains the capabilities that this AAS provides for fault detection functionalities. The capabilities basically describe the functionalities in an implementation-independent way. Therefore, in this submodel, there is always a capability along with a "RealizedBy" relationship element that links the capability to the operation that actually provides the functionality. A relationship element has always two values. The first is the reference to the capability element, the second is the reference to the operation.

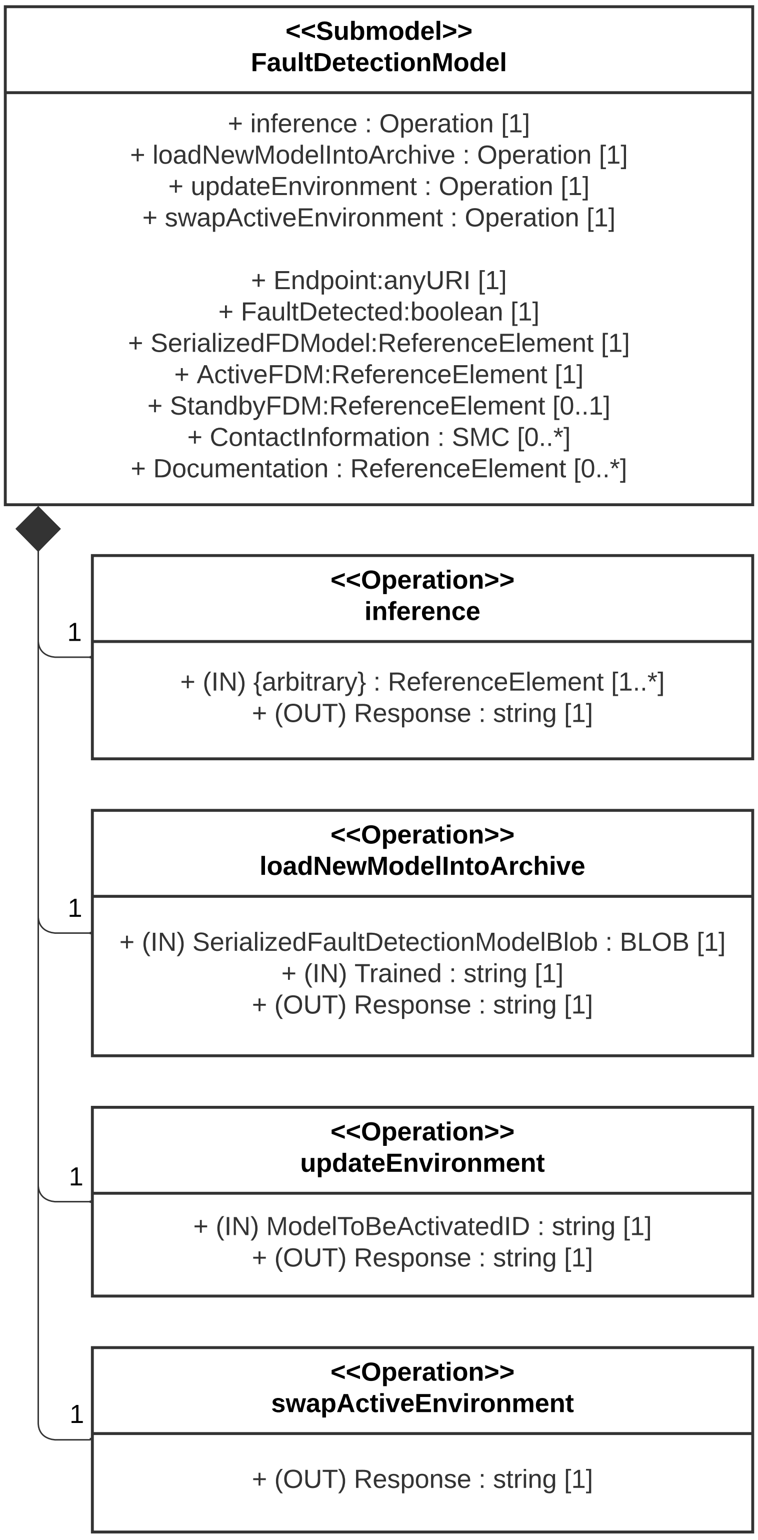
  For performing the actual tasks of ML inference, ML model upload to the AAS, or switching between model versions, the capabilities are not required because they were not needed for the implemented prototype. However, they have been included in the concept to both demonstrate the available features of the AAS metamodel and to provide the basis for further tasks such as capability testing, which checks whether the provided capabilities of a resource match the required ones [BBB+20]. However, these tasks have not been addressed in this paper.

- **FaultDetectionModel**: The *FaultDetectionModel* submodel contains all those elements together with the operations that are directly related to performing fault detection using a ML model. Further details on this submodel are provided in Section 6.2.2.

- **FDMArchive**: Within the *FDMArchive* submodel, all current and previous ML models that are or were used for fault detection are stored along with their necessary properties. Additional information on this submodel is given in Section 6.2.3

6.2.2 Fault Detection Model Submodel

This section describes in detail all submodel elements required to achieve the desired goal of performing ML inference, preparing new ML models, and switching between model versions. Figure 6.3 shows a section of the overall diagram and focuses exclusively on the representation of the fault detection model without depicting the dependencies to other submodels.



<<Submodel>>
FaultDetectionModel
+ inference : Operation [1]
+ loadNewModelIntoArchive : Operation [1]
+ updateEnvironment : Operation [1]
+ swapActiveEnvironment : Operation [1]
+ Endpoint:anyURI [1]
+ FaultDetected:boolean [1]
+ SerializedFDModel:ReferenceElement [1]
+ ActiveFDM:ReferenceElement [1]
+ StandbyFDM:ReferenceElement [0..1]
+ ContactInformation : SMC [0..*]
+ Documentation : ReferenceElement [0..*]
1
<<Operation>>
inference
+ (IN) {arbitrary} : ReferenceElement [1..*]
+ (OUT) Response : string [1]
1
<<Operation>>
loadNewModelIntoArchive
+ (IN) SerializedFaultDetectionModelBlob : BLOB [1]
+ (IN) Trained : string [1]
+ (OUT) Response : string [1]
1
<<Operation>>
updateEnvironment
+ (IN) ModelToBeActivatedID : string [1]
+ (OUT) Response : string [1]
1
<<Operation>>
swapActiveEnvironment
+ (OUT) Response : string [1]

Figure 6.3: UML Diagram of the *FaultDetectionModel* Submodel

First, the elements of type property and reference element are discussed, then the operations are explained in detail. The following list describes the first element group in detail:

- **Endpoint**: This property of the type *anyURI* contains the URL of the REST API endpoint of the ML model used for inference. This endpoint normally retains its value during runtime, even when switching between two model versions. It could only be changed if the whole setup with the container and the REST API configuration is changed intentionally.
- **FaultDetected**: This element of type *boolean* always captures the response of the most recent call of the inference operation. Therefore, this value can potentially change during runtime. Normally it has the value `false`, unless a potential fault has been detected, in which case the value is set to `true`. In addition, the path to this property is published within the MQTT message as AAS identifier, submodel identifier and property ID in case a fault was detected.
- **SerializedFDModel**: The *SerializedFDModel* property references the currently active ML model that is stored in one of the *FDModel* SMCs in the *FDMArchive* submodel. More precisely, the serialized BLOB element is referenced here. After a swap of the actively used ML model, this reference is updated as well.
- **ActiveFDM**: This property references the *FDModel* SMC in the *FDMArchive* that is currently used as the active one. This means that this model is used for inference.
- **StandbyFDM**: The *StandbyFDM* property is similar to the *ActiveFDM* but refers to the model that is currently not actively used for inference. Furthermore, this property is not mandatory as it depends on the actual deployment strategy chosen whether a standby environment is also available or not.

  During the swap between two model versions, the *StandbyFDM* and the *ActiveFDM* references are exchanged (if this is required by the deployment strategy). The *StandbyFDM* property is used to specify the model that is ready to be set as the active model, but also to keep it as a potential rollback option after a switch until it is overwritten by a reference to a newly uploaded model.
- **ContactInformation** and **Documentation**: This submodel contains also the two optional elements *ContactInformation* and *Documentation*. The roles of these two elements are basically identical to those already described in the *Datasheet* submodel.

Besides the above mentioned properties and reference elements included in this submodel, the defined operations are also an essential part of it. Therefore, the operations that are also represented in Figure 6.3 are explained in the subsequent sections, together with their input and output parameters. Each operation has input and/or output variables which are a submodel element of type *OperationVariable*. An *OperationVariable* has a value defined that can be one of a specified list of submodel elements, such as property, BLOB, SMC, or reference element. The level of *OperationVariables* was not considered in the UML diagram for reasons of clarity, and for the property elements the specific data type of the property value was specified directly.

inference This operation defines the functionality of sending the specified (sensor) values to the respective ML model that is currently defined as active in order to potentially detect a fault. The operation requires the following parameters:

- **IN: {arbitrary} Sensor Value**: This operation requires an arbitrary number of sensor values, or any properties that contain a value, as an input variable that is needed by the ML model for inference. Each of these input variables references a property in one of the sensor submodels or the *Datasheet* submodel that capture the respective value. Before the actual ML model is called, the references are resolved and the latest values are retrieved from the linked properties.
- **OUT: Response**: The outcome of this operation is a variable that indicates whether a fault has been detected or not. It is defined as a string variable and can therefore contain a text message, but also the text representation of simple data types such as integer or boolean (e.g., `"0"`, `"1"`, `"false"`, or `"true"`).

loadNewModelIntoArchive This operation is used to store a new ML model in the AAS. For this purpose, a SMC is first created in the *FDMArchive* with all the elements that are required according to the definition. The newly uploaded model, which is received as BLOB, is stored in this SMC and the other properties are also filled with their values. The operation *loadNewModelIntoArchive* expects two input variables and provides a response message as output variable:

- **IN: SerializedFaultDetectionModelBlob**: With this parameter, the operation gets the ML model as BLOB. It is actually expected that the uploaded model is base64 encoded as the MIME type of this variable is `text/plain`. This ensures that the model can be stored as a code page independent ASCII string in the MongoDB backend of the AAS and thus avoids encoding problems.
- **IN: Trained**: The date or timestamp when the ML model was trained is specified via this parameter. This is actually done for two reasons: Firstly, this value is used to create a unique ID for the SMC. Secondly, this value is also specified in the archive as an additional property for documentation purposes.
- **OUT: Response**: The return value of this operation is defined as a string. No particular output is expected, but it can be used, for example, to return an ID, `"OK"`, an error message, or some other message to the caller.

updateEnvironment This operation defines the steps required to prepare the new environment containing the ML model defined by the specified ID in the input parameter. The individual steps performed as part of this operation depend heavily on the deployment strategy chosen. Steps that typically happen within this operation are the reading of property values from the AAS or MongoDB, creating the new Docker images, preparing the new Docker containers, or updating the references in the AAS. This operation expects just one input parameter and returns an output variable:

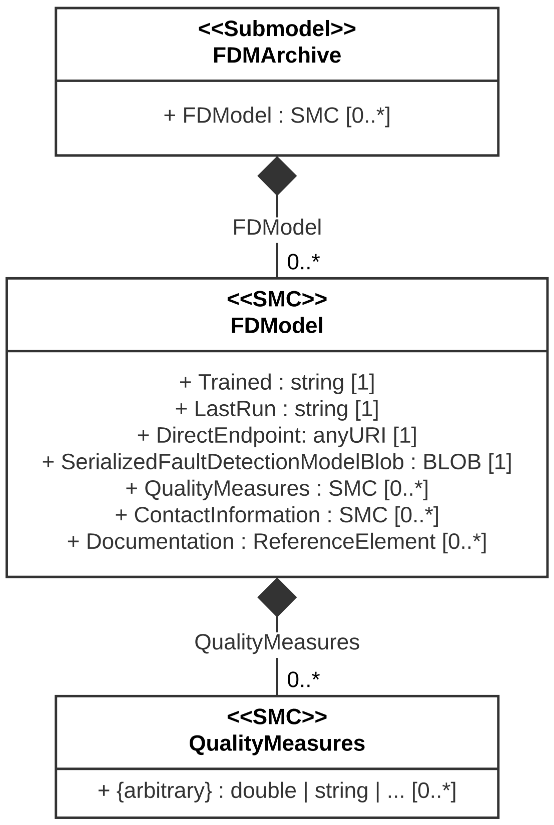
- **IN: ModelToBeActivatedID**: This variable contains the ID of the model stored in the *FDMArchive* submodel that should be loaded into the standby environment for preparation. The expected data type of this variable is string. The provided model ID can be that of the most recently uploaded model as well as that of an older one. Accordingly, all models in the archive can be used for the preparation in the standby environment. This may be necessary, for example, in the event of a rollback to a previous version or for performance comparison reasons.
- **OUT: Response**: The return value of this operation is defined as a string. No particular output is expected, but it can be used, for example, to return a status code, an error message, or some other message to the caller.

swapActiveEnvironment The operation *swapActiveEnvironment* is used to swap the standby environment with the active environment, i.e. the previous standby environment is then the active one and vice versa. Nonetheless, this also depends on the deployment strategy chosen, as not all strategies require two separate environments. For example, one could also use a previously defined Docker image to patch the existing running container(s). This process also updates the corresponding references in the AAS. However, this operation does not require any input parameters, but like the other operations, has a response output variable of type string that can be used to return a status code, an error message, or any other message.

6.2.3 Fault Detection Model Archive Submodel

In this section, the submodel elements of the *FDMArchive* submodel and the therein contained *FDModel* SMCs are described. In Figure 6.4, the components of the archive are shown and it can be seen that the archive consists of an arbitrary number of fault detection models. In turn, any number of quality measures can be assigned to them. The task of the archive is to store all the models that are used both actively or in the standby environment, as well as all those that were previously used or even just uploaded. This is to preserve the history of the used models but also to offer the possibility to switch to any version and use it as active model for the inference.

In the following, the SMCs of *FDModel* and *QualityMeasures* are explained. The *FDModel* element is used to describe one ML model along with its properties and store them together in the AAS. The following list describes the contained submodel elements in more detail:



<<Submodel>>
FDMArchive
+ FDModel : SMC [0..*]
FDModel
0..*
<<SMC>>
FDModel
+ Trained : string [1]
+ LastRun : string [1]
+ DirectEndpoint: anyURI [1]
+ SerializedFaultDetectionModelBlob : BLOB [1]
+ QualityMeasures : SMC [0..*]
+ ContactInformation : SMC [0..*]
+ Documentation : ReferenceElement [0..*]
QualityMeasures
0..*
<<SMC>>
QualityMeasures
+ {arbitrary} : double | string | ... [0..*]

Figure 6.4: UML Diagram of the *FDMArchive* Submodel

- **Trained**: This property contains the date or timestamp when the model was trained. This is provided via an input variable when uploading the model into the AAS. It is used mainly for documentation purposes, but is also part of the ID of the *FDModel* element.

- **LastRun**: The *LastRun* property always contains the timestamp when the model was used the last time for inferences. If it is the active model, then the property is updated regularly on each inference call. On the other hand, if it has never been used, the property is empty or set to a constant value.

- **DirectEndpoint**: This property of type *anyURI* stores the URL of the REST API endpoint for this model, which can be used to access the ML model and use it for inference without declaring it as an active model. The active model also has this property set but differs from that endpoint defined in the *FaultDetectionModel* submodel, which is intended for productive use only. However, it is still necessary that the model is provided in either the active or standby environment. Otherwise, the model will not be able to perform the fault detection and will not be accessible. Therefore, this property is empty or set to a constant value for all models that are neither in the active nor in the standby environment. This endpoint can be used to test the ML model or to determine the quality measures without interfering with the productive use of the active model.

- **SerializedFaultDetectionModelBlob**: This property contains the uploaded ML model as BLOB object. As has already been mentioned at the upload operation, the ML model in this property has to be exported beforehand from any ML tool so that it can be base64 encoded and stored as code page independent ASCII string in the AAS. This is especially important if one designs the AAS with the AASX Package Explorer, for example, to save it as a template, use it as a starting point for an implementation, or to share it with another person. Since currently the supported mime types are limited and thus not all file types can be successfully saved as a BLOB in an aasx package file. For example, serialized ML models cannot be imported into an aasx package directly as *.joblib* files, because the package could not be opened anymore. However, after base64 encoding the serialized ML model, everything works fine.

- **QualityMeasures**: The SMC of quality measures contains any number of quality or performance metrics for the ML model. It does not matter which metrics are chosen as they serve documentation purposes and should help to make decisions on model switches. In addition, the measures also depend on the selected model class, for example, whether it is a classification or a regression model. Each of the properties can basically be of any data type supported by the AAS metamodel, but will most likely be a floating point type.

  Via the direct endpoint element in the *FDModel*, if actually set, the model can be tested against a defined test dataset to determine the defined quality measures and store them in the AAS in the respective SMC. If this is done for two or more models, they can be compared based on their metrics and the model with the best performance can be used as the active model for inference. This can subsequently also serve as a decision support mechanism when it comes to switching from one model to another based on the performance metrics. However, this possibility was modeled only in the theoretical concept, but is for the prototype out of the scope of this work.

- **ContactInformation** and **Documentation**: This submodel contains also the two optional elements *ContactInformation* and *Documentation*. The roles of these two elements are basically identical to those already described in the *Datasheet* submodel.

6.2.4 Remarks on the Developed Integration Concept

During the conception of the integration concept, fields for comments by the user or notes, as well as additional (technical) information fields were deliberately omitted in order to keep it as clear as possible and focus on the essentials. For example, for the ML models in the archive, additional fields could have been added for ML model type, information about the dataset used for training, details about the hyperparameters, and so on. These would be added to the submodel as a SMC, analogous to the quality measures. However, since these are not relevant for the deployment strategy, they were omitted.

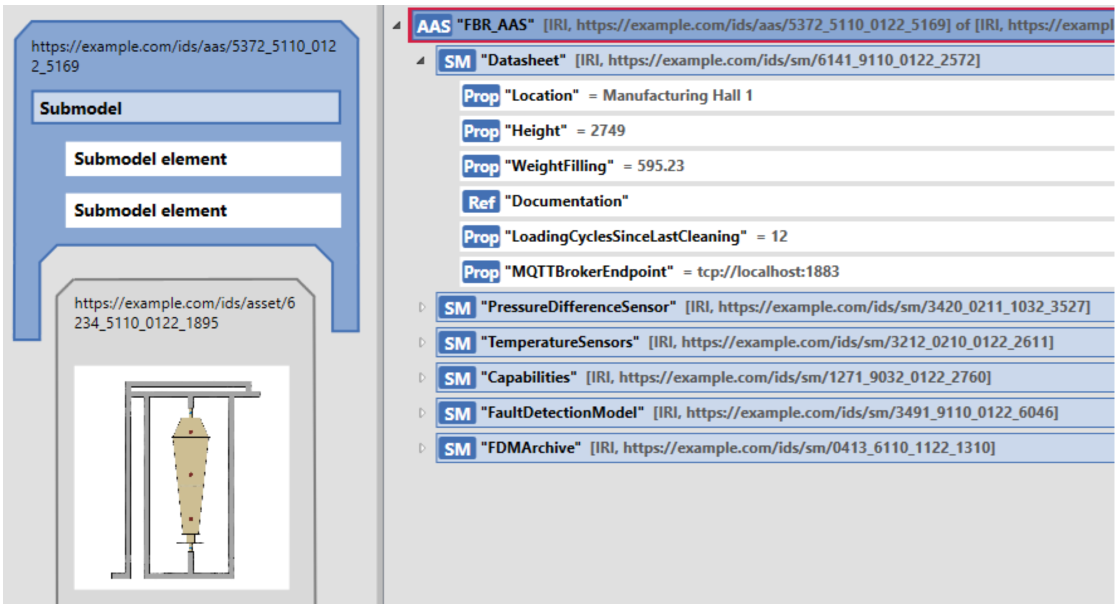
6.3 AAS Integration Concept – Use Case of the Packed-Bed Regenerator

This section deals with the application of the rather generic integration concept described above to the chosen use case – the Packed-Bed Regenerator – which also serves as the basis for the prototype developed later in this thesis. Therefore, a concrete model of an AAS based on the concept described in Section 6.2 will be discussed in the following. The basic structure as well as the mode of operation of a Packed-Bed Regenerator, which provide essential input for the AAS modeled here, have already been described in Section 3.5. In addition, Figure 3.1 illustrates the relevant sensors, which are defined as submodels with corresponding properties in the AAS.

Figure 6.5 shows how such an AAS for a Packed-Bed Regenerator looks like when created in the AASX Package Explorer. The left side of the figure shows a schematic structure of an AAS along with a customized thumbnail. On the right side, the defined AAS is presented with its submodels. There, the first submodel, which is the *Datasheet* submodel together with its submodel elements (properties and reference elements), is visible. The other submodels are collapsed and therefore their submodel elements are not shown. More detailed explanations of the individual submodels are provided below.

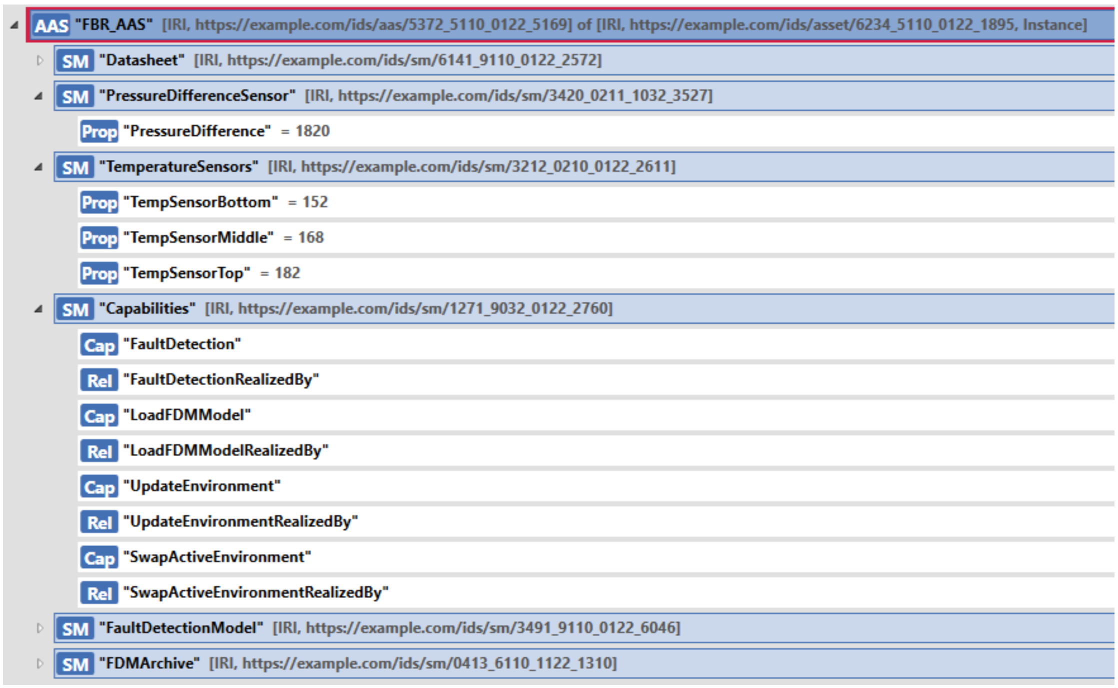
Datasheet In the *Datasheet* submodel, essentially all elements from the concept are represented except for contact information elements. Not only properties such as the location, height, or weight of the Packed-Bed Regenerator were considered, but also the number of charging cycles since the last cleaning or refilling was taken into account. This last property is also relevant for fault detection, as temperature or pressure profiles can change depending on the degree of clogging and introduction of additional particles. Furthermore, a reference to the diploma thesis by Michalka [Mic18] can be found here as a documentation element as well as the endpoint of the MQTT broker, which is needed to send events in case of a fault detected. Since this submodel contains mainly static data, it is hosted on an AAS server.

PressureDifferenceSensor and TemperatureSensors These two submodels are concrete definitions of the rather generic *Sensor* submodel. Figure 6.6 shows that such a submodel can contain not only one measured sensor value, but also several, for example, the temperature values measured in the different layers in the regenerator. Furthermore, several submodels can be defined for different types of sensors to provide a context and a


https://example.com/ids/aas/5372_5110_0122_5169
Submodel
Submodel element
Submodel element
https://example.com/ids/asset/6234_5110_0122_1895

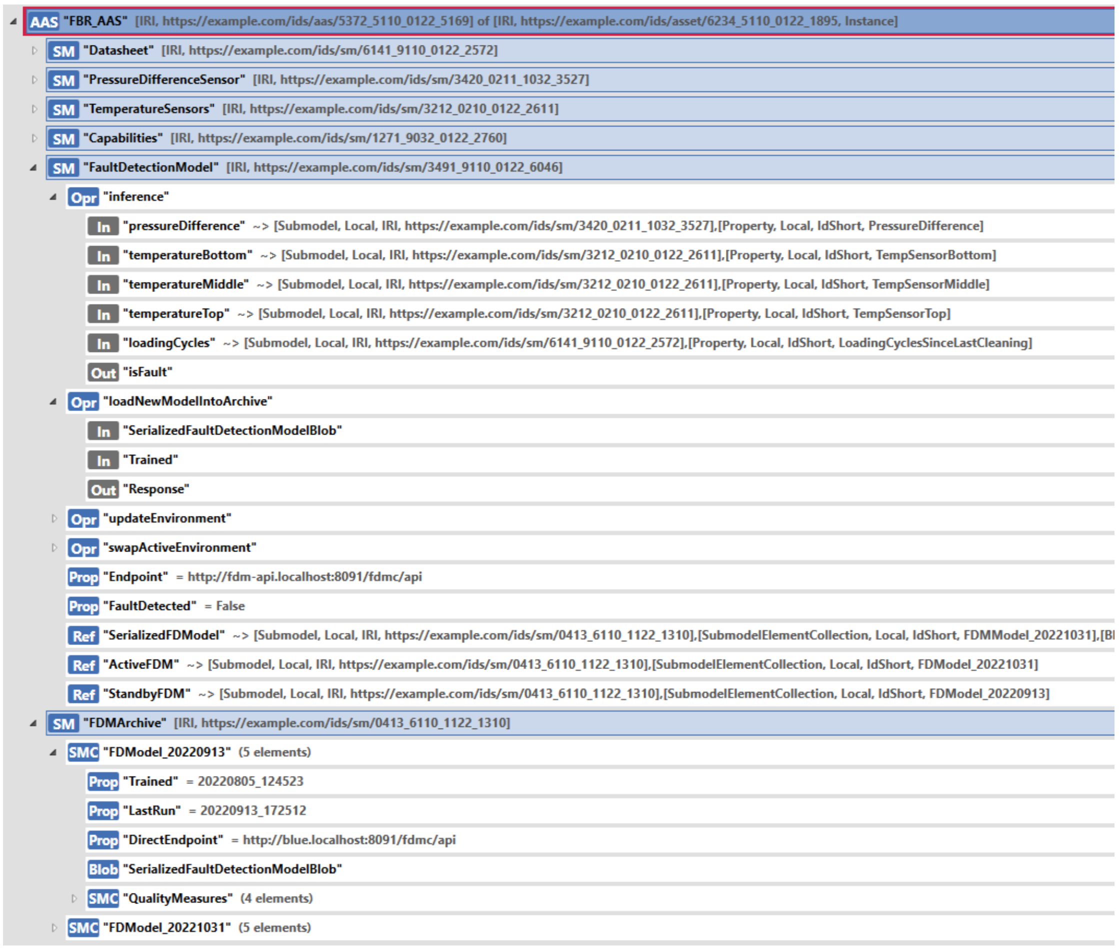
```
AAS "FBR_AAS" [IRI, https://example.com/ids/aas/5372_5110_0122_5169] of [IRI, https://exampl
  SM "Datasheet" [IRI, https://example.com/ids/sm/6141_9110_0122_2572]
    Prop "Location" = Manufacturing Hall 1
    Prop "Height" = 2749
    Prop "WeightFilling" = 595.23
    Ref "Documentation"
    Prop "LoadingCyclesSinceLastCleaning" = 12
    Prop "MQTTBrokerEndpoint" = tcp://localhost:1883
  SM "PressureDifferenceSensor" [IRI, https://example.com/ids/sm/3420_0211_1032_3527]
  SM "TemperatureSensors" [IRI, https://example.com/ids/sm/3212_0210_0122_2611]
  SM "Capabilities" [IRI, https://example.com/ids/sm/1271_9032_0122_2760]
  SM "FaultDetectionModel" [IRI, https://example.com/ids/sm/3491_9110_0122_6046]
  SM "FDMArchive" [IRI, https://example.com/ids/sm/0413_6110_1122_1310]
```

Figure 6.5: AAS Integration Concept for a Simplified Packed-Bed Regenerator



```
AAS "FBR_AAS" [IRI, https://example.com/ids/aas/5372_5110_0122_5169] of [IRI, https://example.com/ids/asset/6234_5110_0122_1895, Instance]
  SM "Datasheet" [IRI, https://example.com/ids/sm/6141_9110_0122_2572]
  SM "PressureDifferenceSensor" [IRI, https://example.com/ids/sm/3420_0211_1032_3527]
    Prop "PressureDifference" = 1820
  SM "TemperatureSensors" [IRI, https://example.com/ids/sm/3212_0210_0122_2611]
    Prop "TempSensorBottom" = 152
    Prop "TempSensorMiddle" = 168
    Prop "TempSensorTop" = 182
  SM "Capabilities" [IRI, https://example.com/ids/sm/1271_9032_0122_2760]
    Cap "FaultDetection"
    Rel "FaultDetectionRealizedBy"
    Cap "LoadFDMModel"
    Rel "LoadFDMModelRealizedBy"
    Cap "UpdateEnvironment"
    Rel "UpdateEnvironmentRealizedBy"
    Cap "SwapActiveEnvironment"
    Rel "SwapActiveEnvironmentRealizedBy"
  SM "FaultDetectionModel" [IRI, https://example.com/ids/sm/3491_9110_0122_6046]
  SM "FDMArchive" [IRI, https://example.com/ids/sm/0413_6110_1122_1310]
```

Figure 6.6: Sensor and Capabilities Submodels of an AAS for a Simplified Packed-Bed Regenerator



```
AAS "FBR_AAS" [IRI, https://example.com/ids/aas/5372_5110_0122_5169] of [IRI, https://example.com/ids/asset/6234_5110_0122_1895, Instance]
  SM "Datasheet" [IRI, https://example.com/ids/sm/6141_9110_0122_2572]
  SM "PressureDifferenceSensor" [IRI, https://example.com/ids/sm/3420_0211_1032_3527]
  SM "TemperatureSensors" [IRI, https://example.com/ids/sm/3212_0210_0122_2611]
  SM "Capabilities" [IRI, https://example.com/ids/sm/1271_9032_0122_2760]
  SM "FaultDetectionModel" [IRI, https://example.com/ids/sm/3491_9110_0122_6046]
    Opr "inference"
      In "pressureDifference" ~> [Submodel, Local, IRI, https://example.com/ids/sm/3420_0211_1032_3527],[Property, Local, IdShort, PressureDifference]
      In "temperatureBottom" ~> [Submodel, Local, IRI, https://example.com/ids/sm/3212_0210_0122_2611],[Property, Local, IdShort, TempSensorBottom]
      In "temperatureMiddle" ~> [Submodel, Local, IRI, https://example.com/ids/sm/3212_0210_0122_2611],[Property, Local, IdShort, TempSensorMiddle]
      In "temperatureTop" ~> [Submodel, Local, IRI, https://example.com/ids/sm/3212_0210_0122_2611],[Property, Local, IdShort, TempSensorTop]
      In "loadingCycles" ~> [Submodel, Local, IRI, https://example.com/ids/sm/6141_9110_0122_2572],[Property, Local, IdShort, LoadingCyclesSinceLastCleaning]
      Out "isFault"
    Opr "loadNewModelIntoArchive"
      In "SerializedFaultDetectionModelBlob"
      In "Trained"
      Out "Response"
    Opr "updateEnvironment"
    Opr "swapActiveEnvironment"
    Prop "Endpoint" = http://fdm-api.localhost:8091/fdmc/api
    Prop "FaultDetected" = False
    Ref "SerializedFDModel" ~> [Submodel, Local, IRI, https://example.com/ids/sm/0413_6110_1122_1310],[SubmodelElementCollection, Local, IdShort, FDMModel_20221031],[Bl
    Ref "ActiveFDM" ~> [Submodel, Local, IRI, https://example.com/ids/sm/0413_6110_1122_1310],[SubmodelElementCollection, Local, IdShort, FDModel_20221031]
    Ref "StandbyFDM" ~> [Submodel, Local, IRI, https://example.com/ids/sm/0413_6110_1122_1310],[SubmodelElementCollection, Local, IdShort, FDModel_20220913]
  SM "FDMArchive" [IRI, https://example.com/ids/sm/0413_6110_1122_1310]
    SMC "FDModel_20220913" (5 elements)
      Prop "Trained" = 20220805_124523
      Prop "LastRun" = 20220913_172512
      Prop "DirectEndpoint" = http://blue.localhost:8091/fdmc/api
      Blob "SerializedFaultDetectionModelBlob"
      SMC "QualityMeasures" (4 elements)
    SMC "FDModel_20221031" (5 elements)
```

Figure 6.7: Fault Detection Model and Archive Submodels of an AAS for a Simplified Packed-Bed Regenerator

certain structure in the AAS. Therefore, the value of the pressure difference sensor and the values of the three temperature sensors are included in separate submodels.

Since these submodels contain values that are updated periodically, it is likely these two submodels will be hosted on a separate node at the edge in close proximity to the machine that provides the values to avoid high latency compared to hosting them on a central AAS server.

Capabilities The submodel *Capabilities* and the submodel elements it contains can be seen as well in Figure 6.6. The purpose of this submodel is essentially to provide a list of the capabilities that this AAS offers and link these capabilities with the respective operations that implement the actual functionality. This submodel is more likely to be located on an AAS server, since it contains mainly static information.

FaultDetectionModel First of all, this submodel contains all four of the defined operations together with the operation variables for input and output. For the operation *inference* it was specified in the corresponding concept that any number of input parameters can be specified. In this case, there are five input variables defined: one for the pressure difference value, three for temperature values, and one for the loading cycles. The concrete type of these input variables is *ReferenceElement*. This means that they directly reference the properties located in the *Sensor* submodels, which can also be seen in Figure 6.7. There, the identifier of the respective submodel and the ID of the property are indicated in gray. Other operations, like the *loadNewModelIntoArchive* operation, require their values to be provided directly.

Further properties in this submodel are the endpoint of the REST API for the ML model for inference, the property that stores whether a fault was detected or not in the last run of inference, and the references to the respective elements in the archive required for preparing and deploying new ML model versions. Whether the standby environment is linked to a model in the archive or remains empty depends on the chosen deployment strategy. The references to the elements in the archive can also be seen via the identifiers in Figure 6.7.

The optional submodel elements for contact information and documentation were not included in the model for this use case. Since it was specified as a requirement in one of the research questions, it is clear that this submodel should be hosted on a node as close as possible to the source of the sensor values in order to provide results as quickly as possible and to keep latency low.

FDMArchive The archive contains all uploaded models that are either active, on standby, recently uploaded, or were already used for inference and are still retained for historization purposes. The archive can contain any number of models, each of which has its own submodel elements defined. An example of this is shown in Figure 6.7. Most of the elements of the concept have been implemented, except for the contact information and reference to documentation. Within the quality measures, any number of properties can be defined, which represent quality criteria for the respective model.

Since the archive mainly contains data that changes only occasionally, this submodel will be located on an AAS server. This offers the possibility that the entire archive can be persisted automatically in the MongoDB.

CHAPTER 7

Identification of Possible Deployment and Switching Strategies

In this chapter, possible deployment strategies for ML models are identified that should enable seamless switching between different model versions without causing service interruptions or downtime. To this end, as with the integration concept for an AAS, there is first a theoretical elaboration of this topic presented, followed by practical applications of these identified deployment strategies, which are also used in the prototype evaluation.

Deployment strategies, also called deployment patterns, provide the ability to control which of the deployed versions of a service actually receive user traffic, since deployment does not necessarily mean that a service is already accessible to users [Cen20]. For a seamless transition from one version to a new version, the objective of having zero downtime is an important prerequisite. There exist three major strategies for performing zero downtime deployments: (1) blue-green deployment, (2) canary deployment, and (3) rolling deployment. The canary deployment strategy is similar to the blue-green deployment, but instead of switching all traffic to the new service at once, as with a blue-green deployment, the former initially redirects only a certain fraction based on factors such as region, user type, user privileges, and so on [Rud20]. However, since such a request segmentation is not helpful for the use case of fault detection, e.g., for a Packed-Bed Regenerator, this strategy will not be discussed further.

7.1 Manual Deployment / Recreate Deployment Pattern

Although the manual deployment strategy or recreate deployment pattern is not considered a deployment strategy in the context described above, it will be mentioned briefly as it

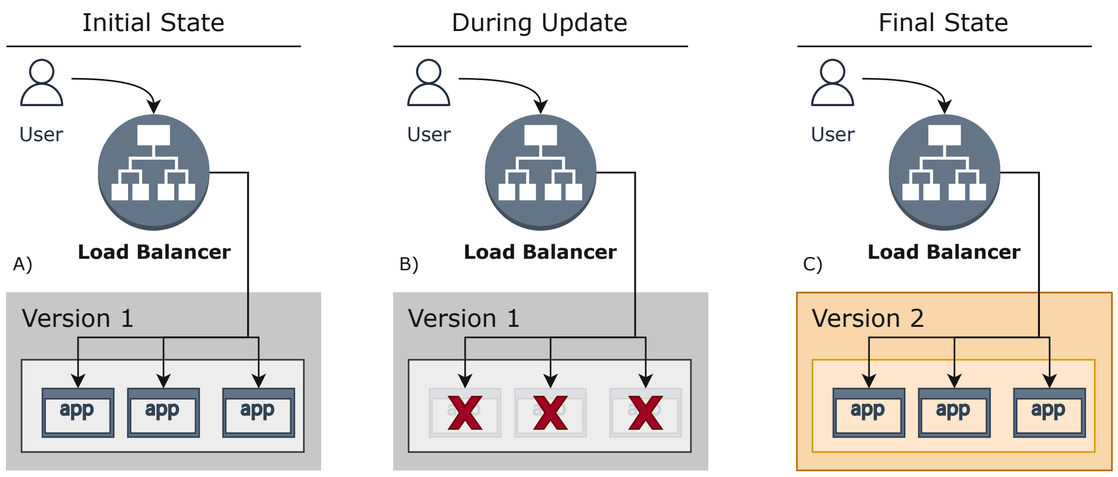


Figure 7.1: Manual Deployment / Recreate Deployment Pattern – Diagram based on [Cen20]

serves as the baseline for evaluating the other deployment strategies. The process for a recreate deployment pattern is graphically illustrated in Figure 7.1. Initially, all instances of the application are stopped, followed by updating the instances, and finally starting all instances with the new version.

This approach provides the advantage of simplicity by eliminating the need to manage multiple versions of an application or multiple environments. Furthermore, it removes challenges related to backward compatibility. However, the major disadvantage with this approach is downtime [Cen20]. Depending on the duration of the update and the time required to stop and restart the instances, the application is unavailable to users. In addition, rollbacks also lead to downtime.

7.2 Blue-Green Deployment Strategy

The basic principle of a blue-green deployment is to replace an old version of a service running in one environment (the blue environment) by a newer version of this service running in another environment (the green environment) without causing service interruptions or downtime [Rud20]. Figure 7.2 shows the basic procedure of this strategy. At the beginning, the current state of the application runs in the blue container and only this one is active at that time (step A). In the course of the update (steps B and C), the new version of the service is first prepared and tested in the separate environment (green) before the switch is performed, based on the test results. Then, all traffic is routed to the services in the green container. Afterwards, the old state can be removed from the blue container to prepare it as staging area for the next update or this state of the application is still retained to perform a possible rollback (step D). The application in the green container is active at this time [Cen20].

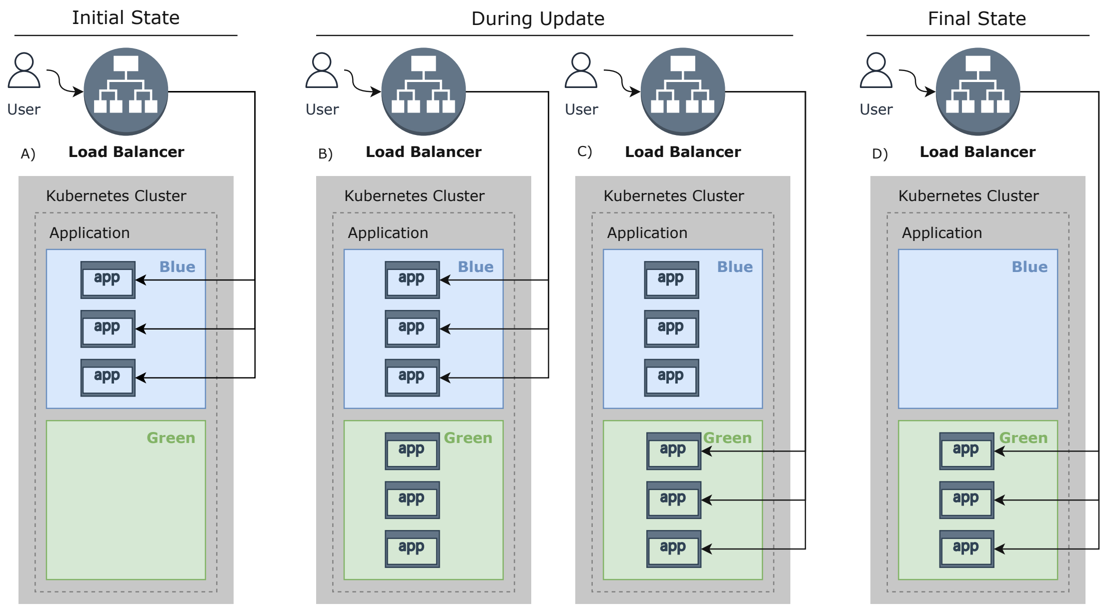


Figure 7.2: Blue-Green Deployment Pattern – Diagram based on [Cen20]

According to the Google Cloud Architecture Center [Cen20], this strategy has the following advantages, but also some considerations that must be taken into account:

Advantages

- **Zero Downtime**: The switch from one version to another can be performed without any downtime.
- **Instant Rollback**: In case of a detected problem with the new version, it is possible to immediately roll back to the previous version if it is still in the standby environment.
- **Clear Separation of Environments**: This strategy can be used to prepare a parallel environment without affecting the resources that are used by the active environment, thus reducing the deployment risks.

Considerations

- **Overhead**: This strategy entails an administrative and cost overhead, as two separate environments need to be managed and maintained.
- **Backward Compatibility**: Different versions of applications can share data stores and resources. It is important to ensure that shared resources are also backward compatible so that it is always possible to switch between versions in the event of a rollback.

- **Appropriate Connection Shutdown**: An appropriate connection shutdown should be considered for remaining open sessions when decommissioning the current version. This ensures that the requests processed by this version can be properly completed or terminated.

7.2.1 A Blue-Green Deployment Strategy using Traefik

This section discusses a blue-green deployment strategy using the Traefik Proxy to provide access to the fault detection service and seamlessly switch between two ML model versions.

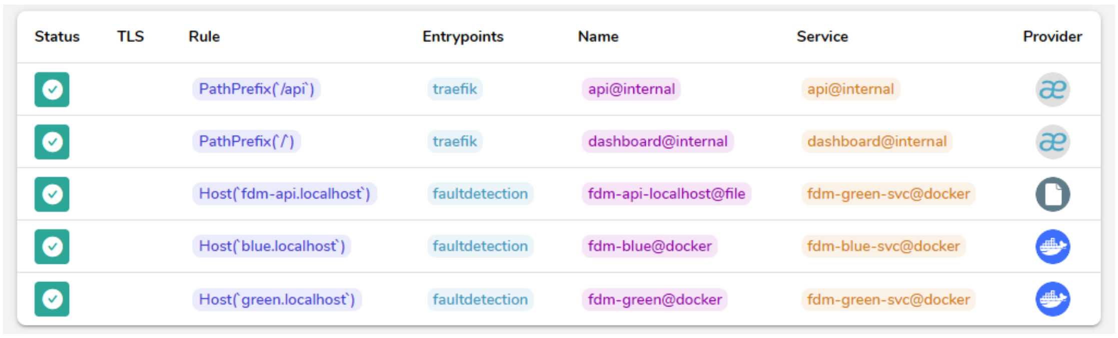
Traefik [1] Traefik is a reverse proxy, or edge router, that processes incoming requests and routes them to the appropriate services that can handle them based on a defined regulatory framework (such as path, host, headers, etc.). Major components in this context are entrypoints, routers, optional middleware, and services. Entrypoints are part of the static configuration of Traefik and basically define the access to Traefik, which in its simplest form is just a port number. Routers are connected to these entrypoints and analyze incoming requests based on rules and optionally transform them using the middleware components. Services are used to define access to the application that will handle the incoming request. They are configured dynamically, which means that changes to the configuration can be made during runtime without any disruption to services or requests. In addition, Traefik offers several ways to define the dynamic configuration, called providers.

For the blue-green deployment strategy, the File Provider and Docker Provider are of interest. Traefik is easily integrated with Docker containers and can be configured using the *yaml* file (used to compose the containers) by defining commands and specifying certain labels for the Docker objects, which can then be used by Traefik for configuration. To implement the strategy, Entrypoints, Routing Rules, Serivces, and Providers must be defined, as shown in Figure 7.3. The first two entries are for the Traefik API and the dashboard, while the others provide access to the fault detection services. The rule essentially specifies the URL for the service, the entrypoint is the defined port, and the service is the name of the Docker container that handles incoming requests on a specific route. The entry in the third line is one that is configured via a configuration file and can dynamically route requests between the blue or green Docker container at runtime. Currently, it is configured to forward requests to the green service.

General Structure and Workflow The basic structure and the components used for a blue-green deployment strategy using Traefik can be seen in Figure 7.4.

User requests for inference can come both from the *FaultDetectionModel* submodel, which invokes the service via the URL *fdm-api.localhost*, and from individual user requests

[1] based on Traefik Proxy Documentation at `https://doc.traefik.io/traefik/` - last accessed: 2023-02-18



Status	TLS	Rule	Entrypoints	Name	Service	Provider
✓		PathPrefix(`/api`)	traefik	api@internal	api@internal	æ
✓		PathPrefix(`/`)	traefik	dashboard@internal	dashboard@internal	æ
✓		Host(`fdm-api.localhost`)	faultdetection	fdm-api-localhost@file	fdm-green-svc@docker	
✓		Host(`blue.localhost`)	faultdetection	fdm-blue@docker	fdm-blue-svc@docker	
✓		Host(`green.localhost`)	faultdetection	fdm-green@docker	fdm-green-svc@docker	

Figure 7.3: Entrypoints, Routing Rules, Serivces, and Providers for a Blue-Green Deployment Strategy visualized the Traefik Dashboard

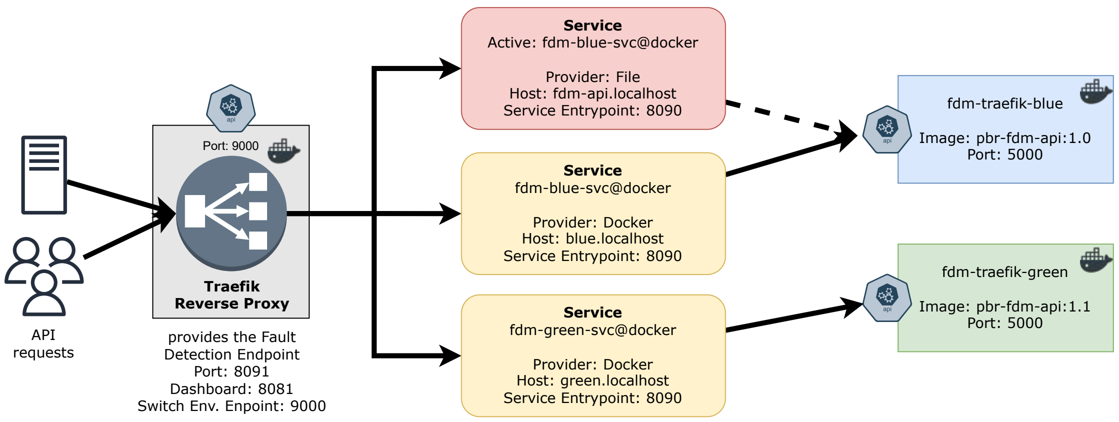


Figure 7.4: General Architecture of a Blue-Green Deployment Strategy using Traefik

that invoke either the red-colored service (the service used productively) or one of the yellow-colored services to invoke the respective model directly in the container. Depending on the used host in the request, the Traefik proxy, which runs as a separate Docker container, processes these requests and forwards them to the respective service, which in turn passes them on to the corresponding model running in one of the containers. It should be noted that the entrypoint on the proxy is defined as port 8091, while the services accept the request on port 8090 and the ML model exposes its interface on port 5000. The reasons for the different port numbers used in different places in this architecture are, on the one hand, that Traefik seemed to have problems with using ports in the range of 5000 directly as entrypoints during implementation and, on the other hand, to make the separation between the individual components visible.

Preparation of the New Environment Before the actual switch between two model versions can be done, the container that is not actively used by the main service must be prepared with the new model version. For this, the following steps are necessary considering the use of the previously described AAS:

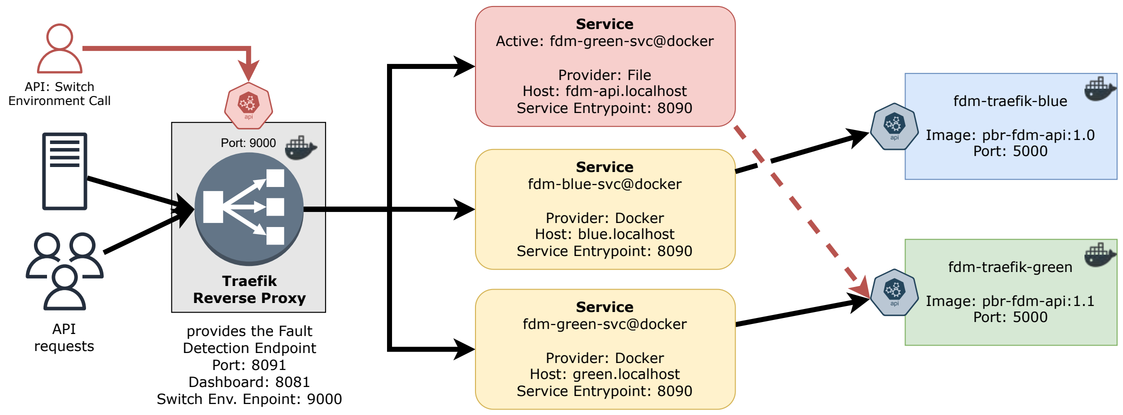


Figure 7.5: Procedure of a Model Switch using Traefik

1. The necessary information for preparing the environment (e.g., currently active environment, reference to standby model, etc.) is read from the AAS and the MongoDB backend.

2. The BLOB object of the model to be loaded is retrieved from the archive and temporarily stored as *joblib* file.

3. The serialized ML model is copied to the standby container and saved as a new Docker image with the minor version number incremented by one.

4. The standby container is stopped and subsequently removed.

5. The standby container is restarted with the newly created image.

6. The temporarily stored *joblib* file as well as the old Docker image are deleted.

7. The corresponding properties of the AAS and in the MongoDB backend are updated with the new values (e.g. direct endpoint, set the new uploaded model as standby reference, etc.).

Switching Between Model Versions Only three steps are necessary for performing the actual switch between the model versions:

1. Retrieve the active environment from the AAS.

2. Modify the dynamic Traefik configuration so that the former standby environment is the active one and vice versa.

3. Update the references for the active and standby environments as well as for the serialized FDM model in the AAS.

Figure 7.5 illustrates the above process and it can be seen (compared to Figure 7.4) that the green Docker container is now used as the active service. The other routes and redirects for the services remain in place and direct access via the corresponding URLs is still possible.

Customized Traefik Docker Image As it can be seen in Figure 7.5, there is also an API highlighted in red on the Traefik Proxy container. However, this is not the standard API provided by Traefik, but a custom API. An additional endpoint was added as there was no other feasible approach to modify the dynamic configuration of the file provider. Alternatively, it would have been possible to use Docker's directory mount feature and use a file, which is synchronized with the one in the container and set that as the configuration file for Traefik. However, this did not work properly on Windows, because there are some issues when files are changed outside the container and, therefore, Traefik did not detect the changes in the configuration file.

Listing 7.1 shows this configuration file. For the switch between the two environments, all what needs to be done in this dynamic configuration is to change the last line to either *fdm-blue-svc@docker* or *fdm-green-svc@docker*.

Listing 7.1: File Provider for fdm-api Service

```
http:
routers:
fdm-api-localhost:
entrypoints:
- "faultdetection"
rule: Host(‘fdm-api.localhost‘)
service: "fdm-green-svc@docker"
```

This API endpoint proposed above has one route that listens on port 9000 and accepts either *blue* or *green* as payload of the request and sets the respective service in the configuration file and saves the changes. Traefik detects these changes as it constantly monitors this file for changes and adjusts the registered service accordingly.

In order to add this functionality to the ready-to-use Traefik Docker image, a new Docker image was created using the Traefik Docker image as a base and adding the functionality implemented in Python with its dependencies. In addition, the Traefik script *entrypoint.sh*, which normally processes the Traefik commands defined in the *yaml* file, had to be adapted to provide the additional functionality as well.

7.2.2 A Blue-Green Deployment Strategy using Kubernetes

In this section, a blue-green deployment strategy is discussed utilizing the Kubernetes container orchestration platform to provide not only access to the fault detection service, but also allow switching between two ML model versions without downtime.

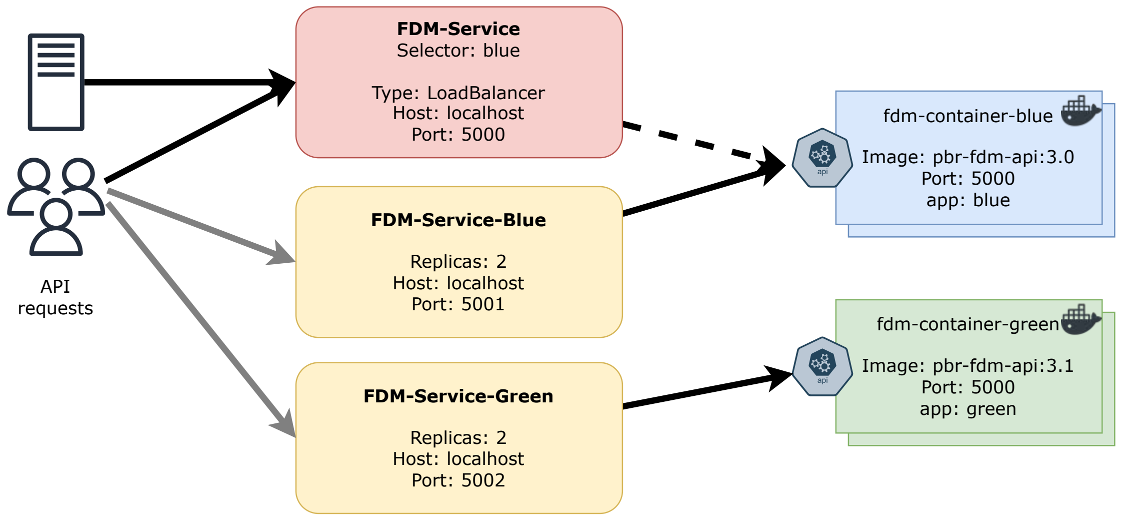


Figure 7.6: General Architecture of a Blue-Green Deployment Strategy using Kubernetes

Kubernetes [2] Kubernetes is a well known and widely used open source container orchestration platform. It allows automating the deployment of containerized applications, but also scaling and managing them. When using Kubernetes for orchestration, it is called a cluster. The main components of such a Kubernetes cluster are the worker nodes on which the containerized applications run. Within these worker nodes, the so-called pods are hosted. Pods are the smallest deployable units that can be managed and refer to a group of one or more containers that share common resources (such as storage or networking). The most well-known runtime used in pods is Docker.

Services are another important component. In Kubernetes, a service is a conceptual representation that identifies a group of pods running within the cluster, each serving the same purpose or functionality. They are mainly used for enabling network access, load balancing, scaling, and traffic splitting for purposes such as A/B testing, canary deployments, or blue-green deployments.

The control plane manages the cluster with the worker nodes and pods. For this purpose, various components run within this control plane for various administrative tasks. A deployment in Kubernetes allows pods and replica sets to be updated declaratively. By describing a desired state in the deployment, the deployment controller can modify the actual state of the deployment to match the desired state.

General Structure and Workflow The basic structure and components used for a blue-green deployment strategy utilizing Kubernetes for load balancing, scaling, and traffic splitting can be seen in Figure 7.6.

[2]based on Kubernetes Documentation at `https://kubernetes.io/docs/home/` - last accessed: 2023-02-18

Similar to the approach with Traefik, there are also three services here, each serving a slightly different purpose. The *FDM-Service* is the one used productively. In the case of Figure 7.6, it refers to the pods with the tag *blue*. The other two services are those for providing direct access to the models in the pods. When defining these services, it was specified that there are two replicas of a Docker image behind each service to illustrate this aspect of Kubernetes as well.

User requests for inference can come both from the *FaultDetectionModel* submodel, which invokes the service via the URL *localhost:5000*, and from individual user requests that invoke either the *FDM-Service* or one of the other two services to access the respective model directly. The last two services have the ports 5001 and 5002 assigned. All the management, routing, forwarding of requests, internal IP and port management is all handled by Kubernetes. Which of the pods processes a specific request is also under the responsibility of Kubernetes.

Preparation of the new Environment In preparation for the model switch between the two versions, the environment that is not currently being used by the *FDM-Service* actively must be updated with the new model version first. This requires the following steps, taking into account the use of the AAS mentioned above:

1. The necessary information for preparing the environment (e.g., currently active environment, reference to standby model, etc.) is read from the AAS and the MongoDB backend.
2. The BLOB object of the model to be loaded is retrieved from the archive and temporarily stored as *joblib* file.
3. The serialized ML model is copied to a base template container, based on which a new Docker image is created with the minor version number incremented by one.
4. The pods that are not actively used are patched by means of a deployment. This deployment defines the newly created image together with some settings as the desired state and lets the controller perform the update.
5. It is waited until all pods with the old image are stopped.
6. The temporarily stored *joblib* file as well as the old Docker image are deleted.
7. The corresponding properties of the AAS and in the MongoDB backend are updated with the new values (e.g., direct endpoint, set the new uploaded model as standby reference, etc.).

Switching Between Model Versions Only three steps are required for the actual switch between the model versions:

1. Reading the active environment from the AAS.

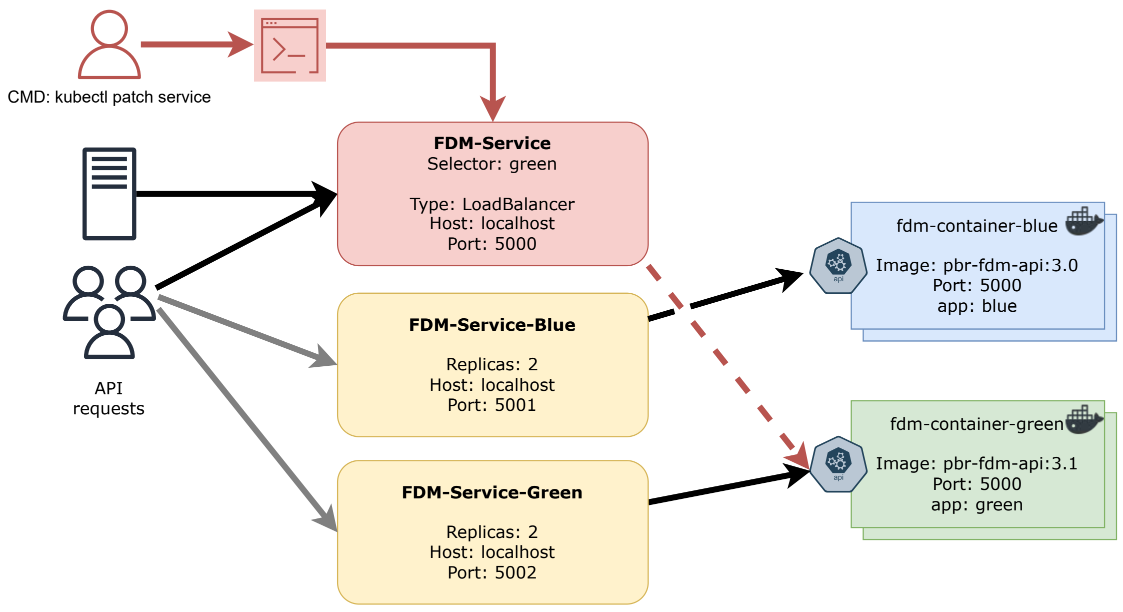


Figure 7.7: Procedure of a Model Switch using Kubernetes

2. Update the *FDM-Service* by changing the selector to the app tag that was not active before.

3. Update the references for the active and standby environments as well as for the serialized FDM model in the AAS.

Figure 7.7 illustrates the steps described above. It can be seen (compared to Figure 7.6) that the selector in the *FDM service* is changed to *green* and requests are now forwarded to these pods. The other routes and redirects for the services remain in place and direct access via the corresponding URLs is still possible.

In contrast to changing the configuration with Traefik in order to switch between models, in this case, it is not done via an API call but via a Kubernetes command on the command line. The command, exemplified for Windows Powershell, is:

```
kubectl patch service fdm-service -p
"{\"spec\":{\"selector\":{\"app\": \"green\"}}}"
```

This simply tells Kubernetes that incoming requests should now be forwarded to the other environment. All further configurations and changes are made by Kubernetes. Yet again, there is no downtime when switching between model versions, as both environments with their respective pods are already up and running and only the selector needs to be adjusted.

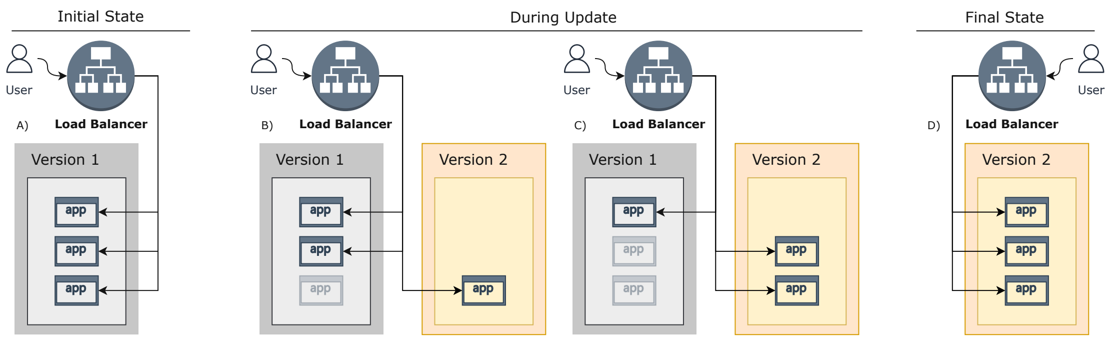


Figure 7.8: Rolling Update Deployment Pattern – Diagram based on [Cen20]

7.3 Rolling Update Deployment Strategy

In a rolling deployment strategy, updates are rolled out incrementally. Hence, only a subset of the application instances is updated to a new version simultaneously. This number of nodes is called window size and its value depends primarily on the size of the cluster. Figure 7.8 illustrates a rolling deployment strategy with a window size of 1, which implies that only one application is updated at a time [Cen20]. This means that during an update (steps B and C), some of the nodes are already running the new version of an application, others are in the process of being patched, and some are still running the old version. Therefore, rolling deployments can be used effectively when there is always spare capacity for the roll out of a new version [Rud20].

The Google Cloud Architecture Center [Cen20] also identified benefits and considerations for this strategy:

Advantages

- **Zero Downtime**: Since only a certain number of instances are updated at a time, the remaining ones are available for requests.
- **Limited Deployment Risk**: As updates are rolled out gradually, only some users are affected by any issues of the new application version.

Considerations

- **Gradual Rollback**: In case of a detected problem with the new version, the rollback has to be performed incrementally as well, which is slower than with other strategies.
- **Backward Compatibility**: Since old and new application versions coexist during deployment and it is not certain to which version a request will be forwarded, it is important that the new version is backward compatible, meaning that it can read data written by the other version, for example.

- **Advanced Session Handling**: If certain session conditions must be met by the application, it might be helpful that the sessions can be decoupled from the underlying resources.

7.3.1 Docker Swarm – A Rolling Deployment Strategy

Docker Swarm [3] Docker Swarm is an orchestration tool that is embedded in the Docker engine and enables the management of a cluster of Docker nodes in swarm mode. The nodes in a swarm can act as either managers or workers, or perform both roles simultaneously. Worker nodes receive tasks from manager nodes and execute them as running containers managed by the swarm manager. Tasks are essentially the running containers that are part of a swarm service. The service, on the other hand, is basically the definition of a task to be executed. Within a service, the desired state is defined in terms of the Docker image used, the number of replicas, networks, storage resources, ports, etc. The swarm manager aims to achieve the desired state of the service, ensuring that the specified number of replicas is running, and taking action to schedule tasks on other nodes if a worker node fails unexpectedly.

Swarm services offer the advantage over standalone containers that configuration changes can be made without requiring service restarts. This is achieved by Docker stopping the containers with outdated configuration and creating new containers with the desired configuration.

General Structure and Workflow The basic architecture and components used for a rolling deployment strategy using Docker Swarm for orchestrating task replicas on different nodes are shown in Figure 7.9.

The swarm service definition specifies that there should be four replicas of the task and that the container should provide a port mapping of 8095 (external) to 5000 (internal). Additionally, an update delay of 30 seconds has been configured, which staggers the updates of individual replicas during the rolling restart process. This helps to ensure a smooth and gradual update process for the service.

User requests for inference can come both from the *FaultDetectionModel* submodel and from individual user requests. In contrast to the strategies mentioned above, there are no separate endpoints for productive access to the currently active model or direct access to the models themselves. In addition, there is neither a standby environment where models can be prepared for switching, nor the possibility to quickly go back to the previous version. The decision to which node an incoming request is forwarded is up to the internal Docker Ingress Load Balancer.

Preparation of the new Environment Compared to the other two deployment strategies mentioned, this rolling deployment strategy does not involve preparing the

[3]based on Docker Swarm Documentation at `https://docs.docker.com/engine/swarm/` - last accessed: 2023-02-19

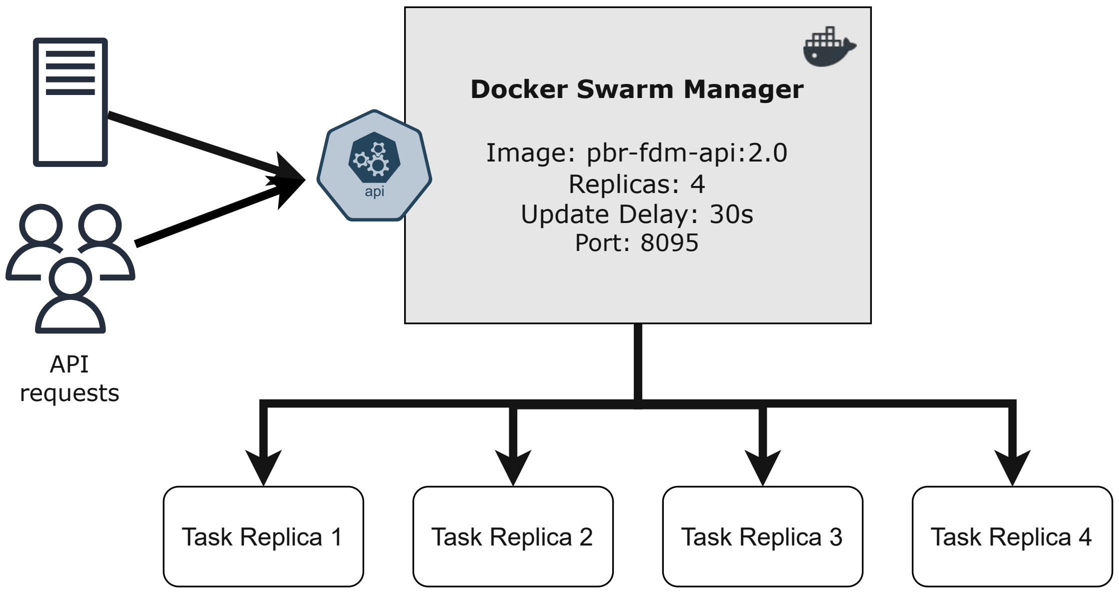


Figure 7.9: General Architecture of a Rolling Deployment Strategy using Docker Swarm

model in a separate standby environment. Nevertheless, some preparation steps are still performed, taking into account the use of the AAS mentioned above:

1. The BLOB object of the model to be loaded is retrieved from the archive and temporarily stored as *joblib* file.
2. The serialized ML model is copied to a base template container, based on which a new Docker image is created with the minor version number incremented by one.
3. The temporarily stored *joblib* file is deleted.
4. The reference to the model to be loaded is stored in the standby reference in the *FaultDetectionModel* submodel, but only for internal purposes, since this information is then needed for the actual switch. This property is also saved in the MongoDB backend.

Switching Between Model Versions Since only the Docker image with the new model was prepared during the preparations for the model switch, more extensive as well as longer lasting steps are now required to perform the actual switch between the models. To accomplish this, the following steps need to be performed:

1. Retrieving the relevant information from the AAS.
2. Update the Docker swarm service with the previously created Docker image.

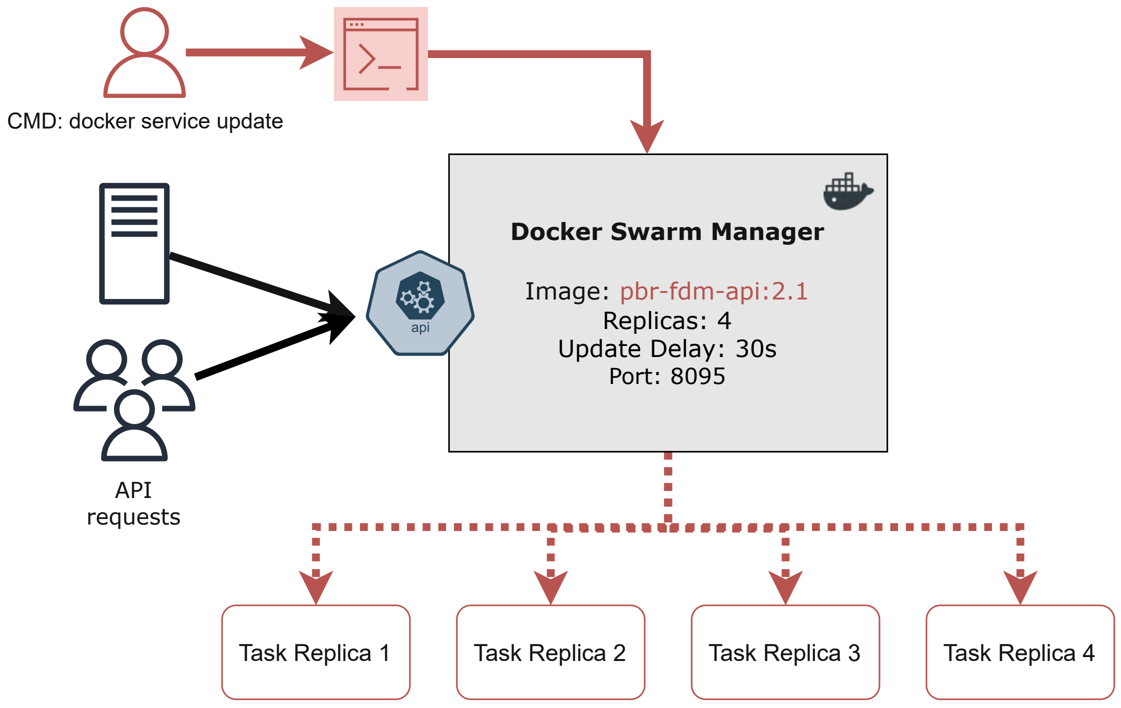


Figure 7.10: Procedure of a Model Switch using Docker Swarm

3. Delete the old Docker image.
4. Update the references for the active and standby environments as well as for the serialized FDM model in the AAS.

Figure 7.10 illustrates the steps described above. It can be seen (compared to Figure 7.9) that, except the Docker image version, nothing has changed. The way of accessing the service, port numbers and routes remain completely unchanged.

As with the Kubernetes switching procedure, the switch is triggered via a command on the command line. This is also indicated by the command line icon highlighted in red in Figure 7.10. The corresponding command may look like the following: `docker service update -image pbr-fdm-api:2.1 fdm-swarm`

Since nothing else was specified in the definition of the service, except for the update delay, the default behavior is used for performing the update. This essentially involves the following steps:

1. Stop the task replica.
2. Perform the update for the stopped task and restart it afterwards.

3. After an updated task returns the state *RUNNING*, the scheduler can continue with the next update after waiting for the update delay time.

4. However, if *FAILED* is returned during the update, the update will be paused by default. This can also be changed using the `-update-failure-action` option (possible values: pause, continue, or rollback).

By default, one task at a time is updated if no specific instructions are provided. Furthermore, the update order is *stop-first* by default, but can be configured using the `-update-order` option. This means that the outdated container is stopped first before the updated one is started. However, it is possible to change the update order to *start-first*, where the new container is started before the old one is decommissioned. In this use case, having four replicas of the service, there was no need to change the update order as three tasks were always available to handle requests.

Unfortunately, the Docker documentation on rolling updates[4] does not provide a clear statement on how incoming requests are handled during such an update process, for example, whether requests are forwarded only to the updated tasks or if they are distributed equally between old and updated tasks. However, some of the statements on this website imply that both old and new containers handle incoming requests. On the other hand, the internal ingress load balancer might be aware of the status of each replica and direct traffic only to those replicas that have been updated with the new image, while avoiding those that are still running an old image.

[4]`https://docs.docker.com/engine/swarm/swarm-tutorial/rolling-update/`

CHAPTER 8

Evaluation and Results

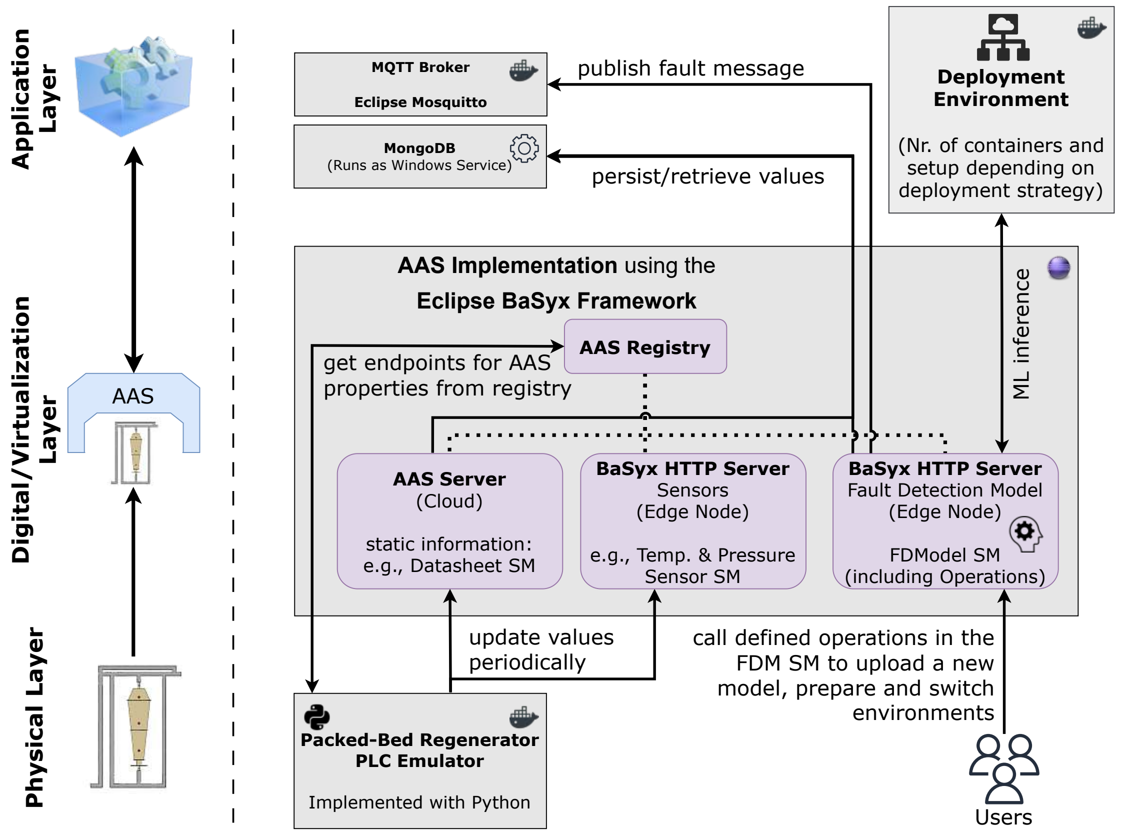
This chapter presents the evaluation results based on the implemented prototype. For this purpose, the use case that serves as the basis for implementing the prototype is explained first. Afterwards, the results of the criteria-based evaluation are presented, followed by the time measurements of the technical experiments as the second part of the evaluation. To this end, Section 2.2 elaborates on the methodological approach for conducting the evaluation of the artifact for both criteria-based and prototype evaluation along with the technical experiments.

8.1 Use Case Description

In Section 3.5, general aspects of the Packed-Bed Regenerator have already been discussed. Based on this explanation, a simplified use case is created, which serves as the basis for the comparative evaluation of different deployment strategies. Therefore, this section deals with the explanation of the created use case, which is implemented as a prototype.

First, an overview of the system architecture for fault detection in the use case of a Packed-Bed Regenerator is given. This includes both the connection and the interaction of the necessary components required for the proper functioning of this use case. Therefore, it builds upon the AAS integration concept described in Section 6.3 and links it with the components necessary for the practical implementation.

The basic idea of this use case can be described as follows: The Packed-Bed Regenerator is equipped with three temperature sensors on three different levels and a pressure difference sensor. However, since there is no access to a physical regenerator, the measured sensor values are simulated by an emulator according to a simple pattern. Thereafter, this emulator updates the simulated values directly in the properties in the AAS at regular intervals via the REST API provided by the AAS as an actual PLC would do. In addition



Application Layer
Digital/Virtualization Layer
Physical Layer
AAS
MQTT Broker
Eclipse Mosquitto
MongoDB
(Runs as Windows Service)
publish fault message
persist/retrieve values
Deployment Environment
(Nr. of containers and setup depending on deployment strategy)
AAS Implementation using the Eclipse BaSyx Framework
AAS Registry
get endpoints for AAS properties from registry
ML inference
AAS Server
(Cloud)
static information: e.g., Datasheet SM
BaSyx HTTP Server
Sensors
(Edge Node)
e.g., Temp. & Pressure Sensor SM
BaSyx HTTP Server
Fault Detection Model
(Edge Node)
FDModel SM
(including Operations)
update values periodically
call defined operations in the FDM SM to upload a new model, prepare and switch environments
Packed-Bed Regenerator PLC Emulator
Implemented with Python
Users

Figure 8.1: Overview of the System Architecture of the Described Use Case

to the sensor values mentioned above, the number of charging cycles is also simulated and updated accordingly in the AAS.

Among other things, the fault detection operation is defined in a dedicated submodel in the AAS and provides the operations described above. If a potential fault is detected by the ML model, an event is published as an MQTT message to the MQTT broker with the specified topic. This fault detection operation is executed regularly according to the predefined interval, accessing the sensor values contained in the respective submodels and transmitting them to the fault detection ML model. Furthermore, the values that are not constantly changing are stored directly in corresponding collections in the MongoDB. The operations described above for uploading the model, preparing the environment and switching between the environments are invoked via the REST interface provided by the AAS. How exactly the individual operations execute their functionality and which software components are required depends primarily on the selected deployment strategy.

Figure 8.1 provides an overview of this system architecture, which can be organized into three layers. The bottom layer is the physical layer that mainly contains the Packed-Bed

Regenerator, which is emulated in this use case. The middle layer is the digital or virtualization layer. This layer includes the AAS defined for the Packed-Bed Regenerator on the physical layer. On top is the application layer that contains the ML models running in the deployment environment as well as further applications required for persisting AAS properties and publishing fault events.

In the left area of Figure 8.1, a more general representation of this architecture is presented, illustrating the three major components: (1) the “physical” Packed-Bed Regenerator, (2) the AAS defined for this regenerator, and (3) other software components needed for this use case. Towards the right, the AAS implementation is broken down in more detail into the registry as well as the individual servers and focuses primarily on the technical details as they were implemented in the prototype. Furthermore, this representation depicts how these components are implemented as a whole and how they are interconnected. In the following sections, the main details of the individual components are presented.

8.1.1 Packed-Bed Regenerator Emulator

The emulator is implemented in Python and runs in its own Docker container. The only information needed in this emulator is the URL of the AAS registry and the IDs of the submodels as well as the IDs of the properties. Using the registry and the provided submodel IDs, the emulator queries the actual endpoints of the submodels at startup, i.e., the URL of the server where they are actually hosted. This ensures that the emulator is independent of the actual distribution of the submodels, namely where they are actually hosted.

At regular intervals, which in the prototype implementation is one second, a random number generator is used to generate new “measured” sensor values within a predefined interval of values that depend on the sensor type. In addition, the number of loading cycles is incremented by one for every tenth update. These values are updated directly in the respective submodels in the AAS using the standardized REST interface of the BaSyx framework.

8.1.2 AAS Implementation using Eclipse BaSyx™

The AAS integration concept described above is implemented using the Eclipse BaSyx platform. BaSyx provides ready-to-use components for hosting, accessing, manipulating, and adding functionality to an AAS. This includes amongst others a registry component, an AAS server component, simple HTTP servers for hosting submodels, but also integrated features for communicating with other components like an MQTT broker or MongoDB.

First, the general structure of the AAS for a Packed-Bed Regenerator was modeled using the AASX Package Explorer and the resulting AAS was saved as an *.aasx* package. This package file was subsequently loaded and processed in BaSyx. The individual submodels contained in this AAS are registered at the AAS registry and hosted on a server depending on the submodel type. Submodels like the *Datasheet* or the *FDMArchive* are hosted on the AAS server with the MongoDB persistence backend, while the sensor submodels are hosted on their on BaSyx HTTP server. Likewise, the *FaultDetectionModel* is hosted on its own BaSyx HTTP server.

Operations The operations required to perform fault detection and to enable dynamic deployment of new ML models are contained in the submodel *FaultDetectionModel*. In Figure 8.1, these operations are indicated by the head icon containing a cogwheel.

Since it is not possible to store operations permanently in a persistence backend, they must be set each time the server is started. In practice, however, the server will not be restarted that often. Users can invoke the provided operations via the standardized REST interface provided by BaSyx.

However, the operations can be also invoked directly within BaSyx. This is done in the prototype with the inference operation, which is called automatically at regular intervals after the server has been started and the operation functionalities have been defined. Hence, this operation is completely independent of any external input, since it always receives the most recent values via the defined references to the sensor values and supplies them to the fault detection model for inference, retrieves the result and, in case of a fault, sets the property in the AAS and publishes the corresponding MQTT message.

Dynamic Deployment of New Models The actual setup required to enable dynamic deployments and switching between models does not play a significant role in the description of the use case. The AAS concept and thus also the implementation were deliberately kept as generic as possible and independent of any specific strategy. Of course, the aspects of the identified deployment strategies from Chapter 7 had to be taken into account when implementing the concrete operations in the submodel.

8.1.3 Other Software Components

For this use case, in particular for the implemented prototype, the MQTT eventing functionality provided by BaSyx is not used but a custom messaging service was defined. The reasons for this have already been outlined in Chapter 6. As MQTT broker, the open source message broker Eclipse Mosquitto was used, since it is a lightweight implementation of the MQTT protocol and can also be run on low-power nodes. For the prototype, the officially provided Docker image of Eclipse Mosquitto was used with the default configuration.

An essential component to store property values permanently is MongoDB and the interface to it is already integrated in the BaSyx framework as a persistence backend. The implications of using this backend in combination with the different server types provided by BaSyx have already been discussed in Chapter 6.

In this use case, the AAS server can automatically store and also retrieve its property values in the MongoDB collections, while for the *FaultDetectionModel* submodel, this functionality had to be implemented from scratch for certain properties that are essential

			Deployment Strategies		
	Requirement	**Baseline**	**Traefik**	**Kubernetes**	**Swarm**
Group 1	R-5.1.a - Containerization	+	+	+	+
	R-5.1.b - Microservices	+	+	+	+
	R-5.1.c - Deployment Strat.	−	+	+	+
Group 2	R-5.2.a - DSU/Timing Const.	−	+	+	+
	R-5.2.b - Fallback Strategy	−	+	+	○
	R-5.2.c - Quality Assurance	−	+	+	−
	R-5.2.d - Infrastructure Req.	+	+	+	+

Table 8.1: Fulfillment of the Defined Deployment and Dynamic Update Requirements per Strategy

for preparing and interchanging ML models. In the prototype implementation, MongoDB was deployed as a Windows service using the default configuration.

8.2 Fulfillment of the Requirements

This section outlines to what extent each of the identified deployment strategies from Chapter 7 meet the defined criteria and requirements of Chapter 5. In addition, the integration concept from Chapter 6 is also evaluated against these aspects. Likewise, the baseline approach was assessed in this regard.

Table 8.1 and Table 8.2 summarize this evaluation in terms of how well each aspect is met or not by each strategy. To this end, three different levels of fulfillment are defined. Thus, the following coding is used: (+) completely fulfilled, (○) partially fulfilled, and (−) not fulfilled. Moreover, the assignment of the criteria to the groups is explained once more: The first group relates to deployment concepts, the second group to dynamic updates of software components, and the third group to AAS-related aspects. In the following, brief remarks on the respective concepts and strategies are presented.

Baseline Approach The baseline approach uses a container to encapsulate the model. Given that the models are identical, the REST interface is also the same as for the other strategies. However, the baseline approach, which follows the recreate deployment pattern, does not have a standby environment and therefore does not support the ability to perform A/B tests for quality assurance, nor does it allow for an advanced deployment strategy.

As for the dynamic update methods, the baseline approach only considers the infrastructure requirements imposed by the AAS concept, all other requirements are not met. Therefore, with this approach, new models cannot be deployed without downtime, there

	Requirement	AAS Concept
Group 3	R-5.3.a - Generic Integration Approach	+
	R-5.3.b - Deployment Strategy Independence	+
	R-5.3.c - Fault Detection Submodel forms Logical Unit	+
	R-5.3.d - Possibility to Archive Models	+
	R-5.3.e - Quality Criteria based Decision Making	+

Table 8.2: Fulfillment of the Defined Integration Concept Requirements

is no possibility for rollbacks, and there is no possibility to test the model to ensure its quality before it is set as active.

Deployment Strategies All three strategies rely on containerization using Docker containers for deployment. The model in the containers is made externally available via a REST API for ML inference. In terms of well-defined deployment strategies, all three strategies follow either a blue-green deployment strategy or a rolling deployment strategy, meaning that this requirement is fully met for all three. Specifically, a rolling deployment approach is not feasible with the Traefik approach presented here. However, it would be possible with Kubernetes, but the configuration would have to be adjusted. With Docker Swarm, as a concrete implementation of a rolling update approach, this is obviously possible. In return, no blue-green deployment is possible with Docker Swarm, but it is feasible with Kubernetes and Traefik.

Regarding the aspects of dynamic update methods, Traefik and Kubernetes meet all five requirements listed here. Together with the rolling update approach, they achieve zero downtime and thus meet the DSU criteria and the timing constraint. While Traefik and Kubernetes fully support a fallback strategy with backward compatibility, Docker Swarm has limited functionality to rollback to a previous version, primarily because no standby environment is available. This is also the reason why Docker Swarm lacks quality assurance capabilities, as no tests can be performed until the model is set as active. Nevertheless, all three deployment strategies follow the infrastructure requirements imposed by the AAS concept.

With regard to the requirement to enable A/B testing, this is technically possible with the blue-green deployment strategies (Traefik and Kubernetes), since at least two separate environments exist here. With Docker Swarm, on the other hand, such an A/B test is not possible, since a second separate environment is missing here.

AAS Concept Although the first two groups of requirements are not relevant for the AAS concept, several criteria are still met but are not explicitly stated in Table 8.1 because they were actually defined for the deployment strategies. Of the first group, all criteria are fulfilled except for R-5.1.a, the containerization. However, this requirement does not play a role, since this is exclusively about the deployment of ML models. The

	upl. to archive	image created	cont. stopped	cont. run.	clean up	total	service n.a.
μ	142	2,093	18,135	7,970	699	29,039	26,105
σ	43	740	8,051	1,802	233	9,273	8,851

Table 8.3: Manual Deployment Times (Duration in ms) – Grouped By Step

AAS concept supports a web API and the use of a microservice architecture. Furthermore, different deployment strategies are supported and it is also possible to define a standby environment to perform tests.

Regarding the dynamic update requirements of the second group, the concept fulfills both the requirement for a fallback strategy and the possibility for quality assurance via the possible standby environment. In addition, the consideration of infrastructure requirements is also complied with. However, the first two criteria of the second group deal exclusively with dynamic update approaches and are thus not relevant.

For the third group, listed in detail in Table 8.2, all defined requirements are completely fulfilled. Therefore, the developed concept is a generic integration approach that is not specifically tailored to any particular deployment strategy and, furthermore, encapsulates all elements related to fault detection in one submodel. In addition, there is also the possibility to archive models and store quality metrics for each fault detection model.

8.3 Technical Experiment Results

This section presents the results obtained from performing the test runs according to the procedure described in Section 2.2.3.

8.3.1 Tabular Representation of the Measured Execution Times

Table 8.3 shows the measured execution times for the baseline approach, a manual deployment following the recreate deployment pattern as described in Section 7.1. Additionally, the average total time as well as the average time in which the service is not available are given in the last column. All times are in milliseconds (ms) and are rounded to whole numbers.

Although the total time for a manual deployment does not seem as long compared to some of the other times in Table 8.4, the service for fault detection is unavailable most of the time, while it is always available for the approaches presented below. The *service n.a.* column contains the time when the service is unavailable and is basically the sum of the values of *cont. stopped* (old container stopped) and *cont. run.* (new container up and running). This means that almost 90% of the total time is needed for stopping the old container and starting the new one.

Table 8.4 summarizes the measured execution times of the four operations defined in the *FaultDetectionModel* submodel (see further Section 6.2.2), including the sub-steps, for the three identified deployment strategies. If no sub-steps are defined for an operation,

Operation	**Step**		**Traefik**	**Kubernetes**	**Swarm**
inference	**total**	μ	156	169	145
		σ	55	62	39
model upload	**total**	μ	125	144	127
		σ	46	50	43
update environment	retr. aas prop.	μ	125	142	30
		σ	42	48	17
	BLOB creation	μ	28	30	34
		σ	16	16	15
	new img. created	μ	1,013	782	708
		σ	285	82	55
	new env. prep.	μ	14,895	61,875	0.05
		σ	4,395	11,415	0.22
	aas prop. upd.	μ	96	101	7
		σ	36	38	3
	total	μ	16,157	62,929	778
		σ	4,476	11,412	64
swap environment	switch act. env.	μ	21	267	159,322
		σ	18	186	2,398
	aas prop. upd.	μ	11	12	6
		σ	6	5	2
	total	μ	32	278	159,327
		σ	21	187	2,398

Table 8.4: Operation Time Comparison (Duration in ms) – Grouped By Step and Deployment Strategy

only the total time of the entire operation is provided. However, if sub-steps are present, the times of these individual steps as well as a total time are given in each case. For each time measurement of a step, the arithmetic mean (μ) of the individual times of the test runs is given together with the respective standard deviation (σ). All times are given in ms and are rounded to whole numbers. Except for the *new environment prepared* cell for Swarm, where values have been rounded to two decimal places to avoid loss of precision. The sub-steps listed in this table coincide with those defined in the concrete elaborations of the deployment strategies in Chapter 7.

The measured times for inference and uploading the model do not distinguish between the different ML model types used, since the goal was not to measure the time for the different model types, but for the execution times as a whole. In addition, all models are fairly simple examples of fault detection models. So, considering them separately would not be meaningful anyway.

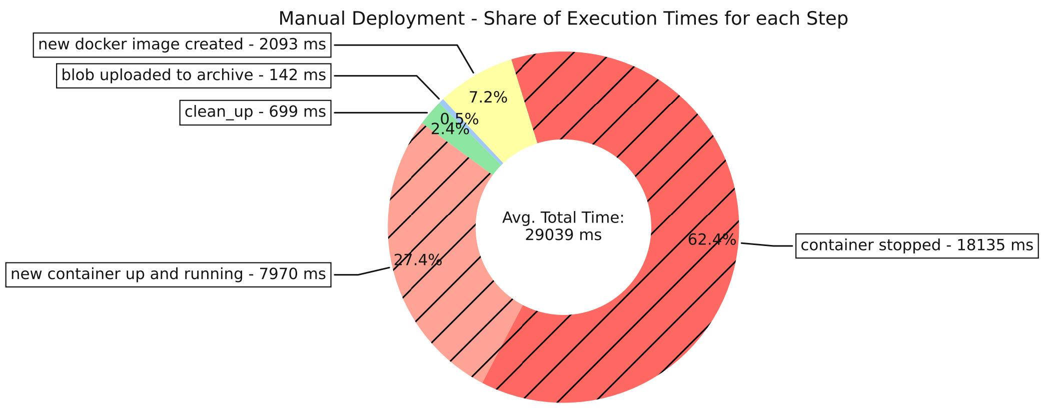


Figure 8.2: Manual Deployment – Duration per Step

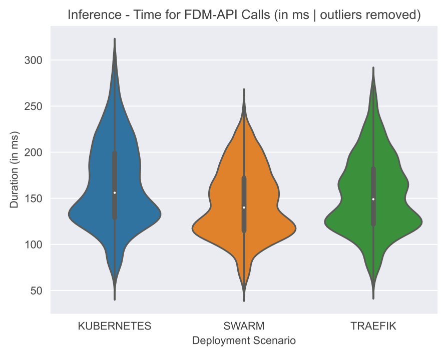
The times for certain steps in the *update environment* and *swap environment* operations differ greatly. This is because the respective strategies have to perform different tasks in these steps. However, this does not mean that the fault detection service is unavailable at any time. For example, in the *new environment prepared* sub-step for Kubernetes, two replicas need to be prepared, while in the approach with Docker Swarm there is no standby environment at all and therefore no preparations need to be made. This also explains why the standard deviation for the Swarm strategy is larger than the mean, since there is a relatively large scatter here for a very short period of time. This happens because only whole milliseconds are logged in the time measurement and for most of the 140 runs the time period is so small that the value is zero due to rounding, but in some cases it is one.

8.3.2 Graphical Evaluation and Details of the Execution Times

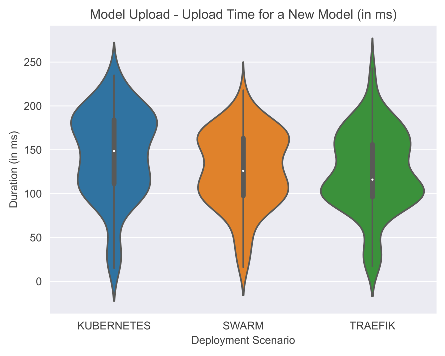
In the following, the obtained results of the technical experiment runs are presented. First, the temporal composition for a manual deployment is presented. Afterwards, the measured times for the four operations, separated by the three identified deployment strategies from Chapter 7, are presented. For the operations *update environment* and *swap environment*, the sub-steps are also considered.

Manual Deployment Assessment Initially, the graphical representation of the measured results of the manual deployment is given. Since the sub-steps and operations here do not correspond to the other deployment strategies anyway, a different type of presentation was chosen in this respect, namely a pie chart.

Figure 8.2 illustrates the duration of each step required for a manual deployment. Both the absolute value in the description and the relative percentage in the slice are displayed. The graphic also highlights the time the fault detection service is not reachable on average,



(a) Inference - Duration per Strategy



(b) Model Upload – Duration per Strategy

Figure 8.3: Distribution of Execution Times for (a) Inference and (b) Model Upload

represented by the striped sections in shades of red. Once again, it becomes clear that this accounts for almost 90% of the total duration, while pre- and post-work is minimal.

Deployment Strategies Assessment The color coding of the strategies is consistently the same: blue represents the blue-green deployment with Kubernetes, orange stands for the rolling deployment method with Docker Swarm, and green is the blue-green deployment using Traefik. In the cumulative views of the last two operations, the color coding does not represent the deployment strategy, but the individual sub-steps. The first two graphics utilize a violin plot to display the distribution of durations, since no sub-step times were determined for these two operations. For operations with sub-steps, both a bar chart is used to allow for comparison of mean values, and a box plot is used to depict the distribution of the measured times. To illustrate the temporal composition of the *update environment* and *swap environment* operations, a stacked bar plot was used.

Figure 8.3a depicts the distribution of the measured durations in ms for the inference operation. It is worth noting that the Interquartile Range (IQR) method was used to remove outliers from the underlying dataset for this plot. The lower bound of the included data range was defined as $Q1 - 1.5 * IQR$, while the upper bound was defined as $Q3 + 1.5 * IQR$. This was done because there were some outliers, particularly for Kubernetes and Traefik. For completeness, according to the measurements in Table 8.4, the mean values for Kubernetes, Swarm, and Traefik are 169, 145, and 156 ms, respectively. Figure 8.3b shows the distribution of upload times for new models into the AAS. No removal of outliers was necessary to get a well arranged graph. Figure 8.4 displays the distribution of the measured values for preparing the new environment using box plots. Important to note is the range of values of the y-axis, because there are different value ranges depending on the sub-step.

In Figure 8.5, the distribution of the measured values is depicted when switching between the environments. Particular in this graphs is the partitioning of the y-axis for the sub-

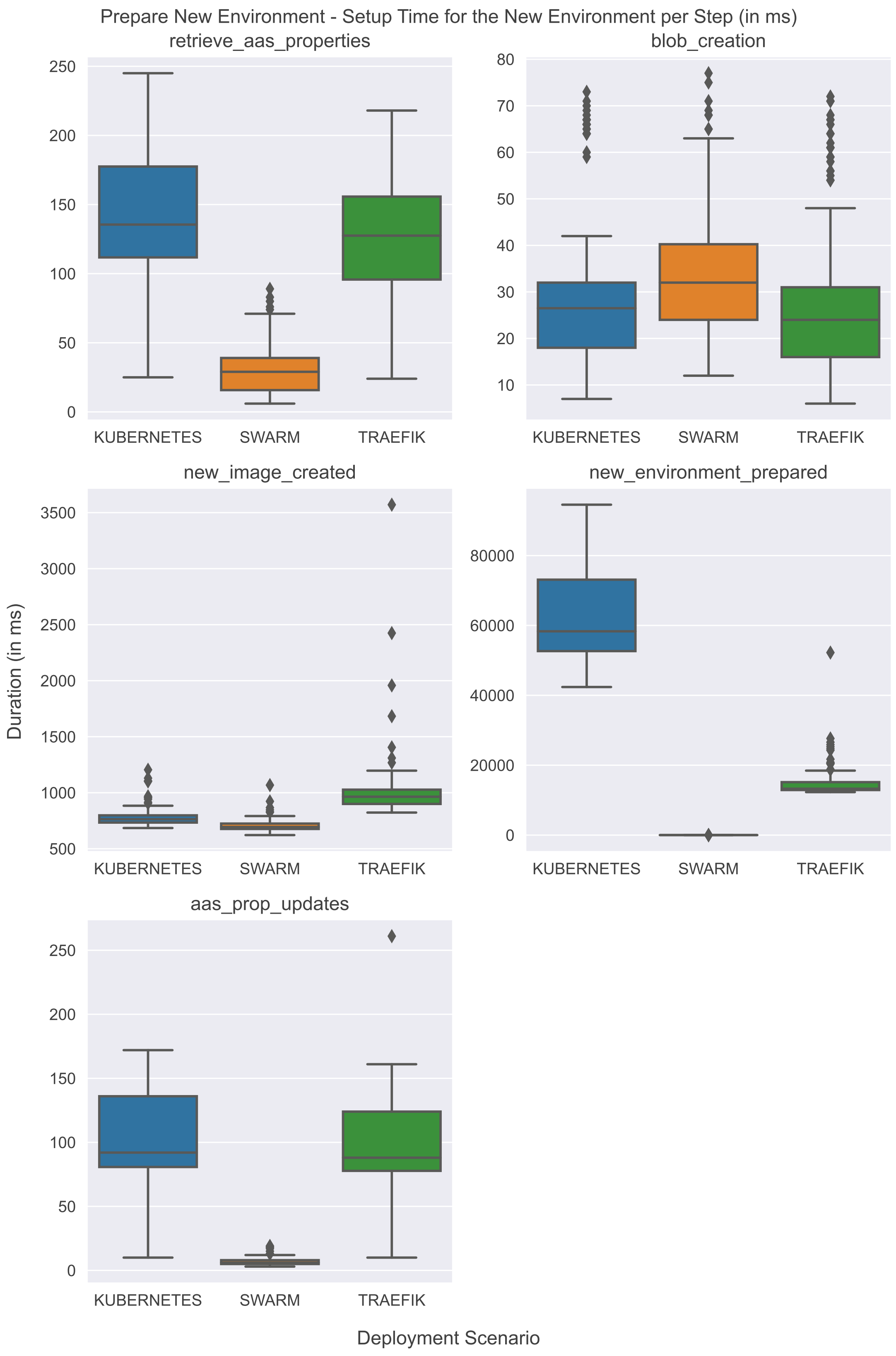


Figure 8.4: Update Environment – Distribution of Duration per Step per Strategy

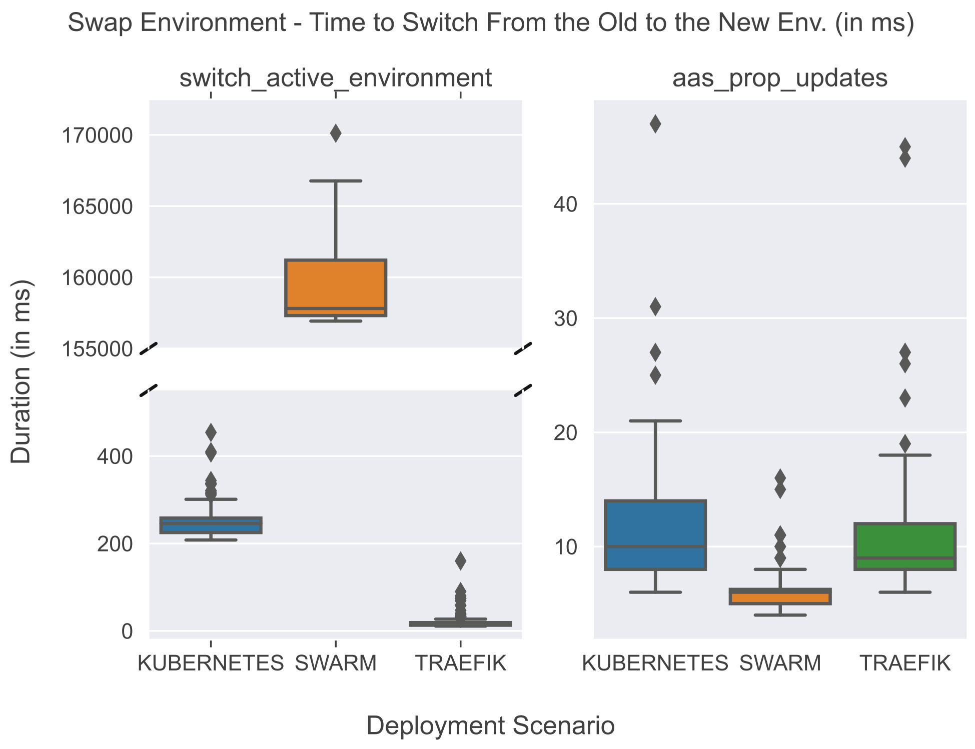
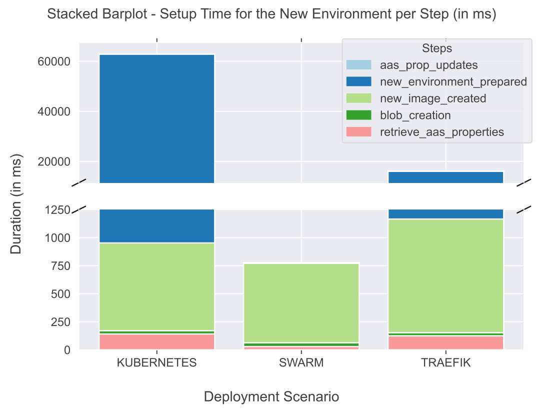


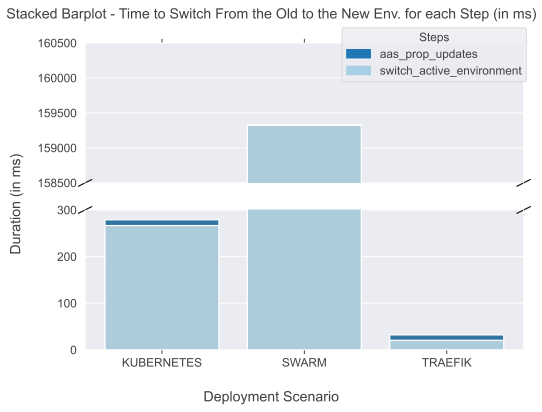
Figure 8.5: Swap Environment – Distribution of Duration per Step per Strategy

step *switch_active_environment*. This is necessary because the Docker Swarm approach has a very high duration on average for this step compared to the other two, which is due to the rolling deployment approach and the tasks required for it in this sub-step. One can clearly see that the two blue-green deployment approaches perform this sub-step on average in under 300 ms, while the Swarm approach requires on average nearly 160 seconds for that, which is also due to the defined update delay.

Figure 8.6a represents the total time of the update environment operation, composed of the times for the individual sub-steps. This is intended to show which step takes how much time, and puts this in direct comparison with the other sub-steps. The y-axis is also split in this figure because the average durations per strategy are quite different, which would otherwise distort the representation and thus the smaller shares would not be visible at all. However, the direct comparability of the bar sections is negatively affected by this. At the bottom of the stacked bar is always the first sub-step and at the top the last one. Because the time portion for updating the properties in the AAS is so small compared to the preparation time, the light blue bar for updating the properties is not visible, although it is included. The total time of the swap environment operation, which is composed of the times of the two sub-steps, is depicted in Figure 8.6b. Again, the y-axis was split for the same reasons as before. Furthermore, the same problem arises with the Swarm approach. Due to its total duration of almost 160 seconds, the 6 ms



(a) Update Environment – Cumulated Mean Duration per Strategy



(b) Swap Environment – Cumulated Mean Duration per Strategy

Figure 8.6: Cumulated Mean Duration for (a) Update Environment and (b) Swap Environment

required for updating the properties cannot be accurately represented.

CHAPTER 9

Discussion of Results

In this chapter, the obtained results, which have already been presented in Chapter 8, will be discussed as well as their implications explored. First, the results of the criteria-based evaluation are discussed, also highlighting the key findings. Afterwards, the deployment strategies as well as the results achieved are analyzed and interpreted. This includes a discussion of the performance of the strategies compared to the baseline approach. Finally, the wider implications of the results are discussed and moreover, limitations of this work are pointed out.

9.1 Discussion of the Criteria-Based Evaluation

This section discusses the results of the criteria-based evaluation, which are presented in Table 8.1 and Table 8.2. At the beginning, there is a brief discussion of the AAS concept on how well the requirements are met, followed by a detailed elaboration of the requirement fulfillment of the deployment strategies. At the end, they are compared with the baseline approach in terms of their requirement fulfillment.

9.1.1 AAS Concept

The integration concept for ML models into the AAS meets all of the criteria specified for it (those numbered R-5.3.<x>), as well as some from the other groups, although they were not considered essential for it. The AAS model, and in particular the *FaultDetectionModel* submodel, was developed in an iterative process to meet all defined requirements. This required intermediate checks of requirements fulfillment and, in parallel, verification that the dynamic deployment approaches could also be achieved.

This ensured that the developed concept was largely independent of the deployment strategy chosen. This was also successfully demonstrated with the prototype using the three deployment strategies and the manual deployment approach. Accordingly, this concept can be used for deployments that require a second environment in order to perform the corresponding preparations for the switch, as well as for those that do not require a second environment. Since Docker is used to encapsulate the fault detection model and only one endpoint to the outside world is required for ML inference, the concept is also independent of the type of model or its specific implementation.

In addition, the submodel for a sensor is also designed in a very generic way and allows the definition of any number of properties. This makes it largely independent of a specific use case. Multiple sensors can be modeled and defined in the AAS, which can be used for fault detection. It is worth mentioning that the input features for the ML model do not have to be restricted to sensors and their corresponding values. Other properties can also be used, even with constantly changing values. Therefore, the sensor submodel can be used to model these properties as well. For description and documentation purposes, as well as to provide contact information and to define responsibilities, corresponding elements have been provided.

Furthermore, the requirement for the fault detection module to form a logical unit is fulfilled. All elements that are exclusively required for fault detection are located within one submodel. This is also true for operations where a way has been found to enable all these requirements with the existing capabilities of the AAS metamodel and the BaSyx framework, despite the issues described above regarding invocation of operations in an AAS server and storage in MongoDB. Some elements associated with fault detection, such as the archive, have been deliberately outsourced to enable automatic saving in MongoDB and to keep the actual fault detection submodel as lean as possible. Among other aspects, this is also intended to avoid resource problems on edge nodes.

Models that have already been used are stored in the archive previously mentioned and newly uploaded models are also initially stored there. References to the models used as active or standby models are deposited in the fault detection submodel. In addition, the SMCs of the models in the archive contain elements for recording quality metrics of the models. This enables well-founded decisions on when to switch between models.

In conclusion, the defined integration concept for ML models completely fulfills all the defined requirements and, thus, the criteria-based evaluation for the concept can be considered a success.

9.1.2 Deployment Strategies

This section examines the extent to which the specified criteria and requirements are met by the deployment strategies. This comprises the first two sections of the Table 8.1. The requirements for deployment approaches are discussed first, followed by the requirements for dynamic updates. Thereby, the following designations are used: *Traefik* refers to a blue-green deployment strategy that uses the Traefik proxy to manage access to the active and standby environment. *Kubernetes* also refers to a blue-green deployment strategy, but with Kubernetes as the container management and routing service. And *Swarm* refers to the rolling deployment approach, with Docker Swarm as the management service for containers and access.

Deployment Concepts All three strategies are based on the same basic structure: The model is containerized and made available for external use via a defined API for ML inference. Therefore, these models can basically be used wherever Docker can be installed and network access is available.

Regarding the deployment strategy requirement, the blue-green and rolling deployment strategies are actually mutually exclusive. Thus, Traefik only fully satisfies the blue-green provisioning requirement and does not satisfy the rolling deployment requirement at all. This would require that the management service is able to update one container while at least one other container is able to serve the incoming requests. In principle, this would be possible with Kubernetes, since it enables the simple scalability of pods and thus accommodates a rolling deployment approach. However, this would require changes to the procedure and configuration shown here. The blue-green deployment approach, on the other hand, is fully implemented by Kubernetes. Swarm, in contrast, only fulfills the requirement for the rolling deployment approach, since a different strategy would simply require a second environment. However, all three strategies are based on a well-defined deployment strategy and thus fully meet this requirement.

It can be concluded that in terms of deployment concepts all three strategies meet the defined requirements and can be rated as equivalent in this regard. Which strategy is best depends primarily on the intended use case as well as the additional requirements and needs specified.

Dynamic Updates A similar picture emerges with regard to dynamic updates. Traefik and Kubernetes fully meet all the criteria and requirements defined for this section. These two strategies allow switching between two model versions without the fault detection service being unavailable in the meantime, i.e., zero downtime is achieved without any interruptions. Swarm also fulfills these aspects completely.

Where Traefik and Kubernetes differ from Swarm is in the requirements for a fallback strategy and quality assurance. While the first two also meet these requirements, Swarm only partially meets the fallback strategy requirement and does not meet the quality assurance requirement at all. One aspect of quality assurance that Traefik and Kubernetes do not sufficiently address is enabling A/B testing. Although this has been mentioned several times in the literature, it is only of limited use in the case of fault detection, whereas it would be different for a recommendation system, for example. However, it is important to test models before they go live, and this is still possible with the Traefik and Kubernetes strategies as they provide a second environment for preparation and testing.

The reason why Swarm differs from the other two strategies in this respect is primarily the lack of a standby environment. Therefore, it is not possible to immediately roll back to a previous or any version of a model, as it is possible with Traefik and Kubernetes. However, Swarm has internal rollback capabilities during an update itself in case of an error, but also to perform a rollback of a service configuration. Nevertheless, this is not implemented directly as an operation in the fault detection submodel and is also not as flexible as with the other approaches. There, it is feasible to swiftly switch to the previous model using the switch operation, or alternatively, to prepare any archived model in the standby environment before making the switch. With regard to the mentioned quality assurance, it is not possible with Swarm to test in advance a model that is not defined as active. This makes it difficult to determine the corresponding quality metrics for the ML models and to base a decision to switch between models on them.

All three strategies mentioned fulfill the compatibility requirement regarding the AAS metamodel as well as those imposed by the BaSyx framework. In addition, the infrastructure requirement that the fault detection submodel should be able to run on an edge node is met. This was achieved by using a separate BaSyx HTTP server to host the submodel and Docker as a containerization solution.

In conclusion, both Traefik and Kubernetes are two very well-suited strategies for performing dynamic updates while striving for zero downtime and also taking quality aspects and fallback possibilities into account. Swarm, in contrast, lacks the latter two aspects and thus can be ranked below Traefik and Kubernetes, which are again tied for the lead. If quality aspects and rollback options are not essential in a use case, then Swarm can also be a safe option.

9.1.3 Comparison With Baseline Approach

This section focuses on comparing the previously discussed strategies with the baseline approach, which was evaluated in Table 8.1 as well. The baseline approach is a simple recreate deployment pattern and can be simplified as a stop-restart approach. Therefore, only some of the defined criteria and requirements are fully met, while the majority of them are not met at all.

The baseline approach likewise uses Docker to containerize the ML model and provides its functionality via an API. It also takes into account infrastructure and compatibility requirements and supports the defined AAS integration concept.

Compared to the other strategies, the baseline approach lacks the avoidance of downtime of the fault detection service. This is because it lacks both a standby environment or, alternatively, a methodology to ensure continuous availability of the service. This fact also eliminates the possibility of testing models with regard to their functionality and performance before they are actually made operational. As a result, service availability and user experience also suffer due to downtime.

The lack of a fallback strategy also has a negative impact on the assessment of the baseline approach. If an error is detected in relation to the model or the provided results during productive use, then the same procedure must be performed for a rollback as if a new model is to be provided. This, too, leads to further downtime. In addition, the baseline approach also eliminates the possibility of mitigating this risk through separate tests before the model is set live.

In summary, all three deployment strategies described, namely Traefik, Kubernetes and Swarm, are superior to the baseline approach in all defined aspects. Traefik and Kubernetes can in principle compensate for all weaknesses of the baseline approach, while Swarm can mitigate some of the mentioned drawbacks, with the exception of quality assurance and rollback features. Thus, it can be clearly stated that in terms of the criteria-based evaluation, all three deployment strategies outperform the baseline approach, and Traefik and Kubernetes have performed very well overall, while Swarm also has performed satisfactorily. As mentioned earlier, the strengths of each approach also depend on the use case in which they are deployed.

9.2 Discussion of the Prototype Evaluation and Technical Experiment Results

This section discusses the results of the technical experiment runs performed with the implemented prototype. These results include the measured execution times of the individual operations for each deployment scenario as well as for the baseline approach. In Section 8.3, these results have already been evaluated in the form of a tabular as well as a graphical representation. The Tables 8.4 and 8.3 summarize the measured execution times for the strategies and the baseline approach. Furthermore, the graphical evaluations are presented in Section 8.3.

The procedure for this purpose is as follows: First, the results for the baseline approach are analyzed and compared to the results of the three identified deployment strategies. Afterwards, the four operations are analyzed step by step for each strategy, then the gained insights are used to provide a justification for the superiority of the three deployment strategies over the baseline approach.

9.2.1 Discussion of the Baseline Results

This section analyzes the results of the baseline approach, which are presented in Table 8.3 and summarized graphically in Figure 8.2. In the following, the baseline approach is also referred to as Manual Deployment (MD).

For uploading a new model via the *loadNewModelIntoArchive* operation, the MD took an average of 142 ms, which is in the range of Kubernetes but nearly 15 to 20 ms slower than Traefik and Swarm, although the time was measured here in the invoking script and not in the operation within the AAS. Creating a new Docker image requires essentially the same steps for the MD as for the other strategies. With the MD, however, this takes about 2 seconds, while it is only 0.7 to 1 second with the others. The standard deviation is also higher than for the other strategies. Since the commands used are basically the same, the significant difference can probably be traced back to the way they were executed. For the other strategies, these are executed via *cmd.exe*, while with MD they are started via the *Powershell*, which is more modern but also more complex and usually more resource-intensive.

The next steps can not be compared directly with the other strategies anymore. Stopping the container takes 18.1 seconds on average with a standard deviation of about 8 seconds. Afterwards, the container will be restarted with the previously created image, which takes on average just under 8 seconds. After startup, the process will wait until the model contained therein is accessible via the API. Finally, the old Docker image is removed, which takes approximately 0.7 seconds.

Overall, the fault detection service is unavailable for an average time of 26.1 seconds, beginning from the moment the container is stopped until the new one is available. The MD can be most appropriately compared to the Traefik strategy, where essentially the same steps are performed in the *new environment prepared* step, but with some additional steps as well. Traefik takes about 14.9 seconds on average to complete this entire step.

What happens in MD directly after starting the container, which is not the case in Traefik, is waiting for the model to be reachable. With Traefik, however, this is checked later in the evaluation program. This implies that in the meantime the container has fully started and the model is reachable with Traefik, while in MD there is always a delay of five seconds before the reachability is checked again. In addition to this factor, the execution of commands via the *Powershell* once again plays a role to the disadvantage of MD.

9.2.2 Discussion of Operation Execution Times

The results for the four defined operations for the three deployment strategies are discussed below.

inference This operation consists of only one step. The time measurement starts before the reference elements are resolved and stops after the result is retrieved and the values in the AAS properties are updated accordingly, along with possibly publishing an MQTT message.

Swarm has the lowest average time for the inference at around 145 ms. The standard deviation is also the lowest of the three strategies. This means that the measured times for this strategy have the lowest scatter, since this strategy hardly has any extreme outliers.

Traefik has a mean execution time of around 156 ms and is thus in the midfield of the three strategies, as does the corresponding standard deviation. The larger standard deviation can be attributed to some larger outliers, among other things. The overall slightly larger mean execution time could be due to Traefik's additional proxy service, which might lead to additional overhead or delays compared to Swarm, which uses only existing Docker components.

Kubernetes has the longest duration for this operation with an average execution time of 169 ms. Likewise, the standard deviation shows the largest value for this strategy, as there are some larger outliers here as well. Again, the administrative overhead of Kubernetes for managing the cluster could explain these slightly longer times for inference. When Kubernetes is activated on the machine, several containers are started that need to run continuously as they are required to perform all of Kubernetes' management tasks.

Overall, the time differences between these execution times are not really severe. Especially since, as shown in Figure 8.3a, 75% of the data points for all three deployment strategies are around or noticeably below 200 ms and there are only a few outliers that are significantly above that. However, no definitive conclusion can be drawn as to why some of these inference calls took so much longer than others and are therefore considered outliers. One possible explanation is that the workload of the entire system was quite high at these moments. In the technical experimental runs, the time required for publishing an MQTT message was not measured explicitly, but is included in the total time for inference. However, additional tests have shown that the time required to publish such an MQTT message is around 7.5 ms, which is far below the standard deviations for the individual deployment strategies and thus not indicative as a cause for the aforementioned outliers.

model upload Again, only the total time was measured without subdivision into steps. To this end, the same pre-trained ML models were repeatedly uploaded for all three strategies. Although these were of different sizes, this factor was the same for all test runs. However, no distinction was made between the respective models for the time measurements, since these were relatively simple and identical for all three strategies anyway. This was not distinguished for the other operations either. In retrospect, additional logging of the actual model type would have improved the interpretation.

Traefik and Swarm are almost on par for this operation in terms of average execution times, with Traefik having a slightly lower mean runtime of around 125 ms compared to just under 127 ms for Swarm. The standard deviation, in contrast, is slightly lower for Swarm than for Traefik. Kubernetes has the longest measured execution time for this operation at just under 144 ms on average. Similarly, the standard deviation is also slightly higher for this strategy than for the others. However, no outliers could be identified for all three strategies.

Since a connection is established solely between the evaluation program and the *FDM-Archiv* submodel hosted on the AAS server, and thus the routing of messages between Docker containers does not play a role at this point, the nearly equal performance of Traefik and Swarm can be explained. One reason why it still takes longer for Kubernetes than for the others can still be indirectly attributed to the setup regarding the containers and their management. As mentioned earlier, Kubernetes requires a number of constantly running containers for cluster management, which leads to a higher system load. This in turn can affect the upload and storage time of new models in the archive, since everything runs on one machine despite being distributed to individual servers.

As can be seen in Figure 8.3b, all three strategies exhibit a bimodal distribution to some degree, suggesting that there are two clusters of execution times. One explanation could be the size of the uploaded model. The pre-trained models range in size from a few KB through 100- and 200-KB models to a model over 1800 KB in size (all in BLOB format). However, a manual examination of the time logs and an attempt to relate them to the particular model uploaded did not reveal a clear relationship. Adding the model name or size to the timing logs would therefore have been helpful for this interpretation.

In conclusion, despite the differences between Traefik/Swarm and Kubernetes, model upload times are completely acceptable. There are no outliers, the scatter of the measured values is also not too high and for all three strategies more than 75% of the measured times are well below 200 ms. Since the implementation of this functionality in the *FaultDetectionModel* submodel is the same for all three strategies, the timing differences between the strategies are likely due to the containerization setup. Furthermore, it can be stated that the model size does not have a significant impact on the duration of upload times.

update environment This operation consists of five sub-steps, each will be analyzed and interpreted separately. A comparison in terms of better or worse between the three strategies is not possible for this operation, since there are some main differences in the way this operation is performed, as described in Chapter 7. Nevertheless, a more general comparison is still possible.

- **retrieve AAS properties** This step already reveals a clear difference between the strategies. While Traefik and Kubernetes need to load both the SMC for the model being prepared and the reference element for the standby environment, and update the direct endpoint property in the archive, Swarm only needs to retrieve the SMC for the model being prepared. This is directly noticeable in the measured times. While Swarm only needs about 30 ms on average, Traefik and Kubernetes need 125 and 142 ms, respectively.

  Especially the full loading of the SMC for the standby model and subsequently writing back the modification in the direct endpoint property that this environment is currently unavailable increases the execution time significantly, especially since this change is also persisted in the MongoDB.

- **BLOB creation** In this step, the same is happening for each strategy. The BLOB is loaded from the model in the archive, decoded, and stored as a *.joblib* file in a defined directory. The measured execution times for all three strategies are around 30 ms, with Traefik and Kubernetes below and Swarm above at around 34 ms. The scatter of these times is proportionally much larger than for other steps and quite a few outliers in the upper time range could also be identified. Figure 8.4 illustrates this vividly.

One possible reason for these fluctuations is the size of the model and thus the size of the BLOB file. However, why this value is higher for Swarm than for the other two strategists cannot be determined in more detail.

- **new image created** For this step, basically the same things happen for all three strategies. First, the necessary information is determined and afterwards the new image is created using the defined model. Nevertheless, the measured times differ significantly.

  Swarm and Kubernetes achieved similar average execution times of around 708 and 782 ms, respectively. Traefik, on the other hand, reached an average time of about 1010 ms. Moreover, the determined standard deviation of the first two is quite low compared to Traefik. In Figure 8.4, the identified outliers contributing to this high standard deviation for Traefik can be clearly seen.

  One main difference of Traefik from Swarm and Kubernetes is the base used to create the new image. While the latter two require a separate base container with a template image from which the new image is derived, Traefik uses the current image of the standby environment container to create the new Docker image. This factor could lead to the shorter creation times for the new image for Swarm and Kubernetes, since the base container is not running while the standby environment container is still running. This fact could also explain the higher standard deviation for Traefik, as more variability is prevalent here.

  Swarm requires a separate base container with its own image, as there is no standby environment that could be used as a base for creating the new Docker image. Kubernetes, on the other hand, does not provide direct access to the containers and images used, as it is possible with the Traefik approach, since they are managed by Kubernetes and deployed as pods. However, this indicates that using a separate container with its own image as the basis for creating the new image containing the updated model could noticeably reduce the average execution time as well as its scatter, since the other factors are not considerably different for these three strategies.

- **new environment prepared** The goal of this step is to set up the standby environment with the new model. However, significant differences can be observed between the three strategies in terms of their average execution times. Since Swarm does not provide a standby environment, almost no time is spent on it in this step. For Traefik and Kubernetes, though, the time required for this step varies significantly due to the different tasks involved in the preparation process. Figure 8.4 shows this difference impressively.

  The tasks for the environment preparation using Traefik are quite straightforward and are managed by the implemented operation. After the old container is stopped and subsequently removed, a new container can be started based on the previously created image. Afterwards, the temporary *.joblib* file and the old Docker image are deleted. These tasks take about 15 seconds on average, with a standard deviation of about 4.4 seconds. For most of the 140 test runs, execution times were less than 20 seconds, although some outliers were identified that heavily influence the calculated mean execution time.

Kubernetes has two replicas for each environment, as can be seen in Figure 7.6. Therefore, two replicas must also be patched during an upgrade. In addition, patching of containers in pods is managed solely by Kubernetes. So, to clean up after this step by removing unused Docker images, it is necessary to wait until all pods running the old image are stopped. Kubernetes, however, does not stop the old pods immediately, but stops them when the new ones are running stable. Since this takes some time, it only checks every 10 seconds to see if the old pods are all stopped so that cleanup can begin. But this task is managed by the implemented operation. On average the new environment is ready in about 62 seconds, with a standard deviation of just over 11.4 seconds. If a 10-second wait time starts just before the pods are actually stopped, it inevitably results in clusters of execution times with intervals of about 10 seconds. Hence, this wait time also delays the actual execution time slightly.

Establishing a clear basis for comparison is difficult here, and it is challenging to draw conclusions. The time saved by Swarm in this step is realized only later. With regard to the two blue-green deployment strategies, the longer execution time of Kubernetes is not necessarily negative, since only the standby environment is affected by downtime and the pods with the new image and model are available relatively quickly. Only the cleanup tasks take considerably longer than with Traefik.

- **AAS properties updates** In this step, it depends again on the number of properties and how these properties are updated. For Traefik and Kubernetes, the direct endpoint for the model and the corresponding reference for the standby environment have to be updated. Additionally, the latter change must also be persisted to the MongoDB. For Swarm, the reference of the model in the archive is stored in the *StandbyFDM* element, even if there is no standby model available in this strategy. However, this information is needed for the actual switch. Subsequently, this change is also stored in MongoDB. Traefik and Kubernetes each take around 100 ms on average, while Swarm only needs 7 ms.

  A key difference between Swarm and the other two strategies when saving the reference is that the former creates a completely new reference, while Traefik and Kubernetes adapt the existing reference. This procedure requires processing via a loop and might suggest that this accounts for the significant time difference. Otherwise, only the endpoint is additionally set in Traefik and Kubernetes. Unlike the other steps, these times are not too significant and are also much shorter than retrieving information from the AAS.

Looking at the total duration for this operation in comparison, it can be seen that on average Swarm needs less than 0.8 seconds for the preparation tasks. This is mainly because only the new Docker image is created. Traefik, on the other hand, takes just over 16 seconds on average to prepare the standby environment with the new model and make it operational and ready for access. Kubernetes needs an average of 63 seconds. However, it should be mentioned that two replicas are involved here.

In Figure 8.6a, this composition of the total time is provided as well as a comparison between the strategies. For Traefik and Kubernetes, the majority of the time is spent starting the new container and cleaning up the components that are no longer used, followed by creating the new Docker image. In the case of Swarm, the latter accounts for the largest amount of time, as there is no standby environment to prepare. From an overall perspective, the other sub-steps are only marginally involved.

swap environment The fourth operation results to be discussed concern the actual switch from one model to another. This operation consists of two sub-steps and differs greatly between the strategies, as they all perform this switch based on a different approach. The details are discussed below.

- **switch active environment** This step is responsible for the critical task of switching incoming traffic from the old to the new fault detection model, resulting in significant differences in execution times between strategies.

  While Traefik simply calls a switch operation within the Traefik proxy container to update the dynamic configuration, Kubernetes requires patching of the service configuration through the command line. In Traefik, all these tasks are contained in self-implemented functions, whereas in Kubernetes, the changes to the service configuration are managed by Kubernetes itself. Swarm, on the other hand, takes a completely different approach by performing a rolling update.

  Traefik needed an average of 21 ms to change the configuration, with a standard deviation of 18 ms, which is a relatively high spread. Since the configuration change is made via a REST API call, the overall system load can also negatively affect the request and response times. In addition, there are also some outliers that have a negative impact on the mean value. However, since this API call only changes the dynamic configuration file in the container, the switching of traffic from one environment to the other is the exclusive responsibility of the Traefik proxy and, thus, only happens within this container. This is therefore executed immediately after the configuration change is detected by the proxy. Furthermore, the third quartile is at low 19 ms and more than 90% of the measured execution times are still below 40 ms.

  Kubernetes took an average of 267 ms to patch the service configuration, with a standard deviation of 186 ms, also indicating a larger scatter. This configuration update is managed by Kubernetes and is initiated via a command line command. There are two factors that may affect these execution times: First, the overall system load tends to be higher when using Kubernetes, and second, executing a command via the command line from Java and reading the response from *stdin* or *stderror* could be slower than a simple REST API call when running on the same machine. For this strategy, the third quartile is 258.25 ms and 90% of the measured execution times are below 300 ms, which is indicative of some severe outliers that can also be seen in Figure 8.5.

For Swarm, on the other hand, only the new Docker image was created in the previous step. Therefore, in this step, the image needs to be distributed to each replica through service updates. Since each replica is updated individually and an additional update delay can be specified, the final execution time for the switch is influenced by the number of replicas and the delay time. For this particular case, there were four replicas and a 30-second update delay, which resulted in an average execution time of just under 160 seconds. As can be seen by the low relative standard deviation of 2.4 seconds as well as in Figure 8.5, the majority of the measured values are close to the mean. Especially in the range above the mean, there are still some values that deviate significantly from the distribution and, therefore, some of them can be classified as outliers. The third quartile is at 161 seconds just above the mean and the maximum value of about 170 seconds is also not too far away.

In summary, the switch with Traefik is very fast and Kubernetes also performs reasonably well. It is worth noting that these times do not represent any downtime, but rather reflect the time needed to adjust the configuration. In contrast, switching with Swarm takes longer due to a different underlying strategy. Reducing the number of replicas and eliminating update delays could reduce these execution times even further. However, the primary goal with rolling updates is often to achieve stable updates without any downtime, not to complete updates as quickly as possible.

- **AAS properties updates** As with the previous operation, this step writes back the changes in the reference elements. For Traefik and Kubernetes, this includes the active environment, swapping the references of the active and standby environments, and updating the reference of the serialized fault detection model currently used. In addition, these changes are also updated in the MongoDB. For Swarm, the value of the standby environment reference is set to the active environment reference, then the standby environment reference is deleted. Also, the reference to the serialized model is updated and all these changes are persisted in the MongoDB.

  On average, Traefik and Kubernetes require 11 ms for these updates, while Swarm only needs about 6 ms. As shown in Figure 8.5, all three strategies have some outliers, but those of Traefik and Kubernetes are more widely spread. The shorter execution times of Swarm can be attributed to the lower number of updates. Overall, these updates are performed quickly for all three strategies.

The Traefik approach can perform the switch the fastest and is clearly ahead of Kubernetes. Switching environments accounts for the majority of the time required here. Swarm is far behind, but this is due to the different approach. Figure 8.6b provides a graphical summary of this composition. Overall, it is worth mentioning again that none of these approaches result in any downtime.

9.2.3 Disadvantages of the Baseline Approach

In this section, based on the comparison of the results of the baseline approach with those of the three strategies above, it is argued why the latter are superior over the baseline approach.

As has already been mentioned above, the baseline approach follows a recreate deployment pattern. The division of the steps differs therefore from the other presented strategies, as all steps are executed within one operation, which is additionally not contained in the *FaultDetectionModel* submodel. This is because it is not an actual automatism, but rather the processing of a script. In addition, the time measurement of the *inference* operation was not considered, since it does not differ from the other strategies and is performed exclusively within the AAS.

In summary, the comparison between the Traefik, Kubernetes, and Swarm strategies in the experimental test runs showed that they are superior over the baseline approach. Even though a downtime of approximately 26 seconds may not seem long, it is still too much when a downtime of zero is required. Especially, if the other three strategies are able to achieve this required downtime avoidance.

Another clear downside is the lack of integration into the AAS and the cumbersome way to access or change values in the AAS. In addition, the integrated persisting possibilities of values in the MongoDB, which is defined in the operations, is not available with this script-based approach. In addition, it is also important to consider the aspects already discussed in Section 9.1.3, which also highlight the advantages of the three strategies over the baseline approach.

9.3 Answering the Research Questions

This section answers the defined research questions from Section 1.2 based on the findings and insights gained.

Research Question 1

RQ1: Which deployment strategies for ML models and dynamic update methods for software components are mentioned in existing literature or have been applied in production environments that could be transferred to the AAS?
This research question formed the basis for the SLR and specifically for the research questions within the SLR, which have already been discussed in detail in Chapter 4. The deployment strategies and dynamic updating methods found were thereby categorized and their occurrence quantitatively measured. Most of the identified concepts that

can be applied to the AAS were also used to derive the criteria and requirements for the integration concept to be developed and the deployment strategies to be identified. Therefore, the following conclusions can be drawn:

Deployment Strategies for ML Models Regarding deployment strategies, containerization techniques are mentioned very often because they offer many advantages, such as easy model packaging, easy deployment, and easy creation of new images. Most models are deployed as a web service and offer their functions through an API. In addition, specific deployment and testing strategies were named, such as canary deployment, rolling deployment, and A/B testing. All of these strategies aim for zero downtime and can also be used in combination with an AAS, but only the rolling deployment approach is useful in the context of a fault detection model.

Dynamic Update Methods DSU as a term for updating software without interrupting users or programs is a quite frequently mentioned method, which can also be used to dynamically exchange models contained in the AAS. Virtualization and containerization technologies are also used for dynamic updates, especially in combination with a traffic management dispatcher. Thus, microservices also play a role for dynamic updates and can be combined with the AAS. In addition, deployment strategies for seamless switching during updates play a key role. Blue-green or canary deployments are mentioned, as is the requirement for testing capabilities in a separate environment.

Some relevant aspects in the course of dynamic updates are the consideration of timing constraints, compliance with infrastructure requirements and ensuring the quality of new models by means of a separate and isolated test environment. Also, the need to perform fast rollbacks and to ensure backward compatibility are aspects that can be applied to the proposed approach.

Research Question 2

RQ2: Given the current state of the AAS, what is a feasible approach to integrate a fault detection model into the AAS considering that the models should be called directly from the edge nodes that contain the real-time data? The identified way to integrate a fault detection model into an AAS is to encapsulate all relevant elements, such as operations, endpoint information, references to the currently used model, and references to te active and standby environments, in a single submodel. This submodel can be hosted on a separate edge node by running it on a simple HTTP server. Sensor values are also encapsulated in their own submodels and can be either co-hosted with the AAS or on their own server near the physical device. The actual functions for inference, uploading new models, preparing the environment, and switching between model versions are defined as operations in this submodel together with the required input and output parameters. These operations can be called from external clients via an API but also internally.

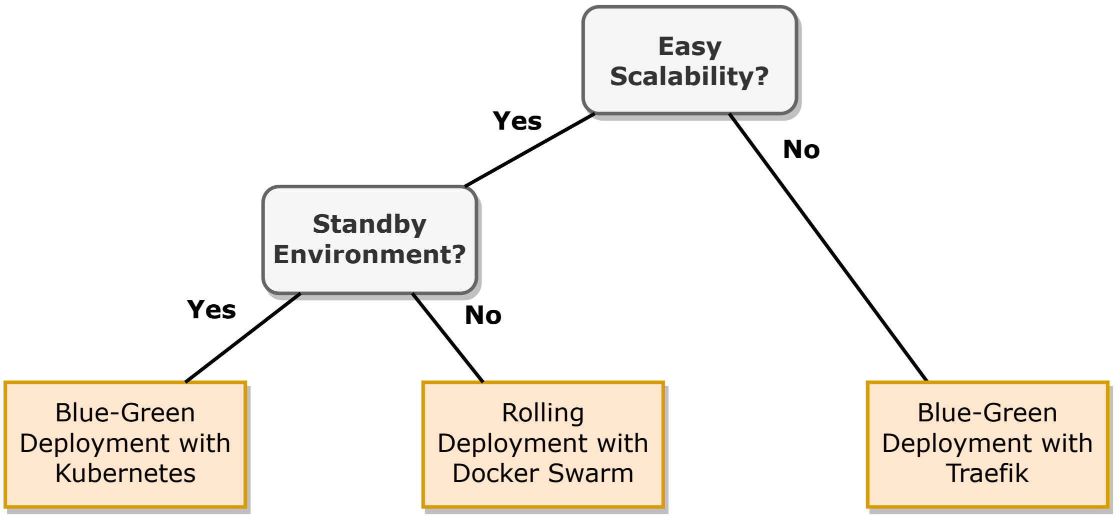

Easy Scalability?
Yes
No
Standby Environment?
Yes
No
Blue-Green Deployment with Kubernetes
Rolling Deployment with Docker Swarm
Blue-Green Deployment with Traefik

Figure 9.1: Decision Tree for Choosing the Optimal Deployment Strategy

In addition to the submodel for the fault detection model, there exists also an archive submodel. This does not have to be on an edge node as it contains all ML models that have been used before, are currently being used, or have just been uploaded but are not yet in use. In the archive, the serialized ML models are stored as BLOBs along with some relevant properties and quality metrics. The actual inference is invoked within the AAS. Thereby, the latest sensor values are retrieved from the submodel elements and subsequently the ML model running in the active container is invoked with them.

Research Question 3

RQ3: What is an appropriate strategy to deploy a fault detection model in a production environment and switch from the previous to the new model without stopping production processes?
A suitable approach that meets the specified criteria and requirements is to encapsulate the fault detection model in a container. This enables effortless deployment and facilitates the creation of new containers with the latest model. In addition, a deployment strategy that does not cause any downtime can be used to seamlessly switch between these deployed containers. All three deployment strategies identified meet this requirement.

The optimal strategy for switching between the old and new model depends on the use case and the associated requirements. Therefore, the decision tree in Figure 9.1 offers guidance on determining the optimal strategy. If the objective is to provide a standby environment for performing tests to ensure model quality and facilitate fast and straightforward rollbacks, a blue-green deployment approach with Traefik or Kubernetes would be preferable. On the other hand, if the priority is scalability to accommodate a high volume of incoming traffic, the strategies utilizing Kubernetes or Swarm would be favored. Although, no standby environment is available for the latter.

Alternatively, if a test environment is not required and the management overhead of Kubernetes or an additional proxy is too high, a rolling deployment strategy using Swarm would be the best choice. In summary, there is no universally “best” strategy, but depending on the use case, one of the three strategies can be considered the most appropriate.

9.4 Limitations of the Developed Approach

This section focuses on the limitations of this work and in particular of the created artifact. The developed approach aims to solve the problem of integrating a ML model for fault detection into an AAS and to enable dynamic updates of this model. Although it has shown promising results in initial tests and evaluations, it is crucial to identify the limitations and potential drawbacks. This knowledge is important to understand the scope and applicability of the proposed solution.

Integration Concept Related Limitations The following list summarizes the limitations, which were recognized or consciously accepted in relation to the developed integration concept for ML models for fault detection:

- **Missing Standardized Semantics** Within an AAS, elements should be provided with a semantic reference so that a context for understanding is provided. Any system accessing a property will know what that specific value means. This can be done in a number of ways, such as using ECLASS or providing a concept description within the AAS. For example, in the case of a temperature sensor, the unit of measure or type of sensor is described with such semantics. However, no data specifications or semantic IDs were provided in the presented integration concept for the AAS, since this was not required for the model integration and switching approach. This is because only one system used the provided values and, thereby, having knowledge of the context and the subsequent interpretation of the values. The use of semantic information would have made the entire concept and prototype much more comprehensive and complex without any direct benefit.
- **Security Aspects Omitted** During the development of the concept and also during the implementation of the prototype, topics related to security such as authorization or encryption were deliberately omitted. Taking this aspect into account would have made the concept and the prototype unnecessarily complex without adding any direct value for demonstrating the benefits of this approach. However, BaSyx for Java as well as the other software components used offer functionalities in this regard.
- **Maximum Model Size not Determined** During the evaluation it was not tested what the maximum BLOB size supported by an AAS is. There was also no indication of the maximum size for BLOBs in the documentation. Therefore, this

can be considered a limitation, since it cannot be stated at which size of the ML models saved as BLOB would cause the AAS to stop working.

- **No Event Handling** One aspect that has been defined outside the scope of this work is the fact that once a potential fault is detected, only an MQTT event is published, but no further action is taken.

Evaluation Related Limitations The evaluation is limited by the absence of recorded information about the type of model utilized during test runs, which makes it difficult to analyze more precisely the effect of model size on performance. Although this was not the main focus of the work, it would still be valuable to investigate how model size affects overall performance. Moreover, the developed concept and its prototype implementation used in the evaluation cannot be generalized to all types of ML models, since only some have been tested. Nevertheless, they should be theoretically supported considering the specifications presented above.

A second aspect related to the prototype and evaluation is that the technical experiments were conducted only on one machine. The reasons for this are explained in Section 2.2.4. Despite the same conditions for all strategies, the system performance or network latencies will have a different effect than if everything runs on separate components. Furthermore, the practicality of this approach is limited in its generalizability to various industrial application scenarios as it was evaluated using only emulated sensor data.

CHAPTER 10

Conclusion and Future Work

In this chapter, the main findings and contributions of this thesis are presented. For this purpose, the developed integration concept and the subsequently identified deployment strategies are briefly summarized and conclusions are drawn based on the evidence presented in previous chapters. It also identifies possible ways to extend this research and outlines open questions.

10.1 Conclusion

In this thesis, a generic concept for the integration of data-driven ML fault detection models into an AAS was developed. In addition, deployment strategies for these models were identified that allow dynamic switching from one model version to another without downtime. The foundation for the concept and deployment strategy approaches were requirements and criteria derived from the findings of a SLR. They were supplemented with requirements imposed by the AAS metamodel and implementation prerequisites of BaSyx. Both the integration concept and the three deployment strategies were assessed against these requirements and criteria in a qualitative evaluation. Furthermore, a prototype of this concept was implemented using the Eclipse BaSyx™ framework to subsequently conduct technical experiments to measure the execution times of the operations of the three deployment strategies for inference, uploading new models, preparing the environment, and switching between model versions. This prototype evaluation proved that the concept does indeed work and demonstrated the utility and suitability of this approach. In addition, a quantitative evaluation of these deployment strategies was performed and a comparison of the different deployment strategies was done.

AAS Concept The defined integration concept for ML fault detection models into an AAS meets all defined requirements and criteria. The core elements of this concept are the *FaultDetectionModel* submodel along with the *FDMArchive* submodel. The

		Blue-Green Deployment		Rolling Depl.
	BL	**Traefik**	**Kubernetes**	**Swarm**
Standby Environment	–	X	X	–
Separate Access to Active Model	–	X	X	–
Direct Access to Standby Model	–	X	X	–
Conduct Tests before Activated	–	X	X	–
Immediate Rollback	–	X	X	–
Easy Scalability	–	–	X	X
Out of the Box Switching	–	–	X	X
Avg. Inference Time (ms)		156	169	145
Avg. Model Upload Time (ms)		125	144	127
Avg. Update Env. Time (ms)		16,157	62,929	778
Avg. Swap Env. Time (ms)		32	278	159,329

Table 10.1: Comparison of the Three Deployment Strategies

former contains all properties together with operations directly related to performing fault detection using a ML model as well as uploading a new model and preparing the environment. The latter contains all current and previous ML models, where each model is described by a set of properties and contains the serialized model as BLOB along with a set of quality metrics. In addition, three further submodels (for general information, sensors, and capabilities) have been defined.

It can be concluded that this integration concept is a generic approach and allows to define various types of sensors for different application scenarios and also to incorporate different types of ML models. Moreover, this concept could not only be used as a way to integrate fault detection models, but also for other use cases such as predictive maintenance or further scenarios where data is periodically collected from sensors or other components and fed into a ML model to predict a specific outcome. Overall, this concept is generic, but also kept quite simple to allow many possible extensions for further use cases.

The following provides a concise overview of the commonalities and differences among the three presented deployment and switching strategies, namely both blue-green deployment strategies and the rolling deployment strategy. To this end, Table 10.1 contrasts these three strategies as well as the baseline (BL) approach. A comparison criterion is listed in each row. If the approach of that column fulfills this criterion or offers this possibility, it is acknowledged with a '*X*', otherwise it is marked with a '–'. Furthermore, the average execution times for each operation for the three deployment strategies are given at the end of Table 10.1.

Table 10.1 highlights similarities between these strategies, but also shows key differences. Only the blue-green deployment strategies provide a standby environment and thus can provide separate access to a model in this environment. Moreover, only in this environment the newly created model can be tested before it is declared the active model. Traefik and Kubernetes additionally provide a separate way to access the model in the active environment without affecting the endpoint used for production purposes. This can be useful, for example, when quality metrics need to be compared between active and standby models. Another advantage of the standby environment is that an immediate rollback can be performed if a particular condition does not meet the expectations.

In contrast, Kubernetes and Swarm allow easy scalability due to their native features provided by both orchestration tools. Therefore, if a model should be hosted on a cluster of nodes, these strategies are the right choice. All traffic management and routing is handled by these tools. Both of these approaches provide the ability to seamlessly move from one version to another with out-of-the-box functionality via the command line interface. Traefik, on the other hand, required some customization in the originally provided Docker image to allow for easy configuration changes via an API. At the end of Table 10.1, the average execution times of the four operations are listed again for comparison purposes. A detailed explanation can be found in the Chapters 8 and 9. It can be concluded that all three deployment strategies described, namely Traefik, Kubernetes, and Swarm, outperform the baseline approach in all defined aspects. Traefik and Kubernetes have the potential to address all the weaknesses of the baseline approach, while Swarm can alleviate some of the aforementioned drawbacks with the exception of quality assurance and rollback features.

10.2 Future Work

This section provides ideas for future work and addresses open questions related to dynamic deployment of fault detection models using digital twins, such as the AAS. While the integration concept of the fault detection submodel and the identified deployment strategies discussed in this thesis have shown promising results, there is still room for improvement.

One possible starting point is to incorporate features of advanced ML methods into the submodel as well. In general, the focus of this thesis was not to develop high-performing ML models for fault detection, but on developing a solid and generic submodel template for integrating a variety of different types of ML models. However, since only a number of rather simple models were used in the prototype implementation, the integration concept cannot be generalized to all types of ML models. Thus, future work could integrate advanced ML concepts, such as deep learning, reinforcement learning, or different types of neural networks, into the presented concept and reevaluate it. Moreover, MLOps platforms, in particular their pipelines to automate development and deployment, could also be integrated into an AAS to cover the whole lifecycle of models, which necessarily requires an extension of the presented integration concept to also cover, for example, the properties of training data, etc. in the AAS as has been done by Rauh et al. [RGB+22], among others. In addition, it could be investigated how the ML model size and type affect the performance for uploading the new model as well as for preparing the standby environment.

But there is also work to be done in the area of the practicality of this integration concept and the deployment and switching strategies. This includes, on the one hand, the implementation on physically distributed nodes, perhaps even at different production sites, and, on the other hand, the use of real sensor data from physical and continuously operating machines. The former means that the AAS registry operates on a completely different node than the AAS server hosting the AAS and the node hosting the fault detection submodel. After all, this concept was only evaluated on a test bench with emulated sensor data. However, in order to prove the practicality of this approach, the modeling of one or more real machines equipped with sensors is essential. In this context, various application scenarios should be considered to further demonstrate the generalizability of this approach. In addition, it could be examined whether the fault detection model can run directly on the hardware on which the PLC of a machine is running.

Other aspects that can be addressed in future work are some of the constraints mentioned in the previous chapter, such as considering security aspects like authorization or encryption, since data could be confidential or certain aspects contained in the AAS should not be accessible or visible for everyone. Additionally, the use of semantic references, such as ECLASS, in order to provide a semantic context especially for inter-application communication are another interesting aspect.

List of Figures

List of Tables

Acronyms

AAS	Asset Administration Shell. 1–4, 7–19, 25, 49–53, 55, 56, 59–64, 66, 67, 69–74, 76, 77, 79, 83, 84, 87, 88, 91, 92, 95–101, 104, 106, 109, 110, 112–116, 118, 120–125, 127, 129, 130
AI	Artificial Intelligence. 30, 32, 49, 51, 52
BLOB	Binary Large Object. 16, 17, 61, 69, 70, 73, 84, 87, 91, 102, 116, 117, 123–125, 128
CI/CD	Continuous Integration and Continuous Delivery. 30, 32, 33, 35
CPPS	Cyber-Physical Production System. 41
CPS	Cyber-Physical System. 30, 36, 41
DSU	Dynamic Software Updating. 40, 41, 44, 45, 47, 54, 55, 99, 100, 122
FMI	Functional Mockup Interface. 50
FMU	Functional Mockup Unit. 50–52
iCPS	intelligent Cyber-Physical System. 30
IDTA	Industrial Digital Twin Association. 15, 51, 52, 64, 66
IIoT	Industrial Internet of Things. 30
IoT	Internet of Things. 43
ML	Machine Learning. 3, 5–7, 9–11, 13, 19–21, 23–27, 29–36, 49, 53–56, 59, 61, 62, 64, 67, 69–74, 77, 79, 82–85, 87, 91, 96–100, 102, 109–112, 115, 121–125, 127–129
MVE	multi-version execution. 41
O-RAN	Open Radio Access Network. 31

PLC Programmable Logic Controller. 36, 42, 43, 50, 57, 95, 130

SLR Systematic Literature Review. 4–7, 25, 29, 33, 34, 36, 38, 46, 49, 53, 121, 127

SMC Submodel Element Collection. 16, 64, 66, 69–71, 73, 74, 110, 116

VAB Virtual Automation Bus. 18

WCET Worst Case Execution Time. 40, 43–45, 48

Bibliography

[AA21] Adrian-Ioan Argesanu and Gheorghe-Daniel Andreescu. A Platform to Manage the End-to-End Lifecycle of Batch-Prediction Machine Learning Models. In *2021 IEEE 15th International Symposium on Applied Computational Intelligence and Informatics*. IEEE, 2021.

[ABP17] Toyosi Toriola Ademujimi, Michael P. Brundage, and Vittaldas V. Prabhu. A Review of Current Machine Learning Techniques Used in Manufacturing Diagnosis. In *IFIP Advances in Information and Communication Technology*. Springer New York LLC, 2017.

[Arm20] Maggie Mae Armstrong. Cheat sheet: What is Digital Twin? `https://www.ibm.com/blogs/internet-of-things/iot-cheat-sheet-digital-twin/`, 2020. last accessed: 2023-03-17.

[AS18] Farshad Ahmadighohandizi and Kari Systä. Application development and deployment for IoT devices. In *Communications in Computer and Information Science*. Springer Verlag, 2018.

[Bö21] Matthias Bölke. Der digitale Zwilling beschleunigt die Industrie. `https://de.industryarena.com/emagazine/01-2021/der-digitale-zwilling-beschleunigt-die-industrie.html`, 2021. last accessed: 2022-09-10.

[Bar22] Nilesh Barla. Model Deployment Strategies. `https://neptune.ai/blog/model-deployment-strategies`, 2022. last accessed: 2022-06-03.

[BBB$^+$20] Andreas Bayha, Jürgen Bock, Birgit Boss, Christian Diedrich, and Somayeh Malakuti. *Describing Capabilities of Industrie 4.0 Components*. Plattform Industrie 4.0, 2020.

[BEdPT18] Fabienne Boyer, Xavier Etchevers, Noel de Palma, and Xinxiu Tao. Architecture-Based Automated Updates of Distributed Microservices. In *Service-Oriented Computing*. Springer International Publishing, 2018.

[Ben20] William C. Benton. Machine Learning Systems and Intelligent Applications. *IEEE Software*, 2020.

[Cen20] Cloud Architecture Center. Application deployment and testing strategies. `https://cloud.google.com/architecture/application-deployment-and-testing-strategies`, 2020. last accessed: 2023-02-17; author unknown.

[CG03] Stefan Cronholm and Göran Goldkuhl. Strategies for Information Systems Evaluation - Six Generic Types. *Electronic Journal of Information Systems Evaluation*, 2003.

[CS20a] Salvatore Cavalieri and Marco Giuseppe Salafia. A Model for Predictive Maintenance Based on Asset Administration Shell. *Sensors*, 2020.

[CS20b] Salvatore Cavalieri and Marco Giuseppe Salafia. Asset Administration Shell for PLC Representation Based on IEC 61131–3. *IEEE Access*, 2020.

[Fis21] Rene-Pascal Fischer. MQTT. `https://wiki.eclipse.org/BaSyx_/_Documentation_/_Components_/_MQTT`, 2021. last accessed: 2023-02-04.

[GCGb+19] Lucas Gisselaire, Florian Cario, Quentin Guerre-berthelot, Bastien Zigmann, Lydie du Bousquet, and Masahide Nakamura. Toward Evaluation of Deployment Architecture of ML-Based Cyber-Physical Systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop*. IEEE, 2019.

[GPR+21] Satvik Garg, Pradyumn Pundir, Geetanjali Rathee, P.K. Gupta, Somya Garg, and Saransh Ahlawat. On Continuous Integration / Continuous Delivery for Automated Deployment of Machine Learning Models using MLOps. In *2021 IEEE Fourth International Conference on Artificial Intelligence and Knowledge Engineering*. IEEE, 2021.

[GPS21] Denis Göllner, Thomas Pawlik, and Thomas Schulte. Utilization of the Asset Administration Shell for the Generation of Dynamic Simulation Models. In *2021 IEEE International Conference on Industrial Engineering and Engineering Management*. IEEE, 2021.

[GSK+22] Anastasios Giannopoulos, Sotirios Spantideas, Nikolaos Kapsalis, Panagiotis Gkonis, Lambros Sarakis, Christos Capsalis, Massimo Vecchio, and Panagiotis Trakadas. Supporting Intelligence in Disaggregated Open Radio Access Networks: Architectural Principles, AI/ML Workflow, and Use Cases. *IEEE Access*, 2022.

[HDZ+20] Huakun Huang, Shuxue Ding, Lingjun Zhao, Huawei Huang, Liang Chen, Honghao Gao, and Syed Hassan Ahmed. Real-Time Fault Detection for IIoT

Facilities Using GBRBM-Based DNN. *IEEE Internet of Things Journal*, 2020.

[HKBH19] Verena Halmschlager, Martin Koller, Felix Birkelbach, and René Hofmann. Grey Box Modeling of a Packed-Bed Regenerator Using Recurrent Neural Networks. *IFAC-PapersOnLine*, 2019.

[HMH21] Verena Halmschlager, Stefan Müllner, and René Hofmann. Mechanistic Grey-Box Modeling of a Packed-Bed Regenerator for Industrial Applications. *Energies*, 2021.

[HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. Design Science in Information Systems Research. *MIS Quarterly*, 2004.

[HMR+19] Waldemar Hummer, Vinod Muthusamy, Thomas Rausch, Parijat Dube, Kaoutar El Maghraoui, Anupama Murthi, and Punleuk Oum. ModelOps: Cloud-Based Lifecycle Management for Reliable and Trusted AI. In *2019 IEEE International Conference on Cloud Engineering*. IEEE, 2019.

[IDT23] IDTA. Registered AAS Submodel Templates. `https://industrialdigitaltwin.org/en/content-hub/submodels`, 2023. last accessed: 2023-02-01.

[JKS+22] Prerna Juhlin, Abdulkadir Karaagac, Jan Christoph Schlake, Sten Grüner, and Julius Rückert. Cloud-enabled Drive-Motor-Load Simulation Platform using Asset Administration Shell and Functional Mockup Units. In *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation*. IEEE, 2022.

[KBA+21] Heiko Koziolek, Andreas Burger, P. P. Abdulla, Julius Rückert, Shardul Sonar, and Pablo Rodriguez. Dynamic Updates of Virtual PLCs Deployed as Kubernetes Microservices. In *Software Architecture*. Springer International Publishing, 2021.

[KC07] Barbara Kitchenham and Stuart Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering. *Engineering, EBSE Technical Report EBSE-2007-01*, 2007.

[Kha21] Renu Khandelwal. Machine Learning Model Deployment Strategies. `https://arshren.medium.com/machine-learning-model-deployment-strategies-985a031f6ae1`, 2021. last accessed: 2022-06-01.

[KKT+18] Werner Kritzinger, Matthias Karner, Georg Traar, Jan Henjes, and Wilfried Sihn. Digital Twin in manufacturing: A categorical literature review and classification. *IFAC-PapersOnLine*, 2018.

[KLZ20] Kálmán Képes, Frank Leymann, and Michael Zimmermann. Situation-Aware Updates for Cyber-Physical Systems. *Communications in Computer and Information Science*, 2020.

[LDKC19] Chun-Cheng Lin, Der-Jiunn Deng, Chin-Hung Kuo, and Linnan Chen. Concept Drift Detection and Adaption in Big Imbalance Industrial IoT Data Using an Ensemble Learning Method of Offline Classifiers. *IEEE Access*, 2019.

[LFK+14] Heiner Lasi, Peter Fettke, Hans Georg Kemper, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business and Information Systems Engineering*, 2014.

[LJSY21] Hoejoo Lee, Youngcheol Jang, Juhwan Song, and Hunje Yeon. O-RAN AI/ML Workflow Implementation of Personalized Network Optimization via Reinforcement Learning. In *2021 IEEE Globecom Workshops*. IEEE, 2021.

[LSA+19] Hajer Lahdhiri, Maroua Said, Khaoula Ben Abdellafou, Okba Taouali, and Mohamed Faouzi Harkat. Supervised process monitoring and fault diagnosis based on machine learning methods. *International Journal of Advanced Manufacturing Technology*, 2019.

[MAAJ19] Imanol Mugarza, Andoni Amurrio, Ekain Azketa, and Eduardo Jacob. Dynamic Software Updates to Enhance Security and Privacy in High Availability Energy Management Applications in Smart Cities. *IEEE Access*, 2019.

[Mal21] Rishabh Malviya. Create a Zero-Downtime Deployment of Your Machine Learning API. `https://betterprogramming.pub/create-a-zero-downtime-deployment-of-your-machine-learning-api-6486cb6394c3`, 2021. last accessed: 2022-06-01.

[Mic18] Andreas Michalka. Experimentelle Untersuchungen eines Festbettregenerators mit feinem Kies als Speichermaterial. Master's thesis, TU Wien, 2018.

[MMH+19] Manil Maskey, Andrew Molthan, Chris Hain, Rahul Ramachandran, Iksha Gurung, Brian Freitag, J. J. Miller, Muthukumaran Ramasubramanian, Drew Bollinger, Ricardo Mestre, and Daniel Cecil. Machine Learning Lifecycle for Earth Science Application: A Practical Insight into Production Deployment. In *IGARSS 2019 - 2019 IEEE International Geoscience and Remote Sensing Symposium*. IEEE, 2019.

[MSI18] Vinod Muthusamy, Aleksander Slominski, and Vatche Ishakian. Towards Enterprise-Ready AI Deployments Minimizing the Risk of Consuming AI Models in Business Applications. In *2018 First International Conference on Artificial Intelligence for Industries*. IEEE, 2018.

[MvSB+20] Somayeh Malakuti, Pieter van Schalkwyk, Birgit Boss, Chellury Ram Sastry, Venkat Runkana, Shi-Wan Lin, Simon Rix, Gavin Green, Kilian Baechle, and Shyam Varan Nath. Digital Twins for Industrial Applications. `https://www.iiconsortium.org/pdf/IIC_Digital_Twins_Industrial_Apps_White_Paper_2020-02-18.pdf`, 2020. last accessed: 2023-03-22.

[NNP+20] Usama Naseer, Luca Niccolini, Udip Pant, Alan Frindell, Ranjeeth Dasineni, and Theophilus A. Benson. Zero Downtime Release: Disruption-free Load Balancing of a Multi-Billion User Website. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. ACM, 2020.

[NOP22] Jörg Neidig, Andreas Orzelski, and Stefan Pollmeier. Asset Administration Shell Reading Guide. *Plattform Industrie 4.0*, 2022.

[OMK+22] Moses Openja, Forough Majidi, Foutse Khomh, Bhagya Chembakottu, and Heng Li. Studying the Practices of Deploying Machine Learning Projects on Docker. In *The International Conference on Evaluation and Assessment in Software Engineering 2022*. ACM, 2022.

[PAHC19] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. Mvedsua: Higher Availability Dynamic Software Updates via Multi-Version Execution. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019.

[PKT19] Devon Peticolas, Russell Kirmayer, and Deepak Turaga. Mímir: Building and Deploying an ML Framework for Industrial IoT. In *2019 International Conference on Data Mining Workshops*. IEEE, 2019.

[Pla22] Plattform Industrie 4.0. *Details of the Asset Administration Shell - Part 1 - The exchange of information between partners in the value chain of Industrie 4.0*, 2022. Version 3.0RC02.

[Pla23] Plattform Industrie 4.0. What is Industrie 4.0? `https://www.plattform-i40.de/IP/Navigation/EN/Industrie40/WhatIsIndustrie40/what-is-industrie40.html`, 2023. last accessed: 2023-03-17.

[PRTV12] Ken Peffers, Marcus Rothenberger, Tuure Tuunanen, and Reza Vaezi. Design science research evaluation. In *Design Science Research in Information Systems. Advances in Theory and Practice*. Springer, 2012.

[RB17] Leandro Batista Ribeiro and Marcel Baunach. Towards Dynamically Composed Real-time Embedded Systems. *Logistik und Echtzeit*, 2017.

[RGB$^+$22] Lukas Rauh, Sascha Gärtner, David Brandt, Michael Oberle, Daniel Stock, and Thomas Bauernhansl. Towards AI Lifecycle Management in Manufacturing Using the Asset Administration Shell (AAS). *Procedia CIRP*, 2022.

[Rud20] Chaitanya Krishna Rudrabhatla. Comparison of zero downtime based deployment techniques in public cloud infrastructure. In *2020 Fourth International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)*, 2020.

[SbAT20] Maroua Said, Khaoula ben Abdellafou, and Okba Taouali. Machine learning technique for data-driven fault detection of nonlinear processes. *Journal of Intelligent Manufacturing*, 2020.

[Sch20] Frank Schnicke. VAB. `https://wiki.eclipse.org/BaSyx_/_Documentation_/_VAB`, 2020. last accessed: 2023-03-24.

[Sch21] Frank Schnicke. Overview. `https://wiki.eclipse.org/BaSyx_/_Overview`, 2021. last accessed: 2023-03-24.

[Sch22a] Frank Schnicke. AssetAdministrationShell. `https://wiki.eclipse.org/BaSyx_/_Documentation_/_AssetAdministrationShell`, 2022. last accessed: 2023-02-03.

[Sch22b] Frank Schnicke. Hierarchical MQTT. `https://wiki.eclipse.org/BaSyx_/_Documentation_/_Components_/_AAS_Server_/_Features_/_Hierarchical_MQTT`, 2022. last accessed: 2023-02-04.

[Sch22c] Frank Schnicke. Operation Delegation. `https://wiki.eclipse.org/BaSyx_/_Documentation_/_Components_/_AAS_Server_/_Features_/_Operation_Delegation`, 2022. last accessed: 2023-02-03.

[Sch22d] Frank Schnicke. Storage Backend. `https://wiki.eclipse.org/BaSyx_/_Documentation_/_Components_/_AAS_Server_/_Features_/_Storage_Backend`, 2022. last accessed: 2023-02-04.

[Sci23] Scikit-learn. scikit-learn - 9. Model persistence. `https://scikit-learn.org/stable/model_persistence.html`, 2023. Accessed: 2023-03-03.

[SGB16] Diego Quirino Silva, Everton Gomede, and Rodolfo Miranda Barros. Strategies for Zero-Downtime Releases: A Comparative Study. In *Proceedings of the 15th International Conference WWW/Internet 2016*, 2016.

[SKA20] Frank Schnicke, Thomas Kuhn, and Pablo Oliveira Antonino. Enabling Industry 4.0 Service-Oriented Architecture Through Digital Twins. In *Communications in Computer and Information Science*. Springer Science and Business Media Deutschland GmbH, 2020.

[SKKM09] Habib Seifzadeh, Ali Asghar Pourhaji Kazem, Mehdi Kargahi, and Ali Movaghar. A Method for Dynamic Software Updating in Real-Time Systems. In *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*, 2009.

[SLDVH21] Michael Sollfrank, Frieder Loch, Steef Denteneer, and Birgit Vogel-Heuser. Evaluating Docker for Lightweight Virtualization of Distributed and Time-Sensitive Applications in Industrial Automation. *IEEE Transactions on Industrial Informatics*, 2021.

[SLJK21] Umer Saeed, Young Doo Lee, Sana Ullah Jan, and Insoo Koo. CAFD: Context-aware fault diagnostic scheme towards sensor faults utilizing machine learning. *Sensors*, 2021.

[SLVH19] Michael Sollfrank, Frieder Loch, and Birgit Vogel-Heuser. Exploring Docker Containers for Time-sensitive Applications in Networked Control Systems. In *2019 IEEE 17th International Conference on Industrial Informatics*. IEEE, 2019.

[TZF+22] Adam Thelen, Xiaoge Zhang, Olga Fink, Yan Lu, Sayan Ghosh, Byeng D. Youn, Michael D. Todd, Sankaran Mahadevan, Chao Hu, and Zhen Hu. A comprehensive review of digital twin — part 1: modeling and twinning enabling technologies. *Structural and Multidisciplinary Optimization*, 2022.

[VM21] Eric VanDerHorn and Sankaran Mahadevan. Digital Twin: Generalization, characterization and implementation. *Decision Support Systems*, 2021.

[VSTK19] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security*. IEEE, 2019.

[WRO09] Michael Wahler, Stefan Richter, and Manuel Oriol. Dynamic software updates for real-time systems. In *Proceedings of the Second International Workshop on Hot Topics in Software Upgrades - HotSWUp '09*. ACM Press, 2009.

[WST+20] Rene Wostmann, Philipp Schlunder, Fabian Temme, Ralf Klinkenberg, Josef Kimberger, Andrea Spichtinger, Markus Goldhacker, and Jochen Deuse. Conception of a Reference Architecture for Machine Learning in the Process Industry. In *2020 IEEE International Conference on Big Data*. IEEE, 2020.

[WZ22] Stephen John Warnett and Uwe Zdun. Architectural Design Decisions for Machine Learning Deployment. In *2022 IEEE 19th International Conference on Software Architecture*. IEEE, 2022.

[XSLZ19] Yan Xu, Yanming Sun, Xiaolong Liu, and Yonghua Zheng. A Digital-Twin-Assisted Fault Diagnosis Using Deep Transfer Learning. *IEEE Access*, 2019.

[Yi19] Wang Yi. Design and Dynamic Update of Real-Time Systems. In *2019 IEEE Real-Time Systems Symposium*. IEEE, 2019.

[ZEK21] Marwin Zufle, Florian Erhard, and Samuel Kounev. Machine Learning Model Update Strategies for Hard Disk Drive Failure Prediction. In *2021 20th IEEE International Conference on Machine Learning and Applications*. IEEE, 2021.

[ZXW$^+$14] Yang Zhao, Fu Xiao, Jin Wen, Yuehong Lu, and Shengwei Wang. A robust pattern recognition-based fault detection and diagnosis (FDD) method for chillers. *HVAC&R Research*, 2014.