Christoph Maurer, BSc

Integration and Deployment of Machine Learning Models

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Ing. Dr.mont. Franz Pernkopf

Institute of Signal Processing and Speech Communication

Graz, February 2021

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Abstract

Over the past few decades, there has been much progress in the field of software development concerning the used technologies as well as the development methodologies. Since the development of traditional algorithms for some problem domains is difficult, the hope for further progress lies in the use of machine learning models.

The integration of machine learning models into software systems introduces new challenges. Though, the development and lifecycle of the models has many similarities with conventional software, the methodologies and technologies do not have reached the same maturity level. The challenges already start with the necessary collaboration of different teams and continue with correct versioning, persisting and serving the models for the production system. Due to debugging is different compared to traditional software, the reproducibility of the original model is of special importance.

Motivated by the problem statements of an existing software system, in this thesis solution approaches for the integration and deployment of machine learning models were worked out. On the one hand best-practices from different companies and projects were researched and on the other hand existing third-party solutions for different parts of the lifecycle relevant for model deployment were analyzed.

A further part of the thesis is about the current architecture of the software system and necessary extensions for the simple integration of the models.

Kurzfassung

In den letzten Jahrzehnten gab es viele Fortschritte in der Software Entwicklung, sowohl bei den eingesetzten Technologien als auch Entwicklungsmethoden. Da sich die Entwicklung von traditionellen Algorithmen bei einigen Problemstellungen als sehr schwierig gestaltet, erhofft man sich hier oft Fortschritte durch den Einsatz von Machine Learning Modellen.

Die Integration von Machine Learning Modellen in Software Systemen stellt eine neue Herausforderung dar. Zwar gibt es viele Gemeinsamkeiten in der Entwicklung und im Lebenszyklus der Modelle mit herkömmlicher Software, die Vorgehensweisen und Technologien sind aber noch nicht im selben Maße ausgereift. Die Herausforderungen beginnen bereits bei der notwendigen Zusammenarbeit unterschiedlicher Teams und gehen weiter mit der korrekten Versionierung, Speicherung und Bereitstellung der Modelle im Produktivsystem. Da sich die Fehlersuche anders als bei herkömmlicher Software gestaltet ist die Rückverfolgbarkeit zum ursprünglichen Modell von besonderer Bedeutung.

In dieser Arbeit wurden motiviert durch die Problemstellungen eines existierenden Software Systems, Lösungsansätze zur Integration und Bereitstellung von Machine Learning Modellen ausgearbeitet. Dabei wurden einerseits empfohlene Vorgehensweisen von verschiedenen Unternehmen und Projekten recherchiert und andererseits bestehende Lösungen von Drittanbietern für bestimmte Teillösungen im Lebenszyklus analysiert, die für die Bereitstellung der Modelle von Bedeutung sind.

Ein weiterer Teil der Arbeit beschäftigt sich mit der aktuellen Architektur des Software Systems und den notwendigen Erweiterungen zur einfachen Integration der Modelle.

Acknowledgement

This master thesis was carried out with the innovative company smaXtec animal care. Therefore, I want to thank all involved smaXtec employees for giving me the chance to work on this interesting project. I especially want to thank Tobias Rauter for the time working together in the backend team, what gave me a reasonable understanding of the system architecture and for helping me structuring this thesis.

I also want to thank my supervisor Franz Pernkopf for his openness for this industry project. I am especially grateful for giving me final feedback right before Christmas holidays and helping me with all other organizational tasks.

Furthermore, I want to thank my study colleagues Benedikt Maderbacher and Rudolf Wörndle for proofreading parts of this thesis and for endless discussions about computer science related topics.

Finally, my gratitude goes to my family for their appreciation of higher education and reminding me to finish my studies in foreseeable future.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Scope . . . 2
- 1.3 Structure of this document . . . 6

2 Background **7**
- 2.1 Traditional Programming vs Machine Learning . . . 7
- 2.2 Challenges of Machine Learning in Software Systems . . . 9
  - 2.2.1 Team Organization . . . 10
  - 2.2.2 Reproducible Model Training . . . 10
  - 2.2.3 Different Frameworks and Programming Languages . . . 12
  - 2.2.4 Existing System Architecture . . . 13
  - 2.2.5 Frequent Deployments . . . 16
  - 2.2.6 Model Definition . . . 18
  - 2.2.7 Model Debugging and Interpretability . . . 19
  - 2.2.8 Training-Serving Skew . . . 20
  - 2.2.9 Monitoring . . . 21
- 2.3 Maturity of Machine Learning Systems . . . 23

3 Existing Technologies and Solutions **25**
- 3.1 Model Persistence . . . 25
  - 3.1.1 Python Pickle . . . 26
  - 3.1.2 Predictive Model Markup Language . . . 27
  - 3.1.3 Open Neural Network Exchange . . . 27
  - 3.1.4 SavedModel . . . 28
  - 3.1.5 Docker . . . 28
  - 3.1.6 MLflow Models . . . 30
- 3.2 Version control . . . 31
  - 3.2.1 Data Version Control (DVC) . . . 31

3.3 Model Serving 32
3.3.1 Self-hosted Model Servers 32
3.4 End-to-End Machine Learning Platforms 34
3.4.1 MLflow 34

4 Architecture and Concept **40**
4.1 Requirements 40
4.2 Existing System Architecture 44
4.2.1 High-level Perspective 44
4.2.2 Apache Kafka 46
4.2.3 Faust 48
4.3 Feature Extraction 52
4.3.1 Implementation of the Feature Extraction Pipeline . . 55
4.4 Model Deployment 58
4.4.1 Model Management 60
4.5 Model Serving and Integration 62
4.5.1 Strategies 62
4.5.2 Serving the model as a REST-Service 64
4.5.3 Serving the model as a Faust Agent 67
4.6 Monitoring 68

5 Results **71**
5.1 Overview of the solution 71
5.2 Fulfillment of the requirements 73
5.3 Deployment Steps 75
5.3.1 Steps for deploying and integrating a new model . . . 76
5.3.2 Updating an existing model 76
5.3.3 Deployment of traditional algorithms 76

6 Conclusion **77**

Bibliography **79**

List of Figures

1.1 Intersection of the subject areas in this thesis 2
1.2 The Machine Learning Pipeline 4

2.1 Traditional Programming Approach 7
2.2 Machine Learning Approach 8
2.3 Steps of a Machine Learning Training Pipeline 11
2.4 Batch Processing vs Stream Processing 13
2.5 Microservice Architecture vs Monolithic Architecture 14
2.6 The model deployed as a service 16
2.7 Deployed Models 17
2.8 The model as a "black-box" 18
2.9 How can the prediction be explained? 19
2.10 Prediction accuracy vs Explainability 20
2.11 Training-Serving Skew 21
2.12 Monitoring ML Systems 22

3.1 MLFlow Web Interface - Overview of the experiments 35
3.2 MLflow Model Registry 38

4.1 Requirements Gathering 41
4.2 High-level perspective of the existing system architecture . . 44
4.3 Overview of Apache Kafka 46
4.4 Single-Threaded Asynchronous I/O 51
4.5 Feature Vector 52
4.6 Data Sources for Feature Extraction 53
4.7 Pipes and Filters Pattern 54
4.8 Feature Extraction Pipeline 55
4.9 Timestamp Bucketing 56
4.10 Simplified UML Diagram of Stream Algorithms 57

4.11 Deployment steps for Software 58
4.12 Model Management 61
4.13 How to integrate the model? 62
4.14 Dimensions of integration strategies 63
4.15 Example of a HTTP GET-Request and JSON-Response 65
4.16 Prediction Request/Response via REST-API, JSON serialized 65
4.17 Model deployed as a docker container exposing a REST-API . 66
4.18 Prediction Request/Response via REST-API, JSON serialized 67
4.19 Monitoring Component 68
4.20 Time until the prediction result can be verified 69

5.1 Overview of the suggested solution 72

Listings

3.1 Serialization and deserialization of scikit-learn models as python pickle files. 26
3.2 Serialization and deserialization with joblib. 26
3.3 Exporting a scikit-learn model to PMML. 27
3.4 Example of a Dockerfile for python scripts. 29
3.5 Example of a MLProject file. 36
3.6 Conda File. 36
3.7 MLflow Custom Model. 37
3.8 Loading a model from the registry. 39
3.9 Serving a Model via the MLflow command-line tool. 39

4.1 Kafka Producer Example. 46
4.2 Kafka Consumer Example. 47
4.3 Python Kafka Consumer Example. 48
4.4 Faust Agent. 49
4.5 Faust Table. 50
4.6 Command using Pipes and Filters. 54
4.7 Agent with Monitoring. 69

1 Introduction

Machine Learning (ML) has become accessible to a wide range of actors in recent years. From a technical- and financial perspective there are ML frameworks such as Tensorflow[1] or Scikit-learn[2] available for free and powerful hardware has become affordable. Additionally there are more and more resources to learn the basics to apply ML.

The potential of intelligent systems to transform many businesses is considered huge and compared to the impact of electricity [1]. This motivates organizations to run first experiments leveraging collected data for their business needs.

However, building suitable ML models is not sufficient and only a small part for most real-world use cases [2]. The models need to be integrated into data pipelines and replaced when necessary. Therefore Data Science- and Software Engineering teams need to work closely together and agree on an appropriate development workflow. A well-designed system architecture and infrastructure should simplify the workflow and help to achieve fast product iterations and high quality.

1.1 Motivation

The company smaXtec animal care GmbH initiated this project to integrate ML to their existing product. By conducting this master thesis the necessary adaptions to their existing software architecture and development processes should be analyzed.

[1] https://www.tensorflow.org
[2] https://scikit-learn.org/

smaXtec provides solutions for monitoring dairy cattle. Mainly by collecting temperature- and activity data from every sensor equipped cow the smaXtec system is capable to alert the farmer with health-, temperature- and calving-notifications.

These notifications are triggered by algorithms running at the backend of the system. To get reasonable results some of these algorithms need to be configured based on characteristics of the farm and animal. For example a cow on a pasture can be expected to be more active than a cow in a barn. To avoid manual farm specific configurations and the possibility to get better results and new knowledge from the collected data was the motivation to start experiments with ML.

Although this project handles the challenges of a company-specific software architecture, the vast majority of challenges will be very similar for any organization that wants to integrate ML models in their products.

1.2 Scope

This thesis is about the challenges and possible solutions of adding ML to a data-intensive system. This thesis does **not** describe how to create the best performing model.

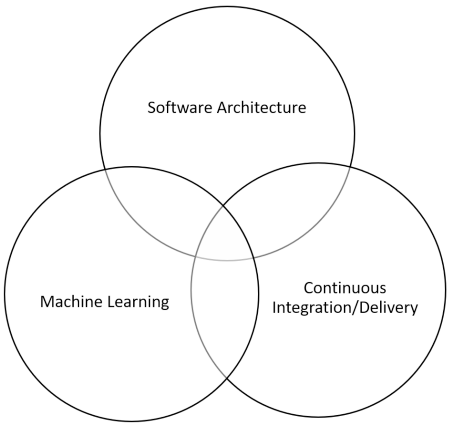

Figure 1.1: Intersection of the subject areas in this thesis.

Figure 1.1 illustrates the intersections of the subject areas handled in this thesis and are described in the following.

- **Software Architecture:**
  Software architecture describes the basic building blocks and relations between them, architectural styles and patterns of a Software System [3]. In this project the existing architecture was analyzed and possible extensions and adaptations were proposed.

- **Machine Learning:**
  Arthur Samuel described machine learning as the ability of computers to learn without being explicitly programmed [4]. This is a fundamental difference to traditional programming concepts like Structured Programming [5] where source code describes every rule of the computer program. The ability of ML to learn from data can solve problems that were not feasible to solve by explicitly programming the rules. A drawback of this approach is that it is hard to debug why decisions were made when the rules are not available. This thesis describes how a software system can benefit from ML and mitigate the drawbacks.

- **Continuous Integration/Delivery:**
  Continuous Integration (CI) is a software development practice where developers are encouraged to merge their work frequently with team members [6]. Continuous Delivery (CD) can be understood as techniques to improve the software delivery process in a way that new versions can be shipped frequently at least to a staging area. Continuous Deployment is an extended form where the software is automatically shipped to the production infrastructure. Those methods are essential when agile software development methods with short iteration lifecycles are practiced [7]. Due to the time-consuming nature of software testing there need to be automated tests to maintain high quality with every release [8].
  CI/CD of ML models is similar to traditional software, but this is a relatively young discipline with far less mature tooling available [9].

With this basic understanding of the subject areas described above it can be said that a software architecture and development processes should be

created that allow frequent deployments of ML models using best-practices like CI/CD.

Figure 1.2 illustrates a machine learning pipeline inspired by [10]. The pipeline can be separated into two parts, the first part is about creating (training phase) the model, the second part (running phase) takes the created model and integrates it into the production system to make predictions on new incoming data. The focus of this thesis is the second part and describes how to deploy ML models to a serving infrastructure and the requirements to run them safely.

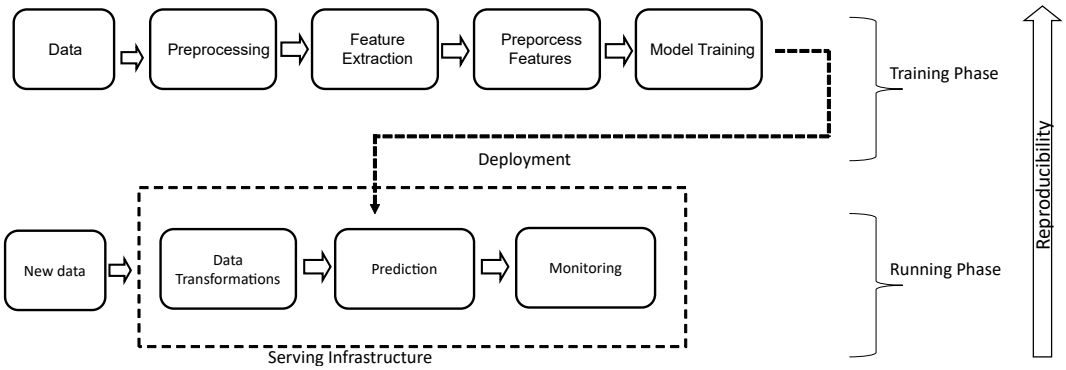

Figure 1.2: The Machine Learning Pipeline.

The following gives a short explanation of the parts in Figure 1.2.

Data: Before training, there needs to be enough data, which often requires data gathering from several sources. If the size of data is not too big it is usually sufficient to dump the data to offline storage like hdf5 [3] where the data can easily be accessed for further processing. However many projects need more advanced techniques to store and access data in the training phase.

Preprocessing: The quality of data is a critical factor in successful data mining projects. Hence, a preprocessing step where the raw data is cleaned is necessary. Typical tasks are outlier detection, removal of redundant or unnecessary data and the handling of missing or noisy data.

[3] https://www.hdfgroup.org/solutions/hdf5/

Feature extraction: After the preprocessing step, more work needs to be done before predictions can be made. Features need to be extracted. Examples of features for time-series are ticks, self-similarities and moving-averages. Features are sometimes called attributes or variables.

Preprocess Features: Features are the input for ML algorithms. To get satisfying results, appropriate features for the task need to be created and selected. This includes modification (value transformations) of features like feature discretization, feature weighting and feature normalisation. This feature preprocessing step is known as Feature Engineering.

Model training: In this step the actual code to create the models is written. The execution of this training code can be a long running task and may benefit from powerful hardware. If not otherwise stated, this thesis assumes supervised learning. In supervised learning the algorithms get labeled training data, which means feature values and the correct corresponding solutions, as input. As output of the training step, the best fitting model describing the data should be selected. It is common to split the data in 80 percent for training and 20 percent for testing a model [11] or doing cross-validation.

Deployment: In [12] deployment is described as the activities to deliver software or partially completed increments to the customer. This is similar in the case of ML models. In the deployment step the trained models need to be packaged in an appropriate format such that other parts of the software system can query predictions. It must be ensured that all necessary software components are correctly deployed and not only the models itself. For example, code for feature extraction needs to be available in the prediction system. Automating the deployment step is essential for doing frequent deployments.

Serving Infrastructure: The serving infrastructure is the part of software that runs the deployed ML models. In this thesis integration means the coupling of the serving infrastructure with other parts of the system. There are several options for the integration. This thesis tries to implement concepts for loose coupling [13] of the components.

Monitoring: As stated in [10], monitoring of ML systems needs to combine traditional software system monitoring like latency and throughput

with model performance metrics similar to the model evaluation in the training step. However, in the prediction phase it is not always possible to immediately check the prediction result and calculating metrics like in the training step, where labeled data with the real result is available, becomes difficult. Monitoring is the last step in the described pipeline and should be a decision helper when to start again at the beginning and create a new model if the results of the prediction phase deviate from the results of the training phase. Additionally, after this last step it should still be possible to reproduce (**Reproducibility**) which transformations were done beginning at the raw data. If something went wrong or for analytical purposes, the question is which model created the output for a given input and how was this model created.

1.3 Structure of this document

Beginning with this first introduction chapter, the stages of the ML lifecycle and the relevant parts covered in this theses are explained. Chapter 2 mainly describes the challenges of productionizing ML models and maturity levels of ML systems.

Chapter 3 names and describes existing third-party solutions for different subproblems more or less relevant for this project. Chapter 4 describes parts of the existing system architecture and discusses different options how ML models can be deployed and integrated. In Chapter 5 a concrete implementation for smaXtec is discussed. And finally, Chapter 6 sums up the key findings of this thesis.

2 Background

This chapter begins with explaining the differences and similarities of traditional programming and ML. The second and main part of this chapter describes the challenges of adding ML models to software systems. The third and last Section talks about the production readiness of ML systems. The main idea of this chapter is to derive requirements for ML systems from related work and to avoid common pitfalls.

2.1 Traditional Programming vs Machine Learning

Software Engineering is a fast evolving discipline with many tools, languages, frameworks and processes that hardly can be overviewed. In the case of traditional programming the basic workflow is well understood and despite that there are many different processes and methodologies, the approach can roughly be described as in Figure 2.1.

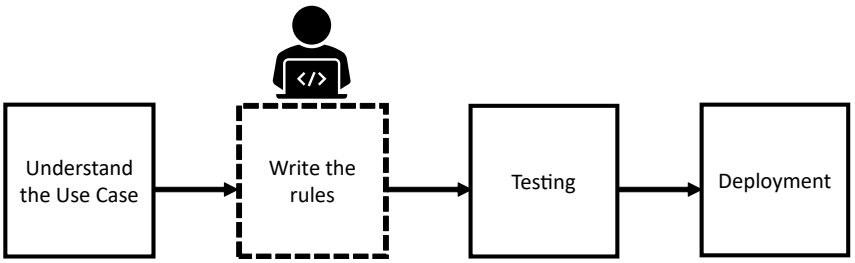

Figure 2.1: Traditional Programming Approach.

After understanding the use case, the programmer needs to write the rules of the algorithms manually. There are tools available that support the generation of source code from a modeling language like Unifed Modelling

Language (UML) [14]. But this is rather useful for the structure of the source code using the class diagram in the case of UML, than for the actual algorithms. And even this modelling is a manual task and only a graphical representation of the rules. After writing the rules, the program can be tested and delivered to the users (deployment).

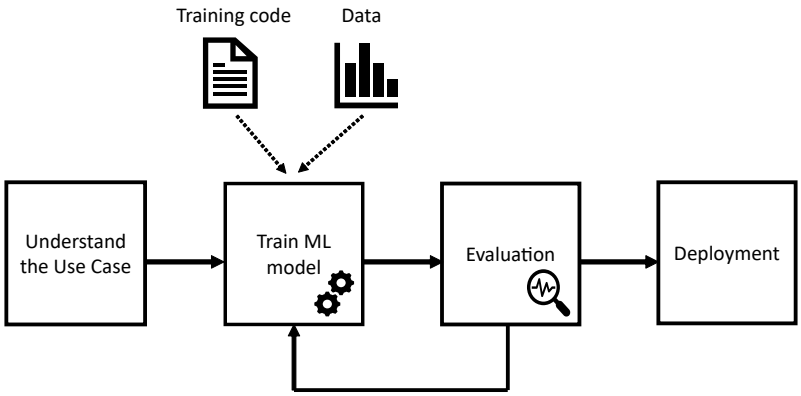

Figure 2.2: Machine Learning Approach.

The traditional programming approach works well for use cases where a human is able to understand the problem and then can write a suitable algorithm with reasonable effort.

For some use cases like image recognition it is hard for a human to find and formalize the rules. With supervised ML it is possible to let the computer learn the rules from historical data which can then be applied for new data. The data scientist needs to write training code so that the computer can optimize the parameters of a learning algorithm. Figure 2.2 illustrates the ML approach where the training and evaluation is done in a loop until a good enough, deployable model is found. Despite from the training code, this approach does not require to write rules by a human, but there is still manual work to do like configuring (setting the hyperparameters) and selecting a learning algorithm. To accelerate the ML approach and to make

it accessible to more developers there is research [15] and commercial tools (AutoML) to automate these manual tasks.

2.2 Challenges of Machine Learning in Software Systems

The task of running a production-ready ML System is often underestimated [2] and has several challenges to solve. Large corporations like Uber, Facebook and Google already made huge profit with ML in their products and shared their experiences with their ML systems [16]–[18]. Microsoft did research on the comparison between ML and traditional software engineering [19]. Amazon described several model management challenges and divided them into the groups *conceptual challenges*, *data management challenges* and *engineering challenges* [20].

Even if the requirements concerning the amount of data and the difficulty of the problem statement in this project is smaller it might be a good idea to learn from the lessons of the large corporations to focus on the right tasks and avoid typical mistakes beginners in the field often make. Although this thesis should focus on the time after model training some challenges can not be completely separated from the phases in the ML lifecycle and also need to be addressed here.

The following subsection gives a brief description of common challenges which might be somehow relevant in this project.

2.2.1 Team Organization

On most commercial software projects, there are many different people participating. Traditionally software developers are often categorized into frontend developers who focus on user facing parts of the application and backend developers, who create server-side application code. When adding ML or for other analytical purposes, the new role of the data scientist comes up.

In [21], they identified nine different categories of data scientists based on their usual activities. Having a limited budget, it is not easy to hire the right types of data scientists needed having the required skill-set. Therefore in smaller projects the data scientist needs to act as a generalist which includes data preparation as well as following the traditional software development processes. However, there are many tasks which data scientists can delegate to other team members. In some projects there is the specific role of a machine learning engineer. That role focuses more on making the models and data pipeline production ready than a more research oriented Data Scientist.

2.2.2 Reproducible Model Training

Model training is the central part in the phases of the ML lifecycle. The most suitable algorithm and parameters need to be found. For reasons of practicality it might also be necessary to choose another model for production than the most accurate one found in the training phase. The training phase is often a time-consuming and resource-intensive task and might require specific hardware and frameworks for distributed programming.

From a post-training perspective, it is essential to have all the metadata for a given model and especially the evaluation metrics. Having the metadata and metrics allows to monitor the model in production and detect deviations from the expected results and trace the production metrics back to the original data and settings which were used to create the model.

Figure 2.3 shows a simplified version of an example training pipeline similar to [22] . The pipeline is described with four steps starting from the raw data

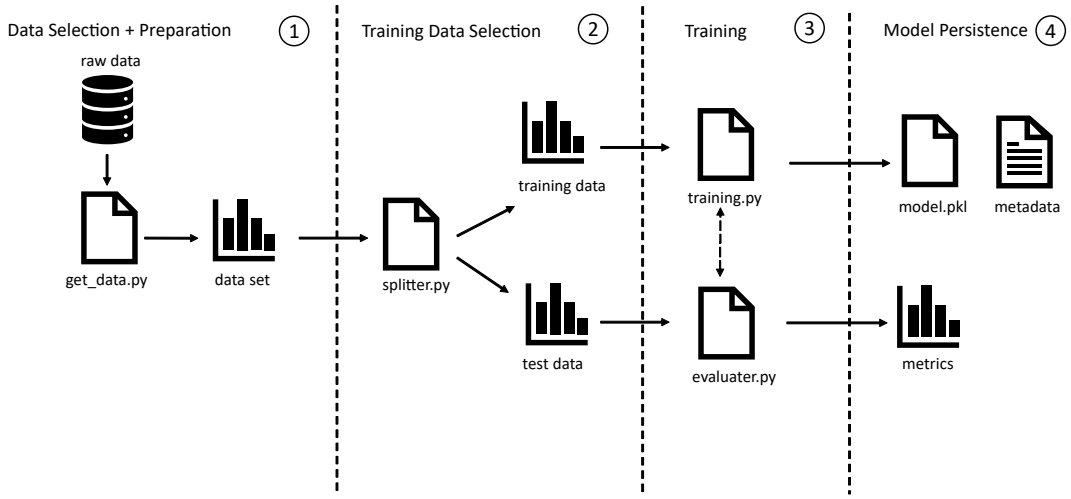

Figure 2.3: Steps of a Machine Learning Training Pipeline. Based on: [22].

selection to the persisted model. In this process are several artefacts and transformations involved which need to be versioned or tracked for reasons of reproducibility.

The data selection + preparation step (1) is in reality a large and important step and depicted very simplified here. The output of this step is the data set which can be used for training and testing.

Before the actual training the data set is splitted into training- and test data in step (2).

Step (3) is the training step where different algorithms with different hyper-parameters, parameters and other settings are evaluated and as a result the model is selected.

The last step (4) deals with model persistence. In this example the model is serialized into a model.pkl[1] file. Additionally corresponding metadata and the performance of the model (metrics) should be stored.

According to [20], examples of useful metadata are:

- Creator (developer)

[1]https://docs.python.org/3/library/pickle.html

- Creation date
- Hyperparameters
- Applied Feature Transformations
- Training/Evaluation Dataset

To acquire information of every training run, it is recommended to track the metadata and evaluation metrics automatically like in the system described in [23].

Additionally it should be easy for other data scientists, as well as for automation pipelines, to run the training code again and create the exactly same results as in the original training run. This assumes all training files and information about the required dependencies and software versions need to be versioned.

2.2.3 Different Frameworks and Programming Languages

Every framework and programming language has strengths and weaknesses and team members have their own preferences. Data scientists may prefer other languages than backend developers do.

In a reliable and maintainable system, data needs to be correctly passed between components and it should be easy to refactor each component. The system architecture and technology choices should avoid the problem of too many different languages known as "Multiple Language Smell" [2].

Many popular libraries for scientific computing use python as a frontend language with an underlying native implementation. In this project python is used as a general purpose language for the data science- as well as the backend tasks.

2.2.4 Existing System Architecture

Not every software project can start from scratch. Extending an existing architecture may take several limitations concerning technology and environmental complexity into account [24].

Before adding ML models to a system the existing architecture and especially data flow needs to be understood. In the training phase, data scientists can develop and run their experiments mostly independent from the target system architecture. Having labeled data it is possible to evaluate the created models without interacting with the production system. However in the prediction phase this is obviously different. Calculating the feature vector, the input of the ML model, is often time-critical and highly dependent on the system architecture.

Basically one can distinguish between *Batch Processing* Systems and *Stream Processing* Systems. In simple words in batch processing, all the data needs to be stored and processed as a whole as it can be seen in Figure 2.4.

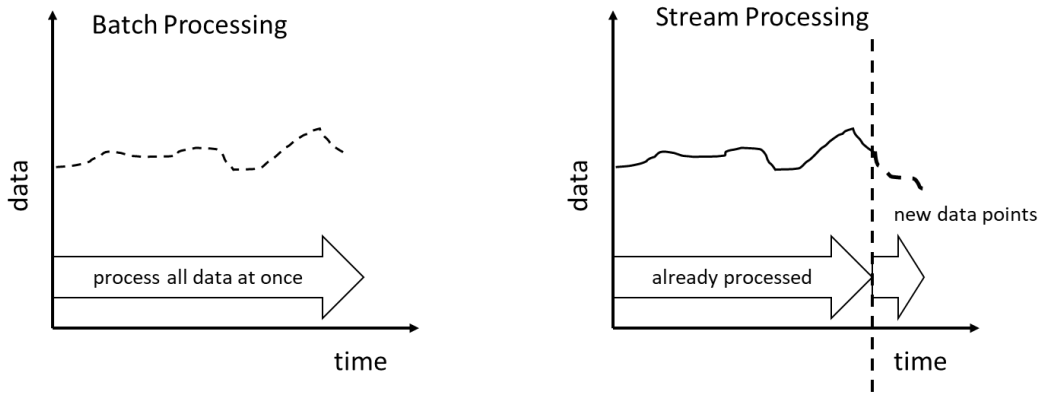

Figure 2.4: Batch Processing vs Stream Processing.

Whereas in stream processing only small pieces of new incoming data are immediately processed. In the training phase batch processing is usually the way to go, since this is a simple approach in a non-time-critical scenario. In the prediction phase stream processing might be the better choice, since it is not always possible to store and process all the data as a whole. Very

often the predictions are needed as soon as some patterns in incoming data occur and for example in time-series analysis a stream processing approach seems like a natural fit.

This distinction between stream processing and batch processing architecture has a large impact how the integration of ML models has to be done.

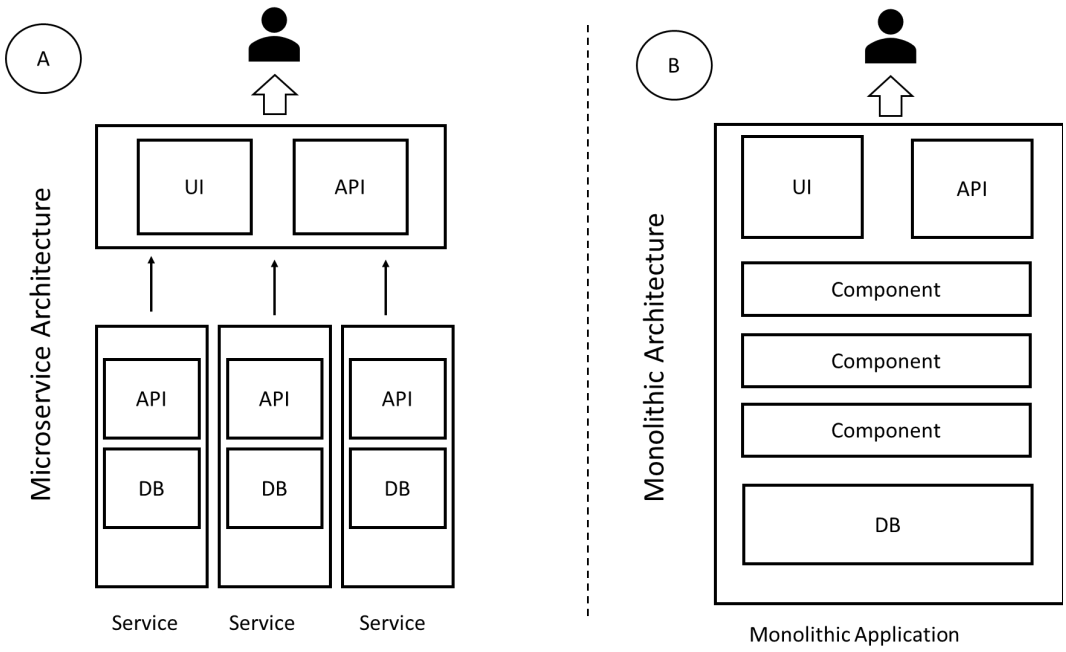

Figure 2.5: Microservice Architecture vs Monolithic Architecture.

Beside that, the structure of the existing application determines how difficult the integration will be. For example the microservice architecture pattern encourages exchangeable and extensibility of components, whereas a monolithic architecture might be required to transition into smaller independent components.

Figure 2.5 shows the difference between a Microservice Architecture (A) and a Monolithic Application (B).

In both architectural styles the client interacts with the system the same way via an application programming interface (API) or user interface (UI). From a deployment perspective, microservices have the advantage that they can be shipped faster since they are smaller and less complex than the whole application. However, an advantage of the monolithic application is that the deployment of the whole application might be less error prone since the communication between the components is done only within one application.

2.2.5 Frequent Deployments

A proven best practice in software development is to deploy frequently. The ability to perform frequent deployments requires the tooling and test environment to be automated as much as possible. The automation is essential to deploy fast and fast deployments are the basis for frequent deployments. A lack of automation here is dangerous because every manual step is not only cost intensive but also error-prone and hence decreases quality. In traditional software development new deployments are necessary when new functionalities should be shipped or when bugs have been fixed. New deployments of ML models are not only necessary for new prediction tasks. Over some time the deployed models can become less accurate because the assumptions made in the training phase do not reflect the current data distribution (concept drift). The model needs to be retrained with new data, although the actual training code might be the same as in the previous model version.

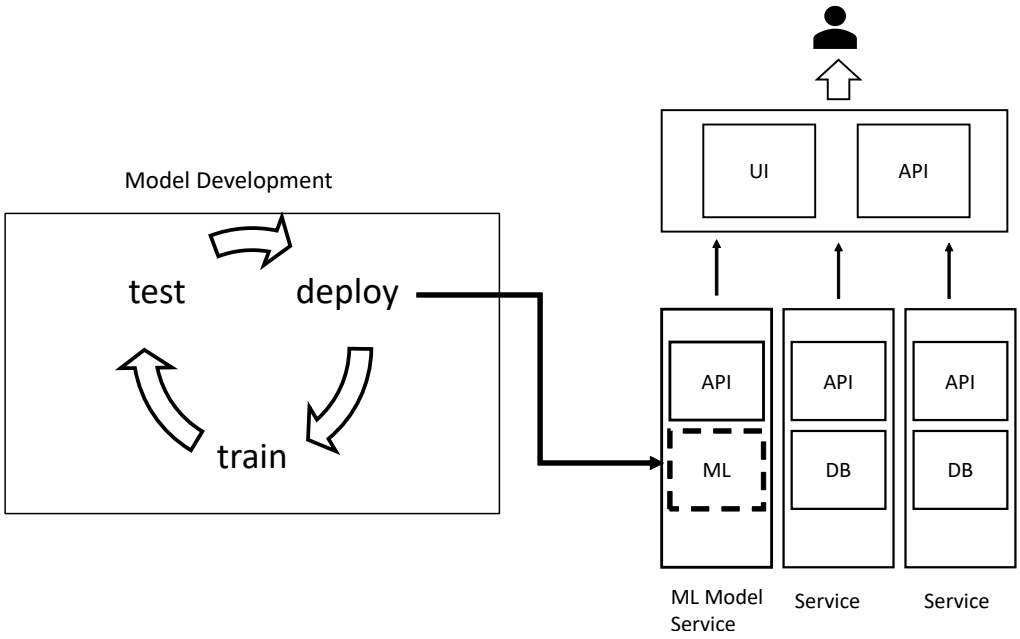

Figure 2.6: The model deployed as a service.

Like described in Section 2.2.4, the system architecture determines how

simple and fast a new deployment can be done. In the simplest deployment scenario, the model service (microservice) is completely separtated from the surrounding software system, where all the other code (feature calculation, communication protocol) remains the same as in the previous deployment of the model service. Figure 2.6 illustrates such a deployment scenario. If the data gathering for a new training run is automated and the new trained model performs better than the previous one, the system gets better or at least keeps up to date without manual interaction and new development work.

The deployment complexity is also influenced by the number of models. It might be beneficial to have multiple models deployed for the same business case. For example, before releasing a new model to production, the model candidate can first be deployed as a ”shadow model”. A ”shadow model” receives the same incoming data as the currently released model and if the ”shadow model” performs well it can become the new live model in production. Additionally it can make sense to have a very simple but understandable and trustworthy model deployed in shadow mode. This simple model can be a rule-based algorithm and does not require to be created with machine learning techniques.

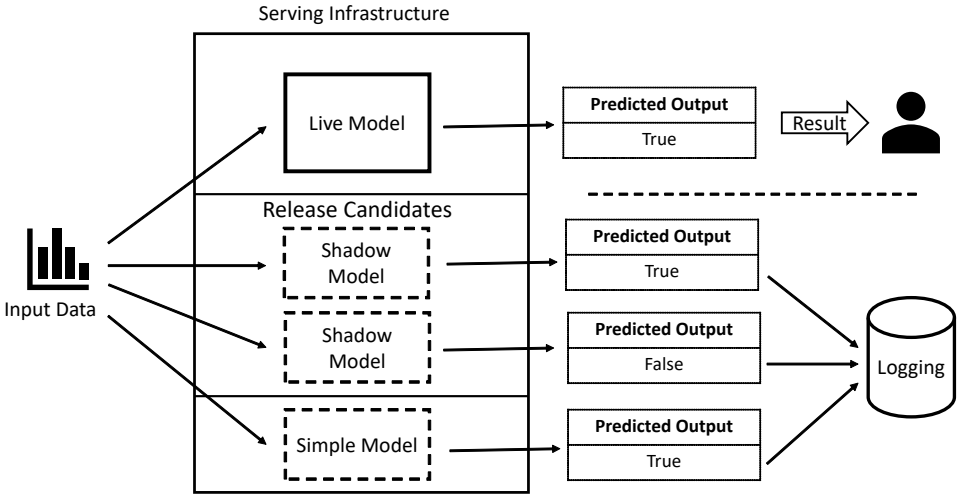

Figure 2.7: Deployed Models.

Figure 2.7 illustrates the deployment of one live model, delivering the prediction results to the user and additionally three shadow models where two of them are release candidates and the third one acting as a simple model which is not considered to be released for real predictions.

2.2.6 Model Definition

There are different viewpoints what a ML model is [20] and what belongs to it. From a mathematical perspective a model is described by the parameters obtained after training. From a technical perspective a model can be understood as a "black-box" with defined inputs and outputs, providing a predict-function like in Figure 2.8.

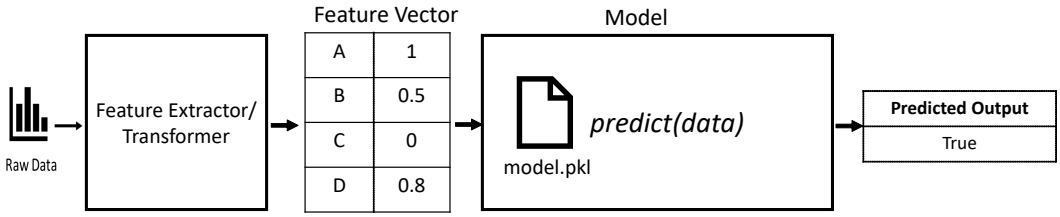

Figure 2.8: The model as a "black-box".

It is a technical task to persist the model and make it accessible to the software system. The inputs (features) for the models need to be calculated which can include several transformation steps and in more complex cases the output of a model can be the input of other models. For a successful integration and deployment, a model needs to be seen as persisted mathematical representation including all the feature calculation steps.

2.2.7 Model Debugging and Interpretability

To see the model as a black box like in the previous Section 2.2.6 might be sufficient when thinking about the integration of the packaged model into the software system. When running the model in production or for analytical purposes, the question can occur why and how the model calculates a specific prediction result for a given input (Figure 2.9).

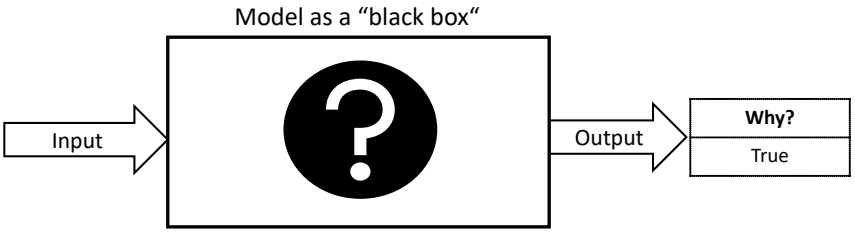

Figure 2.9: How can the prediction be explained?

Other than traditional rule-based algorithms, there are no instructions of the source code which can be debugged. The situation is similar like only having the binary of a compiled software program or library where the original source code was lost and for insights the binary needs to be disassembled.

Reproducible model training (see Section 2.2.2) is only one part to understand the model, but can be used to detect errors due to bad training data. Explainability or interpretability goes beyond debugging errors. Sometimes the accuracy of the model is sufficient and it would be interesting to know how the model can give such good predictions. Checking the causality of the features may give reasonable insights. Additionally in some domains regulations require the ”Right for Explanation” [25].

The learning algorithm is a critical factor when developing for explainability. Choosing the learning algorithm is a trade-off between good enough prediction accuracy and explainability. In [26], they analyzed different algorithms for explainability in the subject of software analytics (Figure 2.10). It can be seen, that simple models (Decision trees, Linear Regression) should be preferred when they are accurate enough and explainability is desired.

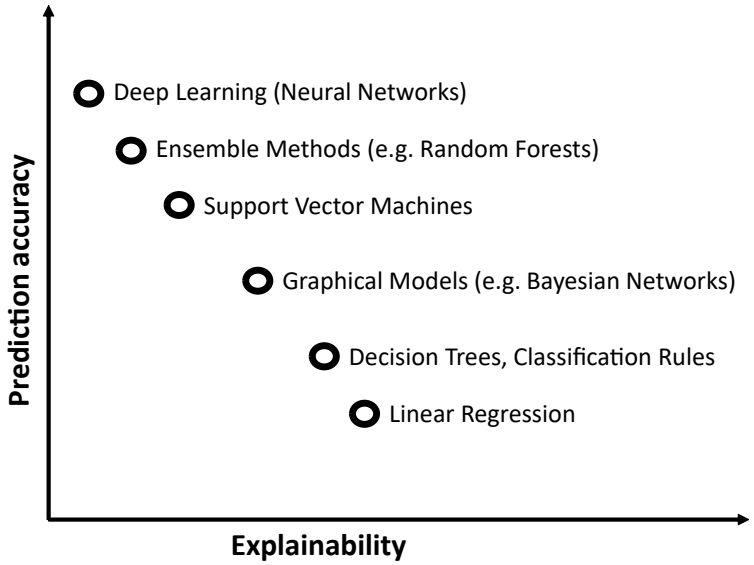

Figure 2.10: Prediction accuracy vs Explainability, Source: [26].

2.2.8 Training-Serving Skew

In the training phase and serving phase, the same transformations have to be done to get the same results. It is a common problem to get reduced prediction quality due to slightly different transformation steps in the prediction pipeline [27]. It is a good advice to reuse the feature calculation code. If there are multiple programming languages involved (see Section 2.2.3) reusable code is much harder to achieve.

Figure 2.11 illustrates an abstract example of training-serving skew. The right side of the picture shows the data flow at serving time. The serving infrastructure retrieves **A B C** as input. After transformations of **A B C** the feature vector for the prediction function of the deployed model is **1 2 3** at serving time which leads to the predicted output of ***0***.

On the left side (training phase) the raw data **A B C** is transformed to **0 1 2** and labeled with ***1***. Although the model itself from the training phase was deployed correctly to the serving infrastructure, here is a training-serving skew due to deviations in the transformation step. This off-by-one transformation error seems obvious and easy to detect in this simple

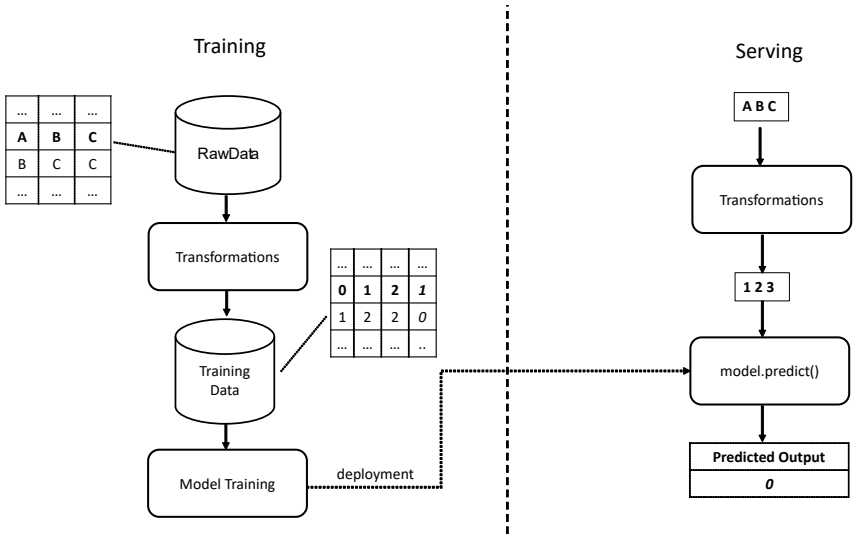

Figure 2.11: Training-Serving Skew. Based on [28].

example, however in a real example where multiple versions of the same model type are deployed it might not be easy to detect at first. It can also be hard to resolve when different models need different versions of transformation code. In [27] it is suggested to log the features calculated at serving time and use them later for further model training to avoid training-serving skew. This requires a suitable monitoring infrastructure, what is described in the next Section (2.2.9).

2.2.9 Monitoring

Monitoring in the context of software operations means observing a running software system. With traditional software systems, monitoring should answer the following technical questions:

- Is the system running or down?
- Which exceptions occurred?
- What is the latency, throughput and resource consumption of the system?
- Are there any anomalous requests?

- Are there bottlenecks in the system infrastructure or system architecture?

Additionally to those technical questions monitoring ML systems adds further requirements. In the prediction phase the main monitoring question is: Does the model still perform the same as evaluated in the training phase? This question can be hard to answer in a fast and reliable way. At training time there are clear metrics like precision, recall and F1-score to evaluate the model. To calculate those metrics, knowledge about the real prediction result is needed. For example in the problem domain of this project, the farmer wants to be alerted if a cow is in heat, what means a high chance of pregnancy of the animal after an insemination. The problem is, that it takes about forty days after the insemination to have certainty if the cow is pregnant or not and only after this time the prediction result can be confirmed. Even if there are many examples where the prediction result can be verified faster it explains that monitoring in the prediction phase requires additional or other metrics than for model evaluation in the training phase.

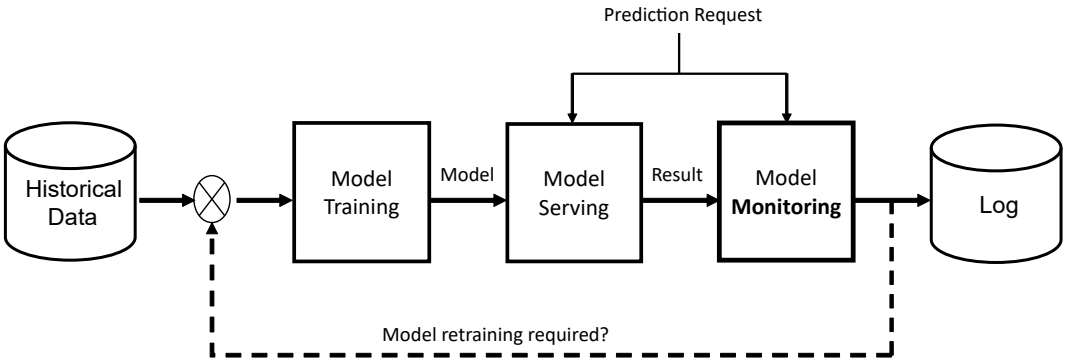

Figure 2.12: Monitoring ML Systems.

Figure 2.12 illustrates the monitoring phase in the ML lifecycle. The monitoring component collects the prediction requests and prediction results from the models and stores them for further analysis. After some time the it should be clear if a retraining or remodelling is required. For retraining and remodelling the new logged data can be added to the historical data.

2.3 Maturity of Machine Learning Systems

Compared to traditional software development, ML in production is a rather new field and the processes are not that mature. Agile software projects encourage short iteration cycles allowing the shipment of new features in frequent intervals to the customers. To solve this technical and organizational challenge the development operations (DevOps) [29] methodology was introduced. On a similar basis but also considering the particularities of ML, the Machine Learning Operations (MLOps) [30] methodology came up.

To identify gaps in an organization attempting to operate a successful ML environment Microsoft defined 5 levels (0-4) known as the Machine Learning Maturity Model [31]. These levels are:

- **Level 0:** *No MLOps* - Basically everything needs to be done manually (manual builds and deployments, manual testing, manual training, no centralized model performance tracking)

- **Level 1:** *DevOps but no MLOps* - at least DevOps best practices for the non-ML parts are established (automated builds and tests for application code)

- **Level 2:** *Automated Training* - reproducible and managed training environment, automated model training and performance tracking

- **Level 3:** *Automated Model Deployment* - full traceability from the deployed model back to original data, automated tests for all code, entire managed environment for training, testing and production

- **Level 4:** *Full MLOps Automated Operations* - full system automated and monitored. The production system can automatically detect improvements and optionally can deploy new models with zero down-time.

Another paper from Google [32] described the production readiness of ML systems by calculating the "ML test score". Additionally to the previous mentioned ML maturity model, the ML test score concentrates on what to test and monitor in more detail.

The ML Test score describes four test categories:

- **Tests for features and data** Traditional software is tested with unit tests and integration tests. When using ML this approach is not sufficient, also tests concerning the data are necessary like the distribution of the data. But also the calculation of the features belong to a model and should be tested.

- **Tests for model development** This category describes a list of considerations for model development. For example checking if a model exceeds the performance of a simple baseline model.

- **Tests for ML infrastructure** The ML infrastructure is a complex pipeline and not only a single running program. Therefore it makes sense to do integration tests for the whole pipeline but also unit tests for model specific code. Complementary it should also be tested how fast a model can be reverted and rolled back to a previous version in production.

- **Monitoring tests for ML** Monitoring is an important component when running ML in production. The monitoring component should check for data assumptions of the inputs and outputs of the model but also technical performance metrics like latency and throughput.

3 Existing Technologies and Solutions

In the field of software engineering the landscape of technologies, tools, languages and methodologies has been evolving a lot. Most professional projects use version control, automated testing, continuous integration and recently also containerized deployments.

Adding software components, that are highly influenced by data and developed in a different way than traditional software, increases the complexity and the tooling requires adaptations. This chapter is about existing (third-party) solutions and products for managing the ML lifecyle with a focus on the deployment and serving phase.

The first three sections describe basic technologies for different challenges of the ML lifecycle. The last section describes solutions for managing the whole (end-to-end) ML lifecycle.

3.1 Model Persistence

After the training phase, the model needs to be saved (persisted) for further usage without retraining. The problem is, that there is no standardized approach and the different ML frameworks use different data formats. Not only the data formats vary, also the programming languages of the training system and the prediction system can differ (see Section 2.2.3). The problem already starts when different versions of languages and frameworks are used. But the situation is similar in cross-platform software projects consisting of several components and third-party libraries. The following describes a few solutions to persist ML models.

3.1.1 Python Pickle

The Python Pickle module[1] provides a simple interface for serializing and deserializing Python code. This is a very simple but effective approach (Listing 3.1), especially like in this project because the primary language of the system is Python.

Listing 3.1: Serialization and deserialization of scikit-learn models as python pickle files.

```
from sklearn import svm
import pickle
clf = svm.SVC()
X, y= load_dataset()
clf.fit(X, y)

# serializiation
pickle.dump(clf, open("model.pkl", "wb"))

# deserialization
clf2 = pickle.load(open("model.pkl", "rb"))
result = clf2.predict(X[0:1])
```

Scikit-Learn recommends to use joblib[2] for persisting models[3]. Listing 3.2 shows the dump/load interface of joblib.

Listing 3.2: Serialization and deserialization with joblib.

```
from joblib import dump, load
dump(clf, 'model.joblib')
clf = load('model.joblib')
```

A problem of pickle and joblib is the incompatibility of different versions of scikit-learn (and probably many other libraries) what can lead to unexpected results.

[1] https://docs.python.org/3/library/pickle.html
[2] https://joblib.readthedocs.io/en/latest/persistence.html
[3] https://scikit-learn.org/stable/modules/model_persistence.html

3.1.2 Predictive Model Markup Language

Predictive Model Markup Language (PMML) is an XML-based data format for predictive models. The first version was already released in 1997. The importance of PMML as a de-facto standard for model archiving was discussed in [33]. PMML is not only capable of saving the model, but also specifying information about the feature types (categorical, continuous, ordinal, ...) is possible. Even data transformation pipelines for preprocessing (normalization, discretization, custom functions, ...) and post-processing can be described. The big advantage of PMML compared to simple solutions like the Python Pickle is the interchangeability between programming languages. There exist many tools and libraries to export PMML files. Listing 3.3 shows the code to export PMML files of a scikit-learn model.

Listing 3.3: Exporting a scikit-learn model to PMML.

```
from sklearn2pmml.pipeline import PMMLPipeline

pipeline = PMMLPipeline([
        ("classifier", DecisionTreeClassifier())
])

# pipeline.fit(...)

from sklearn2pmml import sklearn2pmml
sklearn2pmml(pipeline, "DecisionTree.pmml",
with_repr = True)
```

A disadvantage of PMML models is that it does not support some types of models and advanced methods like Online learning (models can update themselves at runtime) [34].

3.1.3 Open Neural Network Exchange

Open Neural Network Exchange (ONNX)[4] is a rather new interchange format for ML models with a focus on deep learning models. ONNX gained

[4] https://onnx.ai/

popularity due to the support of Microsoft and Facebook announced in 2017 [35]. An advantage of ONNX are the existence of hardware accelerated scoring engines. This acceleration can lead to faster inference times of the deployed models. One drawback of ONNX is that due to its novelty not all traditional model types are supported yet. A list of supported model types for scikit-learn can be found here[5].

3.1.4 SavedModel

In contrast to the rather library agnostic PMML and ONNX formats, the SavedModel is a specific file format for storing TensorFlow models. The SavedModel format works without third-party extensions for TensorFlow models via the SavedModel API[6].

SavedModel is a good option if it is clear that only Tensorflow/Keras is used for model development and the TensorFlow-Serving[7] or the Google Cloud Platform[8] can be used for serving.

3.1.5 Docker

Docker is per se not a technology for model persistence or data science at all. Docker is compared to virtual machines a light-weight virtualization solution for isolated applications. Docker container share the same operating system whereas virtual machines need to start a whole operating system for each running instance. Docker users need to distinguish between the following concepts:

- **Dockerfile:** A Dockerfile is a text file describing step by step the commands to build an executable docker image.

[5]http://onnx.ai/sklearn-onnx/supported.html
[6]https://www.tensorflow.org/api_docs/python/tf/saved_model
[7]https://www.tensorflow.org/tfx/guide/serving
[8]https://cloud.google.com/ai-platform/prediction/docs/deploying-models

- **Image:** The image is the result after executing the building steps described in the Dockerfile. It is possibly to load and share docker images via a docker registry.

- **Container:** This is the term for a running instance of an image. There can be multiple instances of the one image running at the same time.

Listing 3.4: Example of a Dockerfile for python scripts.

```
FROM ubuntu:20.04

RUN apt-get update -y && \
    apt-get install -y python-pip python-dev

COPY ./requirements.txt /app/requirements.txt
WORKDIR /app
RUN pip install -r requirements.txt
COPY . /app

CMD [ "python", "app.py" ]
```

Listing 3.4 shows a Dockerfile describing the steps beginning from an Ubuntu 20.04 image (FROM command) to executing a python script. RUN executes commands inside the container, COPY copies files from the system running docker to the container. WORKDIR defines the working directory for further instructions in the container. The last command *CMD ["python", "app.py"]* is the default command executed when running the image as a container. All the other commands in this example are commands for building the image.

Docker is not only useful for distributing modern web applications. In the context of ML there are three important use cases [36]. The first one is **reproducibility** as already described in Section 2.2.2. Docker was also discussed as a tool for reproducible research [37]. At second, it is **portability**. This means it is very easy to run the same image on different machines. And most important for productionizing ML models is the **ease of deployment**. For example Python pickles packed can trouble-free deployed packaged with docker. Without docker it is very error prone to have all the right dependencies installed ("dependency hell").

3.1.6 MLflow Models

MLflow Model[9], part of the MLflow project (see Section 3.4.1), is a packaging format supporting persisted models of several ML libraries. In the context of MLflow the support of different libraries are called *flavours*. MLflow defines several "standard" flavors that all of its built-in deployment tools support. At the time of writing this theses MLflow supports the following built-in flavours[10]:

- Python Function
- R Function
- H2o
- Keras
- MLeap
- PyTorch
- Scikit-Learn
- Spark
- TensorFlow
- ONNX
- MXNet
- XGBoost
- LigthGBM
- Spacy
- Fastai

The "Python Function" flavour provides a default interface for loading persisted models with MLflow. Models loaded as a "Python Function" can be scored with the same prediction function, receiving a Pandas DataFrame[11]:

predict(model_input: pandas.DataFrame) $\rightarrow$ *[numpy.ndarray | pandas.(Series | DataFrame)]*

Furthermore, MLFlow models can easily deployed and served on various platforms (Azure, Sagemaker, Spark) or as a docker container exposing an API endpoint.

[9] https://www.mlflow.org/docs/latest/models.html
[10] https://www.mlflow.org/docs/latest/models.html#built-in-model-flavors
[11] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html

3.2 Version control

Version Control Systems (VCS) are probably used in almost all professional software projects for tracking source code changes. Git[12] is a de facto standard encouraging an agile development workflow with fast product iterations.

VCS like Git work great for rather small source files or other text files, but have limitations for large and/or binary files. ML projects are built with different kinds of files. The training code consists of normal source files, but the data required for training is usually stored in rather large (binary) files. Additionally, there is the model as the output of the training step and other metadata and metrics. All of these files and data is required for reproducibility (see Section 2.2.2).

3.2.1 Data Version Control (DVC)

DVC[13] is a free Git-based version control system especially designed for data science projects. Similar to Git, DVC is used via the command line and many commands are basically the same (dvc pull/push/checkout). Behind the scenes there are some differences. Large files (data and models) are not directly stored in the repository, they are replaced with small meta files pointing to the original large files. Those small files are easy to handle for git, the large files can be stored somewhere else. The most important DVC functionalities are:

- Git-compatible version control
- ML pipeline framework: Reproducible Steps for building ML models can be defined
- Metric tracking: Tracking metrics in JSON files and comparing metrics in different branches
- Storage agnostic: Large files can be stored at self-hosted systems or thrid-party cloud storage (eg Amazon S3).

[12] https://git-scm.com/
[13] https://dvc.org/doc/user-guide/what-is-dvc

3.3 Model Serving

Model serving is the main task after persisting the model. For many use cases it is sufficient to load a pickled model and serve it via a HTTP using lightweight frameworks like Flask[38]. This Section names more sophisticated third-party solutions for model serving without claim for completeness.

Basically it can be distinguished between managed hosted cloud services and self-hosted model servers. The major platforms (SageMaker/Amazon, Azure/Microsoft, Google Cloud AI Platform) support to upload the model and expose an API Endpoint for scoring (predictions). The communication protocol for scoring is HTTPS or gRPC (gRPC Remote Procedure Call). Alternatively when large amounts of data instances should be scored and/or when higher latency of of the prediction results is not a problem, batch predictions can be used (GCP: [14], SageMaker:[15], Azure: [16]).

Those platforms are under heavy development and new features appear very often, however before choosing one platform it should be checked if the model persisting format is supported. As a fallback the model can be served via a docker container, what all major platforms are capable of.

3.3.1 Self-hosted Model Servers

Openscoring[17]: This is an opensource implementation of REST Service for PMML (see Section 3.1.2) models written in Java. JPMML-Evaluator[18] is the PMML Runtime. The simple REST API exposes multiple endpoints (Table 3.1) for model management (deployment / undeployment, meta data information) and prediction requests (single and batch).

[14]https://cloud.google.com/ai-platform/prediction/docs/online-vs-batch-prediction
[15]https://docs.aws.amazon.com/sagemaker/latest/dg/how-it-works-batch.html
[16]https://docs.microsoft.com/en-us/azure/machine-learning/how-to-use-parallel-run-step
[17]https://github.com/openscoring/openscoring
[18]https://github.com/jpmml/jpmml-evaluator

HTTP method	Endpoint	Description
GET	/model	Get the summaries of all models
PUT	/model/${id}	Deploy a model
GET	/model/${id}	Get the summary of a model
GET	/model/${id}/pmml	Download a model as a PMML document
POST	/model/${id}	Evaluate data in "single prediction" mode
POST	/model/${id}/batch	Evaluate data in "batch prediction" mode
POST	/model/${id}/csv	Evaluate data in "CSV prediction" mode
DELETE	/model/${id}	Undeploy a model

Table 3.1: Openscoring API Endpoints.

ONNX Runtime:
Since Microsoft open sourced the ONNX Runtime [35] it is possible to use the ONNX format for self hosted solutions. By the time of writing this thesis, the ONNX Runtime Server[19] is still marked as beta, what means without wanting to spend too much effort it is recommended to deploy ONNX models on Azure.

Tensorflow Serving:
This is the serving component of TensorFlow Extended (TFX)[20]. TensorFlow Serving is a high-performance serving system for ML models. As the name suggests, only TensorFlow models are supported without extensions. According to the project's documentation, TensorFlow Serving can easily be extended with other model formats. However, adding other model formats requires extending and recompiling the TensorFlow Server written in C++.

MLflow Serving:
As already described in in Section 3.1.6, MLflow models support models created with various ML frameworks. They can not only be deployed to third-party solutions (Azure, SageMaker) but also be served via an integrated Flask-based web server.

[19]https://github.com/microsoft/onnxruntime#deploying-onnx-runtime
[20]https://www.tensorflow.org/tfx/guide/serving

3.4 End-to-End Machine Learning Platforms

The previous sections described technical solutions for parts of the ML lifecycle. Of course, those solutions can be glued together to create an End-To-End (from data gathering to deployment) solution covering the own requirements. Many companies tried to manage the challenges (see Section (see Section 2.2) of the ML lifecycle by building own ML platforms. One of the first publicly announced examples was Uber's platform "Michael Angelo" [16]. An overview of well-known End-to-End ML platforms can be found here [39]. The following section describes the functionalities of the open-source solution MLflow.

3.4.1 MLflow

MLflow, a software by the company Databricks, was introduced [40] to tackle the challenges (see Section 2.2) of the whole ML lifecycle. MLflow is not a ML library implementing algorithms for model training.

MLflow comes shipped as a python package [21] and therefore can be installed with a single command on a developer machine.

MLflow users interact with the software in different ways. The first one is the MLflow command-line tool (mlflow), the second option is the MLflow web interface (mlflow ui, for viewing experiments see Figure 3.1) and the third one is MLflow as a python library (import mlflow). Additionally MLflow can be controlled via a REST API, so it can also be used with other languages than Python.

The MLflow functionalities can be grouped into the following components:

- **Tracking:** The tracking component provides a set of functions for logging parameters, metrics and other artefacts like diagrams. It can be specified (MLFLOW_TRACKING_URI environment variable) where the logged data should be stored. The most simple case is running

[21] https://pypi.org/project/mlflow/

MLflow only on a developer machine without a tracking server. This setup can be compared with using the Git version control system without a remote repository. The tracked experiments can be viewed with the MLflow web interface what is especially useful when multiple developers work on the same model.

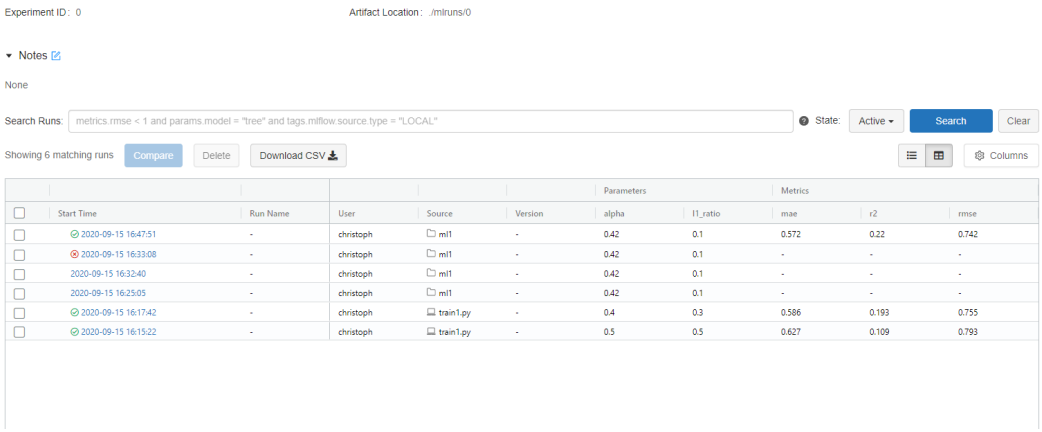

Figure 3.1: MLFlow Web Interface - Overview of the experiments.

- **Projects:** MLflow projects is a data format for creating reusable ML development environments. It is also possible to use MLflow without using the projects format and instead using other tools like virtualenv[22] and build scripts like make[23], however the project component does perfectly fit into the MLflow ecosystem.
  MLflow projects are described via a YAML file named MLProject (see Listing 3.5). The code dependencies of the project can be described via Conda[24] or Docker environments. Docker environments have the advantage that they can be used with non-Python dependencies (eg Java libraries) and other Linux specific dependencies. Alternatively it is also possible to only use the dependencies of the system environment.

[22]https://packaging.python.org/key_projects/#virtualenv
[23]https://www.gnu.org/software/make/
[24]https://docs.conda.io/projects/conda/en/latest/index.html

Listing 3.5: Example of a MLProject file.

```
name Project1
conda_env: conda.yaml
entry_points:
  main:
    parameters:
      alpha: float
      l1_ratio: {type: float, default: 0.1}
    command: "python train1.py {alpha} {l1_ratio}"
  validate:
    parameters:
      data_file: validate.csv
    command: "python validate.py {data_file}"
```

Listing 3.6: Conda File.

```
name: Project1
channels:
  - defaults
dependencies:
  - numpy=1.14.3
  - pandas=0.22.0
  - scikit-learn=0.19.1
  - pip:
    - mlflow
```

Listing 3.5 is an example MLProject file declaring the dependencies with a conda file (Listing 3.6). The project file defines two entry points (main and validate) and parameters for those commands. MLflow projects can be run via the command-line with the "mlflow run" command. Per default the main entry point will be used for execution. The big advantage of using the MLflow project description is that the projects can be executed with one single command covering the download from Git to the actual execution in a reproducible environment.
Execution from version control (git):
mlflow run https://github.com/mlflow/mlflow-example.git -P alpha=0.5
Execution from the local file system (in the current directory needs to

be a MLProject file):
mlflow run . -P alpha=0.5
Due to this simple execution commands, MLflow can easily be integrated in CI/CD pipelines.

- **Models:** The models component was already described in Section 3.1.6 as a general packaging format supporting several ML libraries. Additionally, it is also possible to create custom models derived from mlflow.pyfunc.PythonModel implementing the predict method. The following Listing 3.7 shows the implementation of a custom model without using any ML algorithms returning random values in the predict method.

Listing 3.7: MLflow Custom Model.

```
import mlflow.pyfunc
from random import randrange

class CustomModel(mlflow.pyfunc.PythonModel):
  def __init__(self, maxvalue):
      self.maxvalue = maxvalue

  def predict(self, model_input):
      return model_input.apply(lambda column:
        column + randrange(maxvalue))

#Saving the model
model_path = "custom_model"
custom_model_model = CustomModel(maxvalue=20)
mlflow.pyfunc.save_model(path=model_path,
    python_model=custom_model)

# Using the model
loaded_model = mlflow.pyfunc.load_model(model_path)
import pandas as pd
model_input = pd.DataFrame([range(10)])
model_output = loaded_model.predict(model_input)
```

The example above does not only demonstrate the implementation,

saving and loading of custom models it should also make clear that using MLflow can also make sense for traditional non-ML algorithms. For testing purposes saved models can used for predictions via the "mlflow models predict" command, but is also possible to serve the model via HTTP using the "mlflow models serve" command. For production-ready deployments it is possible to create docker containers (command: "mlflow models build-docker") or deploy the model to cloud services.

- **Model Registry** The Model Registry is the component for managing the lifecycle of a model accessible via the MLflow web interface (Figure 3.2) and API. Using the Model Registry is optional and the newest component of MLflow, however the registry is a suitable component for keeping track of models in different stages. Per default, the registry organizes the models in the stages **None**, **Staging**, **Production**, and **Archived**. These are common deployment stages, also for other web services. The MLflow API supports model registration in several

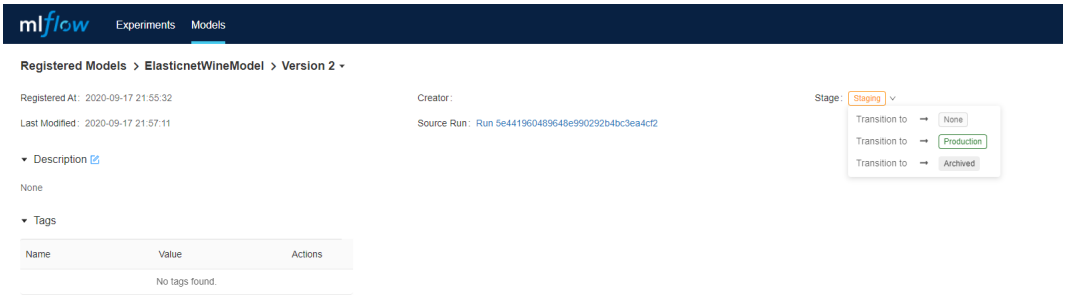

Figure 3.2: MLflow Model Registry.

ways.
One simple solution is to pass the *registered_model_name* parameter to the log_model() function what makes sense when the model should be saved and registered after a training run. Another common way is to use the register_model() function to register a specific training run.

Listing 3.8: Loading a model from the registry.

```
import mlflow.pyfunc

model_name = "example-model"
stage = 'Staging'

model = mlflow.pyfunc.load_model(
    model_uri=f"models:/{model_name}/{stage}
)

model.predict(data)
```

Once a model is registered it can easily be loaded using the load_model() function (Listing 3.8). Only the name of the model and the stage or version needs to be specified. This is extremely useful when decoupling of a software system and the actual deployed models is required. It is also possible to serve a model directly from the model registry using the MLflow command-line tool (Listing 3.9).

Listing 3.9: Serving a Model via the MLflow command-line tool.

```
#!/usr/bin/env sh

# link to the MLflow Model Registry
export MLFLOW_TRACKING_URI=http://localhost:5000

# Serve the example-model via MLflow command line tool
mlflow models serve -m "models:/example-model/Staging"
```

4 Architecture and Concept

This chapter can be understood as a case study of productionizing ML models in the context of the system architecture of smaXtec. The existing system architecture and possible adaptations are discussed.

4.1 Requirements

The proposed implementation in this chapter does not make a claim to completeness of a finished solutions. Rather it should be demonstrated if the existing software architecture can be extended with ML models and which adaptations are necessary. Furthermore, integration and deployment of ML models is more a repetitive process than a software solution. Despite that, the proposed solution should support the automation of the deployment tasks as much as possible. Understanding the requirements of a ML system is one of the main goals of this project. Figure 4.1 illustrates the requirements gathering approach.

The requirements are derived from company internal sources (the specific business needs and existing solution) and external sources (best practices, technologies and case studies from other projects). By analyzing these inputs, adaptations to the existing architecture and development processes can be done. The gathering of the requirements is an iterative process. This master thesis can be seen as the initial iteration to get a first prototype to the production phase. Having a prototype solution in production, or at least in a separated staging area, should deliver experience over time and the requirements can be refined.

From a technical point of view, the specific business needs of a company define the requirements concerning latency and scalability. The question is

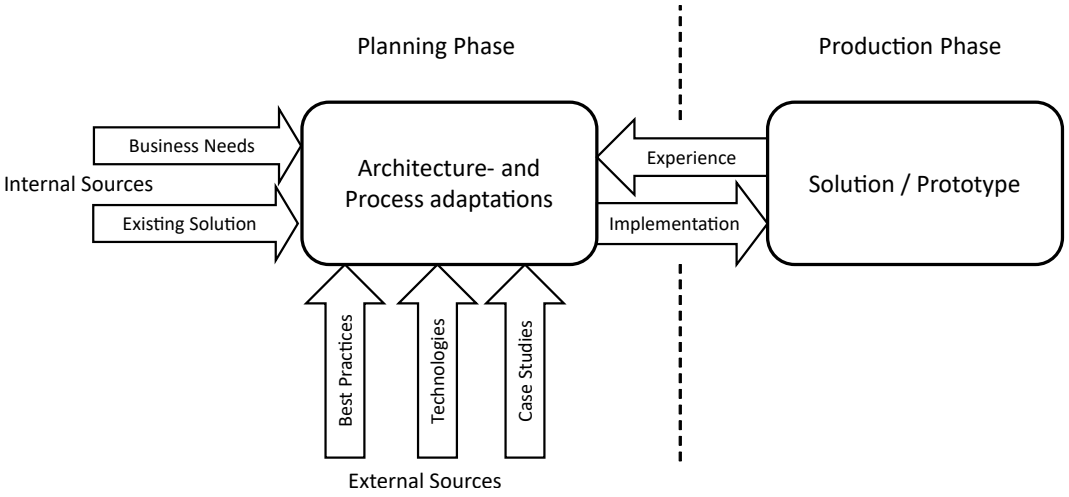

Figure 4.1: Requirements Gathering.

what is the latest time when a prediction result needs to be delivered to the end user or for further processing? Is the system capable to guarantee the processing times also for large amounts of data? From a legal perspective and to increase the confidence of the customers, are there special requirements or regulations to explain and communicate the prediction results? Business specific constraints also determine the number of different models the whole system needs and the type of the prediction tasks.

Obviously it is important to know the own business needs, although without hardly any further know-how of ML in production the requirements need to be completed with the experience from best-practices of other projects and documented case studies. Knowledge about the current state of the art technologies should avoid the development of already existing and affordable tools but also to know the limits of what is possible by now. For example, there exist many technologies to package or containerize code, what makes the sense of an own implementation questionable. On the other hand there are (cloud-)solutions to serve ML models, but for that purpose an own implementation can reduce the dependence on third-party solution providers. For some requirements like explainability it may be that there are no sufficient solutions available yet and the development would be an own research topic, what means such requirements can not be completely

fulfilled in reasonable time.

The following is a short description of the functional requirements, the components and features of the implementation and the non-functional requirements, the quality constraints which should be fulfilled.

- Functional requirements
  - **Feature extraction**: The system must be capable of extracting the input features as input for the model.
  - **Model deployment**: The deployment step is about packaging the trained model into a form that other components or applications can request predictions.
  - **Model serving and integration**: This requirement is about the communication between the model and the rest of the system. A suitable method to serve the model should be worked out. Before requesting predictions from the model, all the necessary input data needs to be calculated.
  - **Model monitoring**: A basic monitoring approach to analyze the predictions of each model should be found.
  - **Model Versioning**: It should be possible to have multiple versions of models. This includes the versioning of the training code with the evaluation metrics and also to have multiple versions side by side in the production phase.
- Non-functional requirements
  - **"Real-time" predictions**: The usefulness of a prediction is highly depended on the time, because the user who is notified by the system with a prediction result needs enough time to take action. A prediction is worthless for the user when the notification comes too late it is only interesting for historical reasons. To support "real-time" predictions it is not only necessary that the model has fast response times, additionally all input features need to be calculated and provided as fast as possible.

- **Scalability**: The system should be capable or extendable to support fast (”real-time”) predictions even for large data sets.

- **Reproducibility**: It should be reproducible which models were the source of generated events including the input data.

- **Frequent deployments**: The system architecture and tooling should encourage fast deployment cycles of new models and the frequent updating of already deployed models. This can be summarized with the properties maintainability and extensibility.

Section 2.2 described the challenges of ML systems by summarizing related work. Finding solutions for those challenges or analyzing the situation in that particular project led to the requirements named here.

4.2 Existing System Architecture

This project did not start from scratch, it is the extension of an existing commercial software solution already in production phase. Besides the specific business needs, the existing system architecture determines how much effort the integration of ML components will cause. Section 2.2.4 is a brief introduction about the categorization of a software architecture relevant for this project. The main distinguishing characteristics were about data processing (batch- and stream processing) and the composition of services and components (Microservices or a rather monolithic architecture).

4.2.1 High-level Perspective

Without going too far into depth Figure 4.2 shows the main building blocks of the smaXtec system architecture.

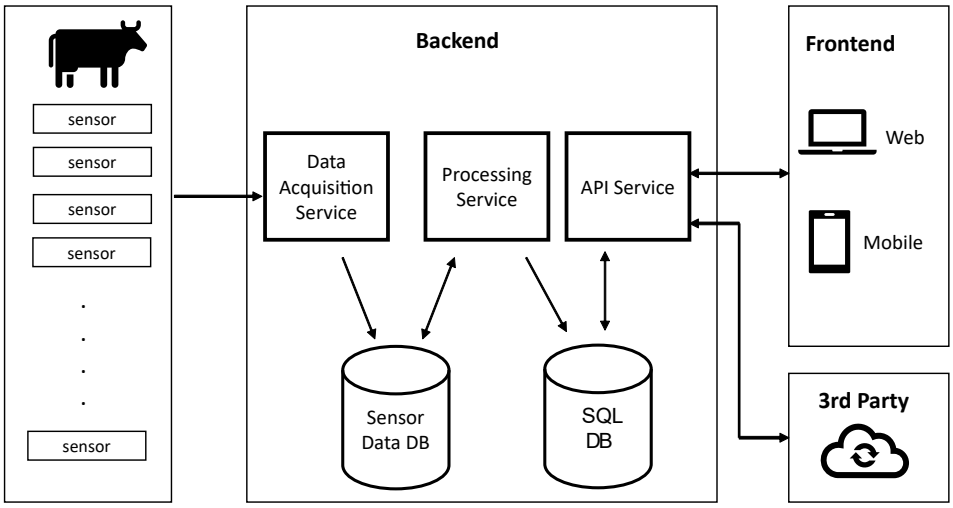

Figure 4.2: High-level perspective of the existing System Architecture.

The smaXtec system can be seen as an Internet of Things (IoT) application where cows are equipped with sensors (boluses) monitoring activity-, ph- and temperature data. Additionally, there are external climate sensors to measure temperature and humidity. The technical details like the data transmission of those devices are not important for this project. It only needs to be considered that due to limitations (battery lifetime and computing power) of the sensors, most of the processing has to happen on the server side. Moreover, cows are often out of reach of a basestation or repeater, where the transmission to the server is done and hence the server receives the measured data delayed for further processing.

The backend (server side or cloud solution) consists of three basic services from a logical perspective. First, a *Data Acquisition Service* which is responsible for receiving and storing the raw measured data. Second, a *Processing Service* for the further processing of the raw data and running algorithms. And third, an API Service providing an interface to the "outside world". Consumers of the API are the web- and mobile application providing a graphical user interface for humans, but there is also the possibility for third-party organizations for easy data integration to other systems using the smaXtec API.

From a more physical viewpoint on the architecture, the system has to manage the state of two databases. A database for storing the sensor values as time series data and a relational database (SQL DB) for all the application specific data for animals, customers or similar. Real-world systems like smaXtec have to evolve and need to be extended over time for several reasons like increasing scalability requirements and new business needs. Most of the components described here in this high-level perspective will remain for a longer time, however the approach how data is processed changed during this project was conducted. The architecture changed from a batch processing to a stream processing approach as described in Section 2.2.4. This change is relevant for this project because the ML models need to have every input feature prepared for predictions. Section 4.3 shows how the feature calculation code can be integrated into the new stream processing architecture, but before that the next two subsections will give a short introduction to Apache Kafka and the programming model with Faust.

4.2.2 Apache Kafka

Apache Kafka[1] is a software solution for stream-processing originally developed by the company LinkedIn. Without going too far into detail, Figure 4.3 gives an overview of the main terminology necessary for understanding kafka-based applications.

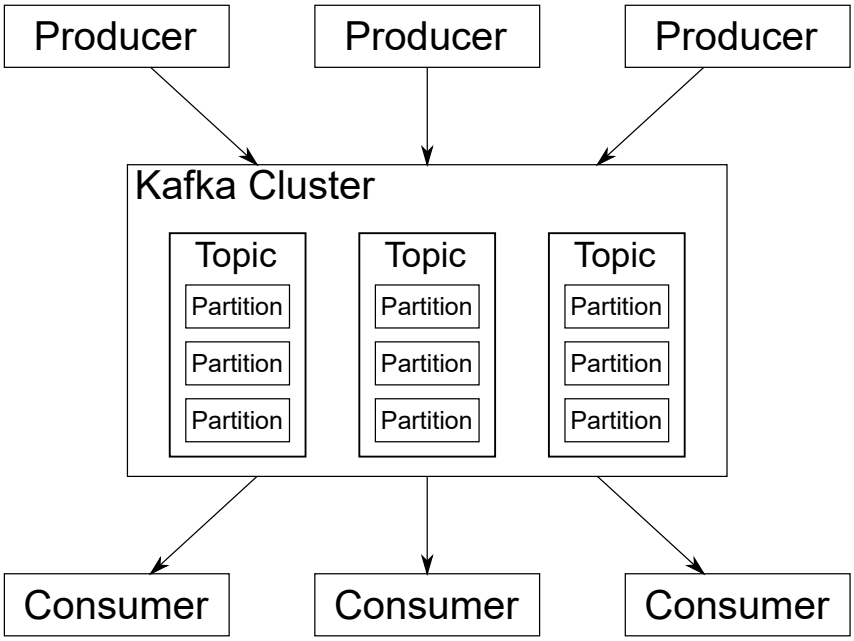

Figure 4.3: Overview of Apache Kafka, source: Wikipedia.

Basically there are producers that write to topics and consumers that read messages from topics. Kafka stores those messages as key-value pairs separated into partitions. Messages are strictly ordered within a partition. Kafka is able to run on a cluster of multiple servers (brokers) where the partitions of the topics are distributed. This cluster architecture makes kafka a fault-tolerant system for processing massive data amounts.

Listing 4.1: Kafka Producer Example.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
```

[1] https://kafka.apache.org/

```
//... set more properties here ...
props.put("key.serializer",
"serialization.StringSerializer");
props.put("value.serializer",
"serialization.StringSerializer");

Producer<String, String> producer =
new KafkaProducer<>(props);
for (int i = 0; i < 100; i++)
    producer.send(
    new ProducerRecord<String, String>("my-topic",
    Integer.toString(i), Integer.toString(i)));
producer.close();
```

Listing 4.1 shows an example of a Kafka producer from the reference implementation in java [41]. A Properties-Object is used to configure the producer. First a "bootstrap-server" needs to be specified to connect to the Kafka cluster. It is also important to define how the key and value of the message should be serialized for the transmission. In this examples hundred numbers are produced and serialized as a string.

The API for a Kafka consumer [42] is similar and can be seen in Listing 4.2. In this short example a consumer subscribes to the topic "my-topic" and prints incoming messages with a polling interval of 100 milliseconds.

Listing 4.2: Kafka Consumer Example.

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
// ... set more properties here ...
props.put("key.deserializer",
"serialization.StringDeserializer");
props.put("value.deserializer",
"serialization.StringDeserializer");
KafkaConsumer<String, String> consumer =
new KafkaConsumer<>(props);
consumer.subscribe(Arrays.asList("my-topic"));
while (true) {
    ConsumerRecords<String, String> records =
```

```
        consumer.poll(100);
        for (ConsumerRecord<String, String> record : records)
        {
            System.out.printf("offset = %d, key = %s, " +
            "value = %s%n",record.offset(),
            record.key(), record.value());
         }
}
```

The producer- and consumer API is flexible to use, although there are solutions which support a higher level of abstraction. A good trade-off between flexibility and ease of use gives the implementation of Kafka-Streams[2]. An even higher abstraction based on Kafka-Streams can be reached with the SQL-like query language KSQL [43].

4.2.3 Faust

The previous Subsection 4.2.2 describes the basic ideas of Apache Kafka and simple listings for producing and consuming messages in Java. Since most of the services of smaXtec are created with Python, there is a need to connect to Kafka using an API or framework suitable for Python with an appropriate level of abstraction. A simple implementation for Python is kafka-python[3], this library supports a similar API like the previous examples of the listings 4.2 and 4.1.

Listing 4.3: Python Kafka Consumer Example.

```
from kafka import KafkaConsumer
consumer = KafkaConsumer('my-topic',
                         group_id='my-group',
                         bootstrap_servers=['localhost:9092'])
for message in consumer:
    print("%s:%d:%d: key=%s value=%s" %
        (message.topic, message.partition,
        message.offset, message.key, message.value))
```

[2]https://kafka.apache.org/documentation/streams
[3]https://pypi.org/project/kafka-python/

The example Python consumer of Listing 4.3 reads messages from the topic "my-topic" and prints them to the terminal. It needs to be considered that without further configuration the data of the messages is interpreted as raw bytes.

In many cases a higher level of abstraction than supported by kafka-python is desired. Faust[4], an open source stream processing library by the financial services company "Robin Hood Markets, Inc" is porting the concepts of the Java/Scala implementation Kafka-Streams to Python.

The main abstraction concept of a Faust application is called **agent**, derived from the "actor model" introduced in [44] and is widely used in functional programming languages like Erlang or Scala. The actor can be understood as a primitive for concurrent computation where the actor is only allowed to directly modify its own private state to avoid lock-based synchronization. The actor processes messages and communicates with other actors by sending further messages.

Listing 4.4 shows a simple example of a faust agent. It can be seen that faust agents are asynchronous Python functions annotated with the decorator "@app.agent". The agent processes an infinite stream of events, where an event consists of a key-value pair. The result of a processing step is then send to another topic where other agents can listen on. An instance of a running faust application is called worker. This worker is a single-threaded Python process. To achieve high throughput asynchronous I/O by annotating code with the keywords **async** and **await** is used.

Listing 4.4: Faust Agent.

```
@app.agent(my_topic)
async def agent(stream):
    async for e in stream.events():
        new_value = process(e.value)
        await send_value_to_topic(other_topic,
            e.key, new_value)
```

Figure 4.4 is a sequence diagram of single-threaded asynchronous code. In this diagram the Main Thread makes two asynchronous calls. The first

[4]https://faust.readthedocs.io/en/latest/

asynchronous operation takes much longer than the second one. It can be seen that while the first operation is executed, the Main Thread can idle for a short time and then runs the second asynchronous operation. The result of the second call is processed from the Main thread even before the first operation completed. In reality this can be the case when the first operation is a long running network operation and the second operation is an operation with a fast file system or even simpler when the data amount of the first operation is much higher than in the second. Without this asynchronous paradigm, in a single-threaded environment the Main Thread would be forced to complete the first operation before the second operation could be started.

Another important concept of Faust are tables for keeping the state. Implementing a strict stream processing architecture, there is only one event or data point processed at a time, but in many cases it is of advantage if some data points can be kept longer or other data can be accessed quickly via a cache. Faust tables are a built-in solution suitable for keeping state. A faust worker is a Python process having one or multiple agents. Of course, in a Python process there can be global variables for keeping state, but such an approach would violate the "Actor Model" paradigm and furthermore the data stored in global variables would be lost after the Python process is terminated. Faust tables are horizontally distributed dictionaries persisted in Kafka topics and cached with RocksDB[5] for faster access. The easy usage of faust table can be seen in Listing 4.5.

Listing 4.5: Faust Table.

```
# Definition of a Table for int values
my_table = app.Table('my_table', default=int)

# access the table like a dict
my_table[key] += 1
```

[5]https://rocksdb.org/

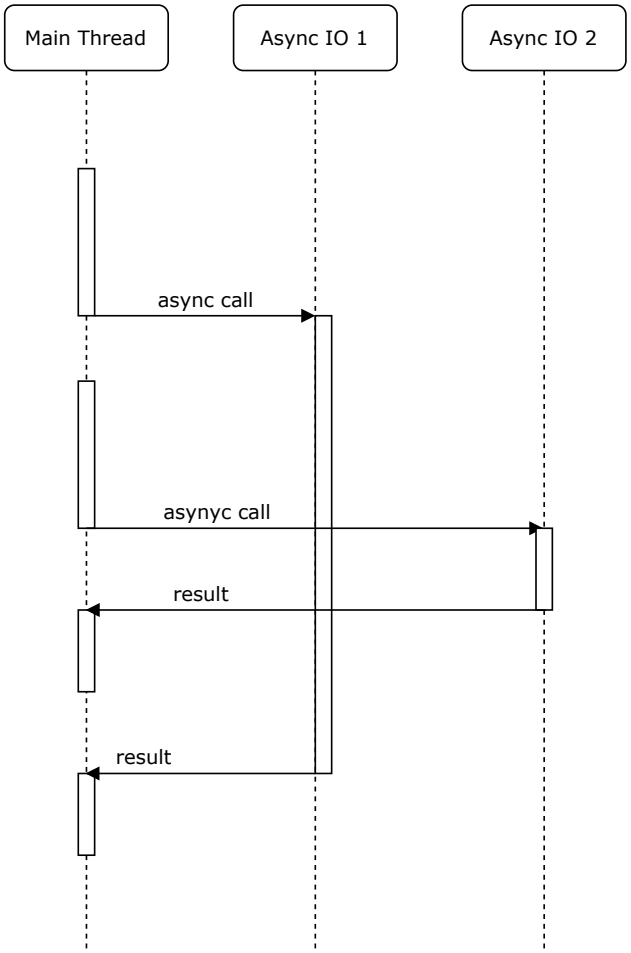

Figure 4.4: Single-Threaded Asynchronous I/O.

4.3 Feature Extraction

This section is about creating the inputs for ML models within the concrete architecture and application domain of smaXtec at prediction phase. In ML, the model inputs are mostly called features, in other literature they are referred as attributes or properties. All the required features (X_1, ... , X_N) for a specific model are formed together to a feature vector of dimension N for N features. Figure 4.5 shows a simplified version of a data pipeline beginning from the raw data until the user got an information.

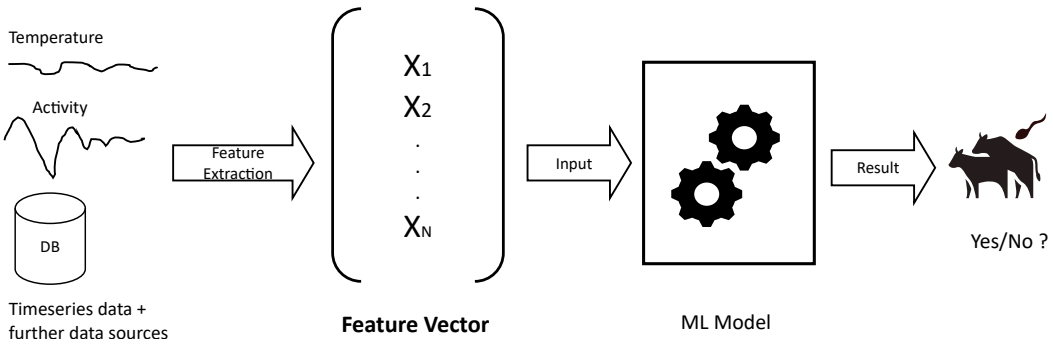

Figure 4.5: Feature Vector.

The features need to be extracted from several data sources. In the case of this project, raw data comes from sensors in intervals of ten minutes, static data about general information of the cows and feedback about certain events from the farmer is stored in a relational database. Figure 4.6 illustrates several data sources needed in this project. Extracting the features from the data sources is in most cases not enough, there might be transformation or encoding steps required. For example the birthday of the cow can be stored as a date in the database, but for further processing the birthday can be transformed to an new feature age in days or years.

The calculation of the feature vector at the prediction phase has to follow the same rules as at the training phase. If there are any deviations the problem

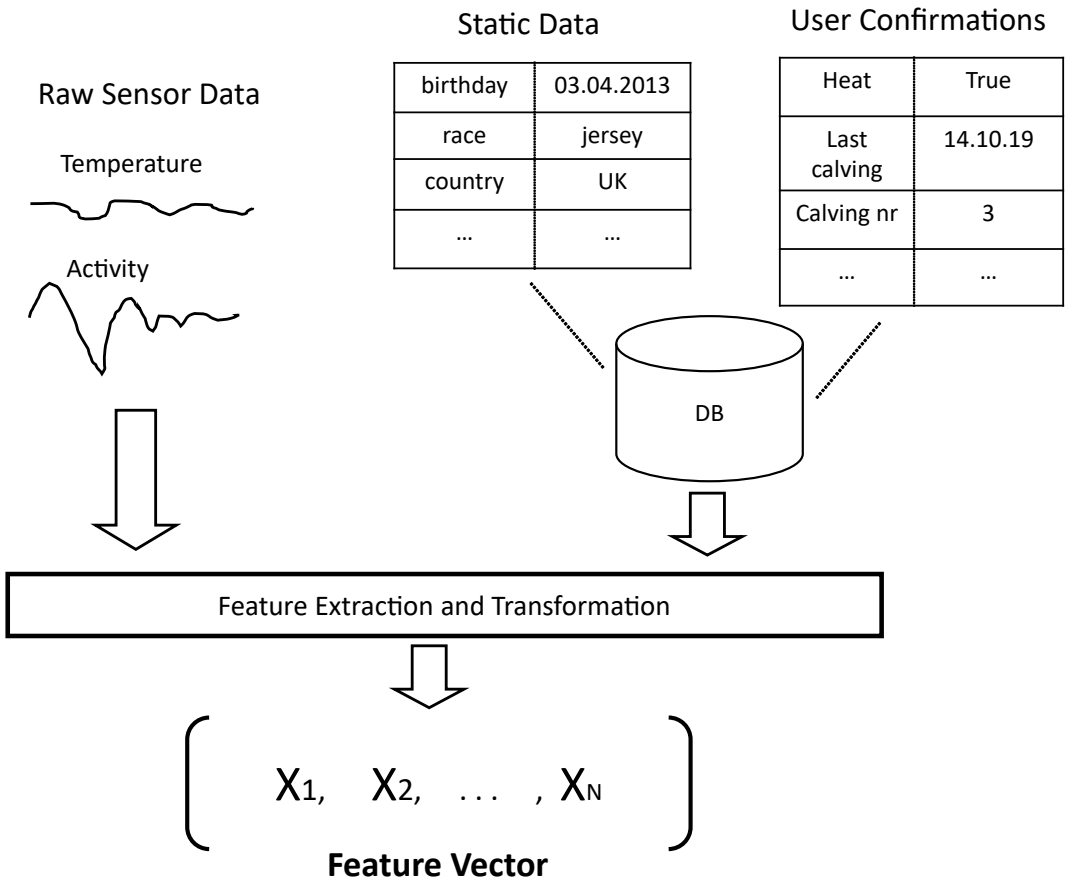

Figure 4.6: Data Sources for Feature Extraction.

of training-serving skew (see Section 2.2.8) occurs, what leads to a not well performing model in production.

From a programming perspective, the feature vector is easier to calculate in the training phase than in the prediction phase, because there are no strict timing requirements and the feature extraction code has not many dependencies to the real software architecture. At the training phase, the data scientist can dump and combine all required data to a suitable store and then constructs and selects all features for model training. At the prediction phase the resulting feature vector has to be the same, but the construction steps need to be embedded into a more complex environment, in the case of this project a stream processing system.

To build reproducible feature extraction and transformation steps, inspiration can be taken from the Pipes and Filters Pattern [45] like illustrated in Figure 4.7.

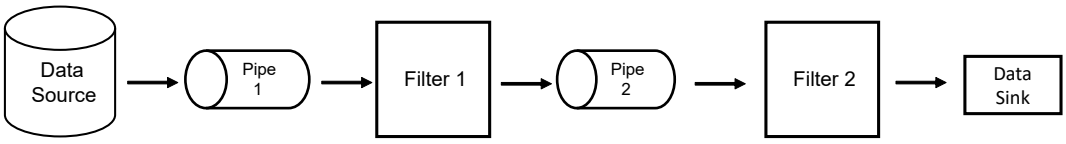

Figure 4.7: Pipes and Filters Pattern.

The Pipes and Filters Pattern describes elements of the data flow beginning from a data source to a data sink. Between the source and sink there can be several transformation steps (Filters) connected via Pipes. This pattern is widely used in the Unix command line, where simple commands can be connected together to a more powerful command. For example the following command (Listing 4.6) concatenates two files (a.txt and b.txt), then filters every line containing the text "hello" (grep) and finally sorts the output in reverse order and saves the result to result.txt. It can be noted that filter steps not necessarily remove data (like grep in the example), rather they can transform and enrich data.

Listing 4.6: Command using Pipes and Filters.

```
$ cat a.txt b.txt | grep -v hello | sort -r > result.txt
```

In the case of the programming model with Kafka/Faust, a feature extraction pipeline can look like in Figure 4.8.

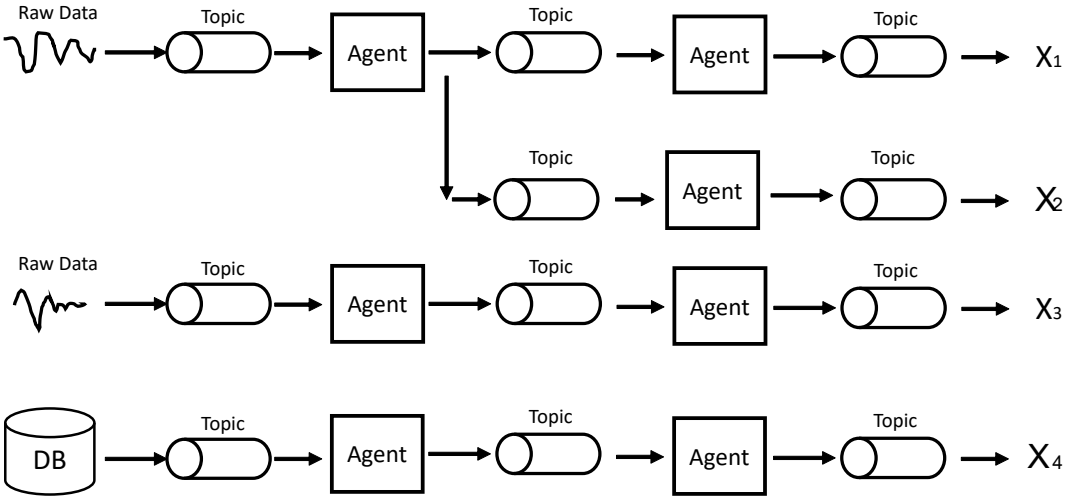

Figure 4.8: Feature Extraction Pipeline.

Using the terminology of the Pipes and Filters Pattern and applying it to the concepts of Faust, agents can act as filters and pipes are connections to topics. Several Faust agents can listen on the same topic and each agent can technically write to multiple topics. To make the data flow simpler to reproduce, every Faust agent should only write to one topic for a data transformation step in the context of feature extraction.

4.3.1 Implementation of the Feature Extraction Pipeline

This subsection describes the main components of the actual implementation of a prototype of the feature extraction pipeline. The main goal is to have a reasonable abstraction of the technical and architectural details like described in the section above. One challenge is to construct the feature vector for a given timestamp, but data comes at different times. This means incoming data needs to be cached for some time and matched together and when all the required data is available a prediction call can be triggered. For this specific application domain the bucketing time for data can be defined with ten minutes. Ten minutes is the minimum time new data can arrive for one sensor. For example one value for feature A arrives at the time 14:01:30 and another value for feature B arrives at 14:02:30, those values can be put

into the same feature vector. Other application domains may have different timing constraints.

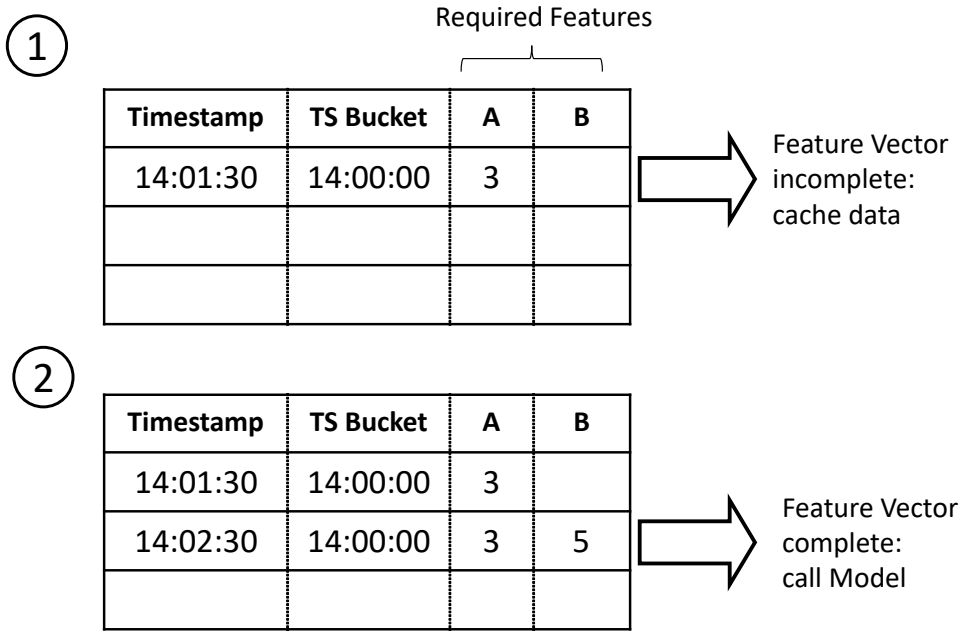

Figure 4.9: Timestamp Bucketing.

Figure 4.9 shows the assumed situation where a model requires only two Features (A and B). First (1), at 14:01:30 the value 3 for feature A comes in, but feature B is missing. The timestamp 14:01:30 is put in the bucket for timestamp 14:00:00. After some time (2), the data for feature B with value 5 is available with timestamp 14:02:30. This timestamp also belongs to the bucket 14:00:00 and now the feature vector is complete and the model can be called to get a prediction result. After that, the cached data can be deleted or should better be stored somewhere else for monitoring purposes (see Section 2.2.9).

Figure 4.10 shows the simplified UML class hierarchy of the stream algorithms. The idea was to reuse and adapt the code of the existing metric calculation classes. The abstract class *StreamAlgo* is responsible for managing the required data and the prototype for the calculation method. Every *StreamAlgo* can define several input metrics which can be understood as

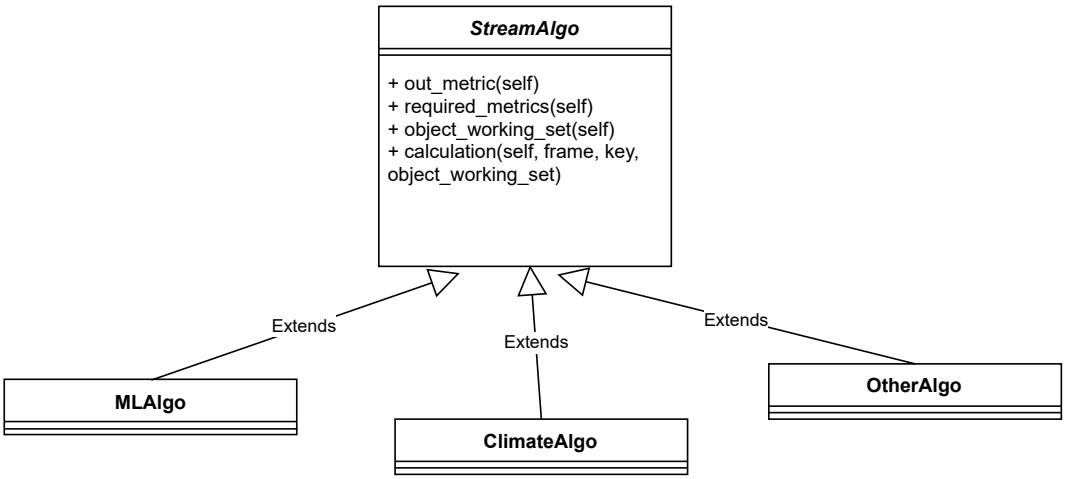

Figure 4.10: Simplified UML Diagram of Stream Algorithms.

the features for a model and as a result one output metric. The ”Object Working Set” is used to access static data like information about the animal or organization. With that abstraction it is simple to implement concrete streaming algorithms but it is also a suitable base for using ML Models. An example of a simple streaming algorithm is the *ClimateAlgo* which derives from *StreamAlgo* and implements the *calculation* method where a simple calculation happens.

The class *MLAlgo* is responsible for calling the prediction function of a specific ML model. How this has to happen depends on how and where the models are deployed (see Section 4.4) and also depends on the communication protocol with the models (see Section 4.5). Additionally the *MLAlgo* class can act as an entry point for logging and monitoring data (see Section 2.2.9).

4.4 Model Deployment

This section explains which technical solutions can be considered to create an appropriate deployment strategy for ML models in the case of smaXtec.

The goal of the deployment process in the software development lifecycle is to transform the code from a development state to a form where the software actually can be used. This includes several steps which should be highly automated to encourage frequent deployments.

Figure 4.11: Deployment steps for traditional software products.

Figure 4.11 shows the typical deployment steps of software. The bare minimum would only be packaging (compiling) the code and shipping the result to the customer or making it accessible via a web server. But incorporating software engineering best-practices, the deployment steps for software products look similar or even more sophisticated than Figure 4.11. The code is usually kept in a version control system where every developer can add changes. After adding these changes the code should pass all unit tests executed from a continuous integration system to make sure that all the existing functionalities, but also the new added features work. It should also be mentioned that a responsible developer runs all unit tests on the local machine before adding the new code to the code repository. The advantage of the tests of the CI system is that even when the developer forgets to check all the tests the system prevents the further deployment steps of faulty code.

After making sure all unit tests have successfully passed the software can be packaged into a shippable form. This step can be different depending on the used programming languages, type of software and destination architecture. For example the packaging step is completely different when packaging a Java web application compared to a firmware for an embedded system.

Before the packaged software is finally shipped to the production system, where the customer can use the software, it is a good practice to have a

separate staging area for testing purposes and getting a first impression of the new features. The production area should be technically very similar like the staging area but needs to be capable to handle the real number of users.

From a high level perspective, deploying ML models has many similarities compared to other software components. Like with traditional software, the code (in this case beginning with the training code of the model) needs to run through a pipeline where the result can finally be used in production. The main difference from a building and deploying point of view is that additional to the (training) code there is also data required. Furthermore, it should not be forgotten that the input pipelines for every model need to be ready before the model can be used in production what may require deployment steps in other parts of the software systems.

The testing is completely different because the rules or logic of the model can not be tested in closed units. Unit tests make only sense for the feature extraction code and to a certain degree for the training code, but for the model itself the build pipeline can only check if the model produces good enough scores and the data distribution meets specified criteria.

The choice of the model packaging format requires some planning about where the model should be stored, served and how the model should be integrated with other systems. For reasons of reproducibility and as a decision helper for further deployments all relevant metadata should be saved.

The staging/production area of software can and should also exist in the case of deploying ML models from an infrastructure point of view. However, instead or additional to a technically separated staging system there can be staged models already deployed to the production area so they can be monitored but the real prediction only comes from the active model. This approach has similarities with A/B Tests [46] mostly known from optimizing the usability of applications.

4.4.1 Model Management

Adding ML models to software systems increases the complexity of the whole software development lifecycle. The following describes an approach how to keep track of experiments, packaging the models and managing deployable models using a Model Registry. Most of these steps are directly supported by MLflow, but can also be replaced with own or other implementations.

A successful deployment starts with the knowledge of what to deploy beginning with the model training. Adjusting the training code with appropriate log-statements, the evaluation metrics can be tracked automatically. To not only have to trust those experimentation metrics from the data scientists, but also to be able to build the model in a reproducible and automatable way using the packaging format of MLflow is a suitable choice. The MLflow packaging format is a convention describing code dependencies, parameters and the command to run the training code. From an operations perspective it should be possible to run this training code automatically in a build pipeline. The automatic run can then be triggered after pushing code to a version control system.

Figure 4.12 shows the main parts of the reproducible model management workflow. In that picture, the file package.yaml describes the code dependencies and parameters of the training code (train.py). The metrics and parameters can be logged with the functions log_metric(name, value) and log_param(param, value). Those logged values are saved with a training run and can be used for further decisions like what model should be deployed next.

Not only parameters and evaluation metrics can be logged. It is also possible to persist the model itself and store it to a Model Registry by calling the save_model() function. The Model Registry keeps track of all saved models and additionally manages the state of each model version. A particular version of a model can per default be in four different states.

First, the model is in "**Experimental**" state what means the model is registered in the Model Registry but there were not further deployment steps.

Like in traditional software deployments the model can first be deployed to a staging environment, what is reflected by the state **Staging**.

The actual model which is used by the serving infrastructure is in state **Production**.

And finally when a model is not in use anymore it can be transitioned to the state **Archive**.

The next section describes how models can be served and integrated with the rest of the software system.

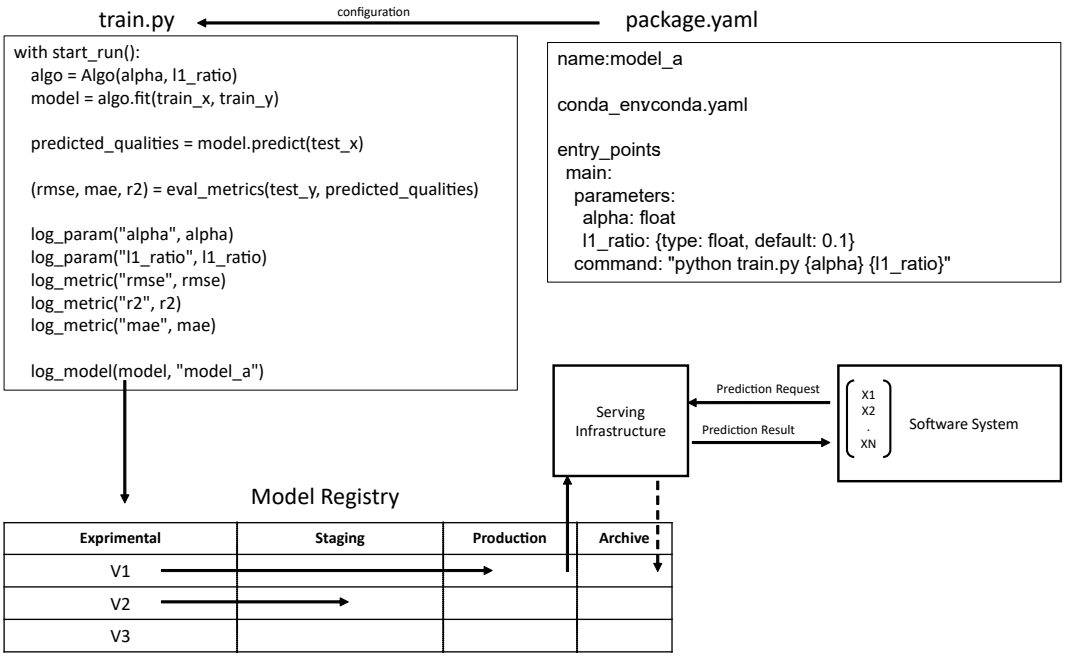

Figure 4.12: Model Management.

4.5 Model Serving and Integration

The previous section described the model deployment pipeline beginning from the training code until the experiments of training runs are saved in a model registry. From an operations point of view, those are important steps for managing the model lifecycle in a professional way to achieve frequent deployments.

From a systems architecture view point and to finally profit from the predictions, the models need to be served and integrated with the rest of the software system. This section is about model serving and integrating the models.

4.5.1 Strategies

There are several known ways [47] how to integrate the trained model with the rest of the software system. The answer to the question of the appropriate strategy to integrate the model with the rest of the software systems (Figure 4.13) depends on the specific business requirements and technical constraints.

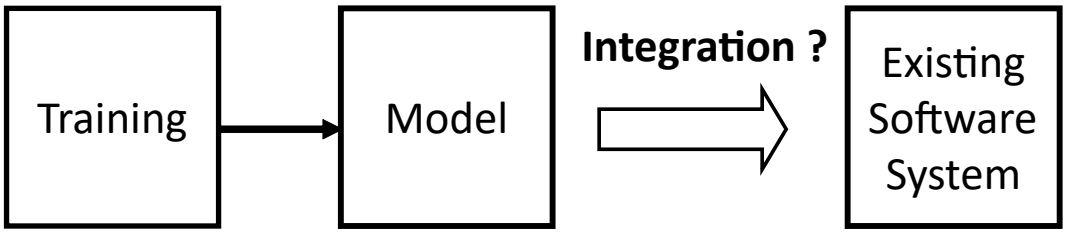

Figure 4.13: How to integrate the model?

The main high-level dimensions (Figure 4.14) to distinguish between integration strategies are in terms of **time** (When should the predictions be done?), **place** (Where should the model be served and integrated?) and **protocol** (What is the protocol to communicate with the model?). The following will discuss different integration approaches considering the three factors time, place and protocol.

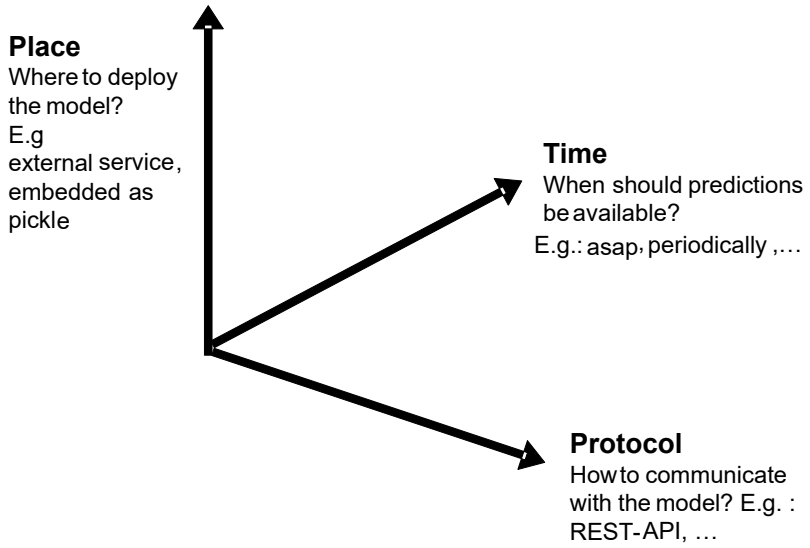

Figure 4.14: Dimensions of integration strategies.

The answer to the question, when the prediction should be latest done depends on the business case. The deadline for getting prediction results is the time where the consumer of the model can still profit from the result. In the context of this project some alerts are time critical so it makes sense to directly call the predict function of the model as soon as all input data is available. Section 4.3.1 already described the creation of the feature vector. As soon as the feature vector is complete prediction results can be triggered.

An alternative would be to persist complete feature vectors and periodically call the predict function to batch process this data. Since the system architecture is already build on the stream processing pattern it makes sense to trigger prediction requests as soon as the feature vector is complete. Although, it can be mentioned that the batch approach has advantages in system operations. For example the batch approach only needs to serve the model when predictions should be made and down-times of the model server are not immediately a big issue.

The place where the model should be integrated is also an important decision. Basically there are three options. First, the model can be directly

integrated into a monolithic software system (see Section 2.2.4) similar like any other library. Going this approach it is challenging to deploy often without down-times because the whole monolith is affected. To be more independent from the rest of the system it makes sense to deploy the model as a separate service. Only for completeness it can be said that there is a (theoretical) third option, which is deploying the model directly on the device of the end-user using tools like TensorFlow.js[6], but this is only a viable option if all the required data is available on the client-side and all other challenges like updates and monitoring can be guaranteed.

The way to communicate with the model depends on how and where the model is deployed. For example in the simple case when the model is deployed as local pickle file, only the parameters (feature vector) for the predict function needs to be provided. On the other hand, when the model is deployed as a separate service a communication protocol and data serialization is required to communicate with the service.

A common solution is to deploy the model as a service exposing a REST-API [48]. Usually REST-APIs are used to create, read, update and delete (CRUD) data in the context of web applications. For example the frontend application sends a GET-Request to retrieve data from the backend system. This abstraction encourages the decoupling of different parts of a software system. Many commercial solutions provide the serving of ML models as a managed REST service. Another possibility to deploy the model as a separate service, but more tightly coupled to the Kafka stream processing system (4.2.2) would be the deployment as a Faust agent (4.2.3). Both, the deployment as a REST-Service as well as a Faust agent can technically coexist at the same time. The following tow subsections will describe both methods in more detail.

4.5.2 Serving the model as a REST-Service

Serving the model as a REST-service separates the model clearly from the rest of the system. REST-Services are mostly built on top of HTTP (Hypertext

[6]https://www.tensorflow.org/js

Transfer Protocol), although the binding to HTTP is not mandatory. REST-Services implement their actions using HTTP-Methods (GET, POST, PUT, DELETE) [49]. A simple GET-Request can look like *GET /user/1* to read a user with id *1* from the backend system. To prepare the response for simple further processing, the data is often serialized in JSON (JavaScript Object Notation) or XML (Extensible Markup Language). Figure 4.15 illustrates a HTTP GET-Request, retrieving a JSON-Response.

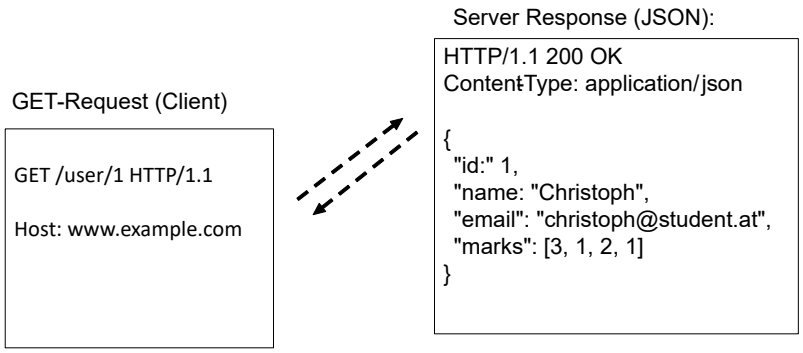

Figure 4.15: Example of a HTTP GET-Request and JSON-Response.

With lightweight web servers and frameworks like Flask[7] it is easy to load a Python model as a serialized pickle, processing the prediction requests and sending the result back to the client as a JSON-Response.

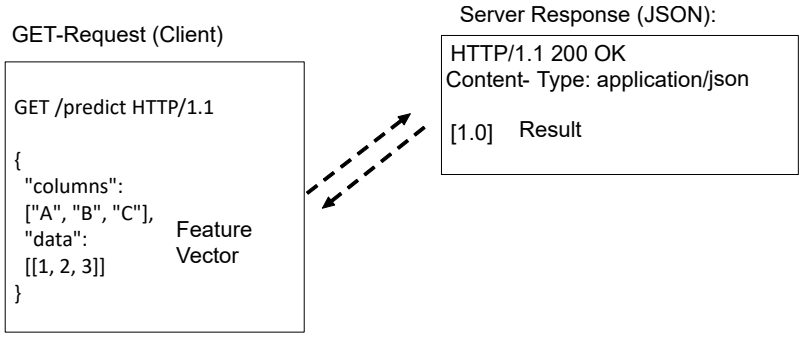

Figure 4.16: Prediction Request/Response via REST-API, JSON serialized.

Figure 4.16 shows a HTTP GET-Request sending a feature vector with three

[7]https://flask.palletsprojects.com/

columns (A, B and C) and the corresponding JSON-Response as a simple list with only one element with the value 1.0.

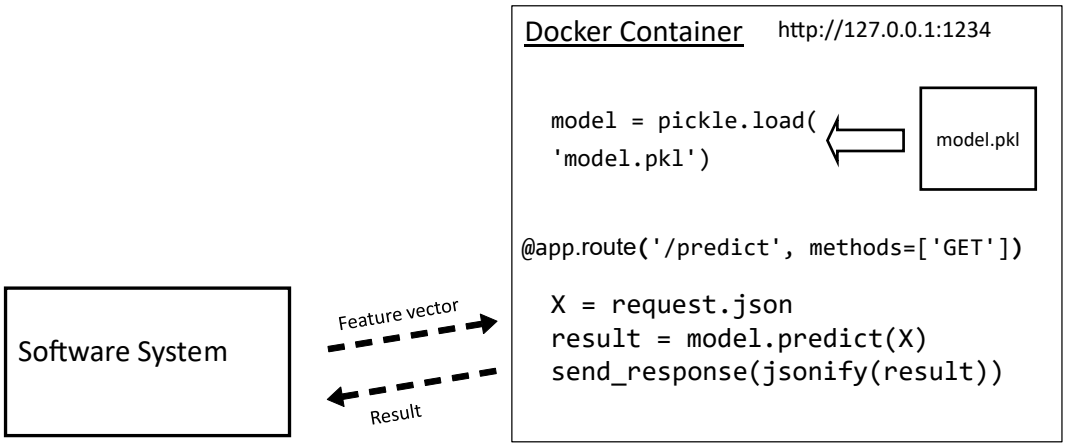

Figure 4.17: Model deployed as a docker container exposing a REST-API.

Loading the model in a Flask web application and serving this application like illustrated in Figure 4.17 as a docker container is a simple and clean approach. Serializing Python code as a pickle file can lead to version problems with different Python versions. Using docker, version problems are not a big deal. For every new model version we can delete old containers and start the a new container with the right versions and dependencies as required. A further advantage of this approach is that every docker container is a stateless instance of the REST-API. This means that there can be as many instances launched as required to handle all requests in time.

The models component of MLflow[8] supports model serving as a REST-API and the creation of docker containers.

[8]https://www.mlflow.org/docs/latest/models.html

4.5.3 Serving the model as a Faust Agent

Technically, there are several protocols possible to communicate with the model. REST-APIs are quite common, but in event/stream processing systems like Kafka/Faust it can also make sense to see the model as a separate stream processor. Deploying the model as a Faust agent has the advantage, that there is no need to communicate with services outside the stream processing system.

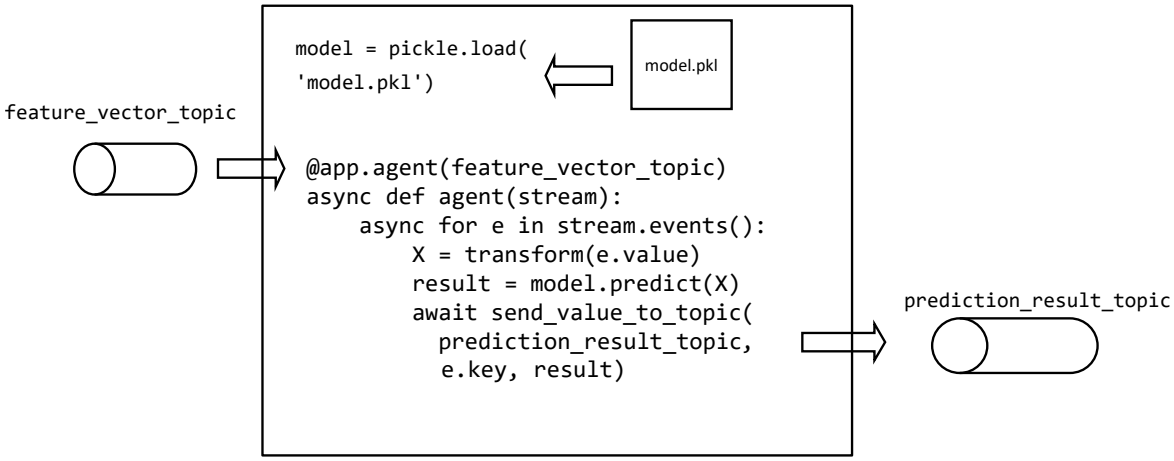

Figure 4.18: Model deployed as a Faust Agent.

Figure 4.18 is an example of a Python model deployed as a Faust Agent. The agent listens for new incoming data on the topic (”feature_vector_topic”). The model itself needs to be loaded as a pickle file. For performance reasons, the loading can be done once outside the event loop of the agent. The data in the topic (”feature_vector_topic”) can be encoded not different than the data of other topics, but needs to be transformed as required for the model’s predict method. In this project, the data serialization format for the topics is Avro[9].

[9]https://avro.apache.org/

4.6 Monitoring

Technically it would be sufficient to just deploy the models and get prediction results. However, for a real system in production it is important to know if the system works like expected. This is especially true for ML models, because they can not be tested like traditional software and their outcome depends on the data distribution what can be different compared to the training phase. Section 2.2.9 already described monitoring as a challenge in the ML system lifecycle.

Monitoring technical parameters (response time, requests per second, ...) is not different than monitoring other services. For example, when deploying the model as a REST Service (see Section 4.5.2) the same monitoring solutions can be used like for any other service.

Non-technical values are difficult to monitor because by now there is no standard solution and approach available. Also this project can not provide a generic solution. Although, this section describes how a monitoring component can be implemented within this system architecture.

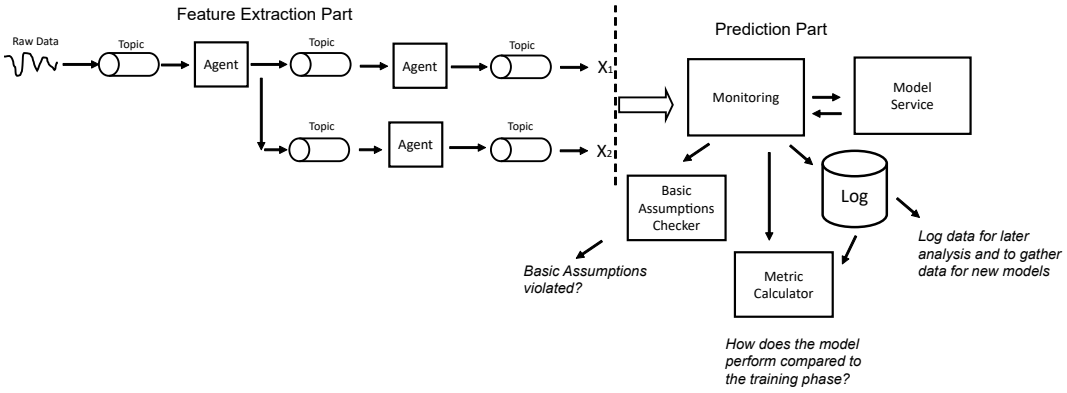

Figure 4.19: Monitoring Component.

Figure 4.19 illustrates the monitoring component in the prediction part of the data pipeline. The technical task to collect the data for monitoring is rather simple because all the required data is available before the prediction request. For every complete input feature vector (see Section 4.3.1) this data

needs to be logged together with the prediction result. In Listing 4.7 this logging is implemented as a write to the "input_and_result_topic" topic.

Listing 4.7: Agent with Monitoring.

```
@app.agent(feature_vector_topic)
async def agent(stream):
    async for e in stream.events():
        result = await prediction_request(e.value)

        # send data for monitoring purposes
        data = InputAndResultModel(e.value, result)
        await send_value_to_topic(
        input_and_result_topic,
            e.key, data)
        # ...
```

In the best case the monitoring component would allow us to directly compare the model in production with the metrics calculated in the training phase. The problem is, that this is not possible or only possible when it is already too late. It is obvious that immediately after the model classified data there is no automatable way to verify the result, because if such a perfect classifier would exist, it would make sense to use this classifier instead of the model.

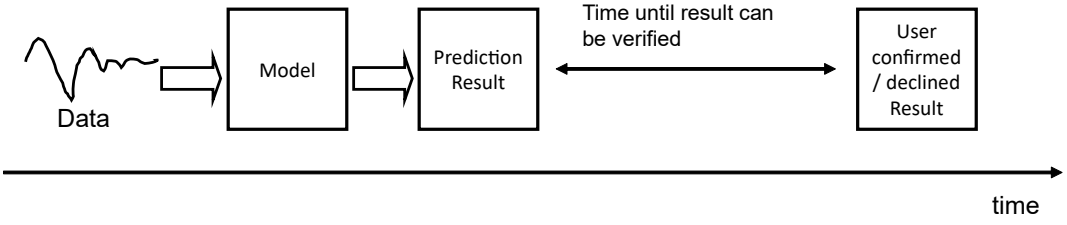

Figure 4.20: Time until the prediction result can be verified.

One way to verify the model's result is to ask the user for feedback for given events. For some events, a human can check if a prediction result was wrong or right very soon . But for some events it can take many days (Figure 4.20) until qualified feedback can be given. Another drawback of waiting for user feedback is, that not all users are willing to give feedback and get annoyed or report wrong results.

Since trusting on user feedback alone is not very practical for all use-cases other or additional performance indicators are required for model monitoring. The question to answer is, does performance of the model in production deviate from the performance in the training phase. As said before, calculating the same metrics is often too late, but it is possible to monitor the incoming data and make statistics about the data distribution of the incoming data and the prediction results. The phenomenon that a model's performance degrades over time is normal and known under concept drift [50]. This is not necessarily a bad thing. On the one hand this can be seen as a problem because the model is probably not good enough anymore. On the other hand a better model can be trained with the new data.

Similar to monitoring the data distribution for model inputs and outputs is to check for basic assumptions. For example, when the problem domain makes it clear that specific events can not happen in very short time frames, the system can generate alerts.

In [51] the monitoring component is described with data processing & storage, visualization and alerting.

1. **Processing & Storage**: This is the part where the data is logged for further analysis. Listing 4.7 showed an agent which intercepted the data before and after a prediction request and logged that data to a kafka topic.

2. **Visualization**: A visualization component displays the logged data in charts and dashboards. There are already a lot of existing solutions and in many cases they can be used without coding an own solution.

3. **Alerting**: Monitoring can not only include watching and storing data, but also automatically generating alerts. Listening to the topics used for storing monitoring data it is easy to generate alerts when specific events occur.

5 Results

This chapter describes the most reasonable solution of a ML pipeline for the smaXtec system. Possible approaches were already described in the previous Chapter 4, but this chapter describes the suggested solution in a compact form including the concrete steps to deploy (see Section 5.3.1) and update (see Section 5.3.2) models and discusses the fulfillment of the requirements (see Section 5.2).

5.1 Overview of the solution

Figure 5.1 illustrates the suggested steps to implement the high-level ML pipeline based on Figure 1.2. It can be seen that shipping a model from the training phase to the running phase does not only include packaging and deploying the model but also reusing data transformation code. Many parts of the pipeline benefit from the functionalities of MLflow. Using the components MLflow tracking and projects the entire training phase is completely reproducible. The final model is stored in the model registry together with the metadata (creation time, performance metrics, version, ...) of the model.

Models from the model registry are deployed as docker containers exposing an API endpoint. In the running phase the feature vectors can be constructed with Faust agents by reusing code from the training phase and then HTTP requests to the API endpoints of the models can be made. Following this approach the serving infrastructure is pretty simple as it is technically a normal stateless web service. Technical parameters like requests per second can be monitored like any other web service. Specific metrics (precision,

recall, ...) are more complicated to monitor and require use case specific implementations.

This solution was chosen because:

- All requirements can be fulfilled (see Section 5.2).
- No expensive third-party solutions are necessary.
- It fits into the existing system architecture.
- Out-of-the-box support for most ML libraries.
- Low entry barriers (costs and complexity) with MLflow.
- Parts of the pipeline can be easily replaced with own or other implementations.
- Simple reuse of data transformation code.
- Most parts of the pipeline are also useful for non-ML algorithms.

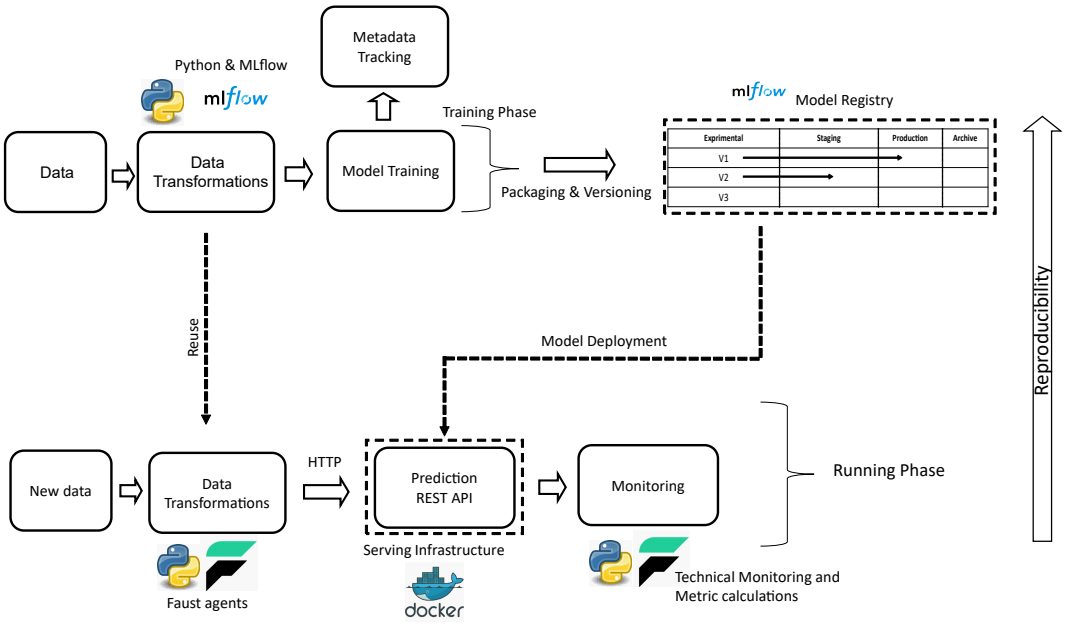

Figure 5.1: Overview of the suggested solution.

5.2 Fulfillment of the requirements

This section describes how the requirements listed in Section 4.1 at page 42 were fulfilled.

- Functional requirements
  - **Feature extraction**: Section 4.3.1 described the implementation of the feature extraction pipeline. The task was to evaluate the usefulness of the existing data pipeline implemented with Faust agents. By reusing data transformation code of the training phase and joining the output of multiple agents with Faust tables, the feature vector can be constructed. Assuming that data transformations in the training phase are implemented with Python, there are no changes in the existing system architecture necessary for this part.

  - **Model deployment**: As described in Section 3.1 there are many existing solutions for model persistence as a first step for model deployment and serving. The MLflow models component seems to be a suitable option for this project. At the beginning of this project, it was not totally clear what frameworks for model training will be used. With MLflow it is possible to consistently persist and load models of various frameworks and create docker images or deploy them to third-party cloud services.

  - **Model serving and integration**: The way the model is served determines the way it needs to be integrated with the software system. Section 4.5 described two possibilities for model serving and integration. The first option directly integrates the model into the Faust data data pipeline (see Section 4.5.3), the second option serves the model as a REST web service (see Section 4.5.2). Basically, both options are fine and can be used together, however, serving the models as a REST web services has some advantages and is suggested for this project. The isolated web service can be deployed completely independent and many instances of the same model can be run in parallel because of the stateless architecture of REST web services. Furthermore, it can make sense to

move the models to third-party providers that offer specialised prediction hardware and also offer the prediction interface via REST per default.

  - **Model monitoring**: When serving the models as web services, the technical parameters like latency and requests per second can easily be monitored like it is done with any other web service. Performance metrics of ML models are not that easy to monitor at runtime. We can not take an existing tool as a generic solution that does this job. For example, in classification the true positives, true negatives, false positives, false negatives are required to calculate the same metrics (precision, recall, f1-score, ...) as in the training phase. Counting these values is use case specific and often takes several days in the domain of animals. When a user confirms or declines a generated event, we can calculate the metrics, assuming the user answered correctly. From a system architecture perspective we can extend the data pipeline with further Faust agents with use case specific monitoring code. Additionally, as a simple generic but less accurate monitoring solution we can just count the prediction results and check for abnormal distributions.

  - **Model versioning**: Using the MLflow tracking component we get a history of every training run and the created model. With registering the model in the model registry (see Section 4.4.1) we can keep track of all models and their deployment state. By making requests to all models for a certain use case we can have multiple models in parallel and take the result of the model in stage "production" for event generation. If we want to switch the active (production) model we can simply change the states in the model registry to take a new model or rollback to an older version.

- Non-functional requirements
  - **"Real-time" predictions**: As soon as all input features are joined together the model web service can be requested. There are no waiting times compared to a batch processing approach.

- **Scalability**: It can be argued that the proposed adaptations of the system architecture do not have a bad impact on scalability. The data pipeline for calculating input data is similar as used for traditional algorithms. Using separate web services for algorithms (the deployed models) is a little bit slower (latency) than directly executing the code. However, due to the stateless nature of the deployed model web services, there can be multiple instances of the same model deployed for horizontal scaling and therefore running the models is not a bottleneck.

- **Reproducibility**: Having trained models stored in the model registry, reproducibility is easy to achieve. A typical use case is to find out why and how a certain event was created by the system. When the system creates an event, we can attach the model-id and the current version of the data pipeline (git-commit-id) in the corresponding database entry. Having the model-id we can look up the model in the model registry and see how the model was created. The git-commit-id gives us the version of the data pipeline what can be used for debugging or checking if the version of the deployed model does fit to the deployed data pipeline.

- **Frequent deployments**: The proposed solution supports frequent deployments as the model services are loosely coupled with the rest of the system. Updates of a model do not influence the rest of the system resulting in zero down-times. Building deployable model services can be fully automated.

5.3 Deployment Steps

The following sections describe how to deploy and integrate completely new developed models as well as the steps to update an existing model. Furthermore, it is explained how the deployment process of traditional algorithms profits from the things learned in this thesis.

5.3.1 Steps for deploying and integrating a new model

The steps for deploying and integrating a model are:

1. Make sure the created model is stored in the model registry.
2. Set the deployment stage (staging, production, archive) in the model registry.
3. Serve the model as a REST API endpoint.
4. Implement model specific monitoring code (recommended).
5. Extend the existing data pipeline if there are new input features required.
6. Build the feature vector as described in Section 4.3.1.
7. Add code to request the model given the URL endpoint of the model service every time the feature vector is complete.

5.3.2 Updating an existing model

Updating an existing model is rather simple. If the model is accessible via the same URL as the previous version, there is no need to update other parts of the software system.

5.3.3 Deployment of traditional algorithms

Despite this thesis examines the impact of ML models on smaXtec's system architecture and deployment processes there will still be traditional algorithms in use. Using the things learned in this thesis the development and deployment processes of non-ML algorithms can be improved. For example, the MLflow tracking component is also useful (automatic performance tracking, versioning) for developing non-ML algorithms. Listing 3.7 shows how to implement non-ML algorithms using the MLflow models component. This way the non-ML algorithms can be deployed following the same steps as ML models.

6 Conclusion

Integrating ML models into software systems is still quite a challenge. Because deployed models need to be updated frequently, appropriate tools and system architecture is required. However, compared to traditional software development, what is also a difficult task, the methodologies, processes and tools for managing the ML lifecycle are less mature, studied and standardized.

This thesis described the challenges in different stages of the ML lifecycle and derived requirements for one real-world software project. To get a better overview and to avoid the development of unstable code, third-party technologies for different subproblems were analyzed. There are already several proven ways to persist ML models and associated metadata.

MLflow seems as a promising solution for most phases of the ML lifecycle and got even more useful functionalities during the creation of this thesis. It supports reproducible model training code, tracking of metadata, persisting the model created with various ML libraries and directly serving the model or deploying it to cloud platforms or as a docker container. Additionally, there is a model registry to keep track of the models in different stages encouraging separated deployments in staging and production environments.

Additionally to evaluating third-party solutions, the suitability of the existing software architecture with its data pipelines and data sources were analyzed and several input sources combined in an example project.

There are multiple options on how to serve ML models. As a simple and scalable option, the deployment as a stateless web service was identified. Serving and deploying web services is well-studied and ensures loose-coupling with the rest of the system, but also the statelessness supports

running arbitrary instances of the service depending on the load requirements.

Deployment of ML models is not only the technical task of persisting a trained model and pushing it to the serving infrastructure, to a certain extend every phase of the ML lifecycle needs to be considered. To reproduce the output of a deployed model, a monitoring component needs to keep track of the produced outputs, but also the model itself and the data including data transformations need to be correctly versioned. It is also highly recommend to store the training code and the dependencies in a way that the model created by a data scientist can always be automatically recreated. Overall, this thesis showed possibilities to deploy and serve ML models and integrating them with a software system. However, there are useful existing technologies, it is very challenging to apply best-practices compared to traditional software projects.

Bibliography

[1] A. Ng. (2017). Why ai is the new electricity, [Online]. Available: https://news.stanford.edu/thedish/2017/03/14/andrew-ng-why-ai-is-the-new-electricity/ (cit. on p. 1).

[2] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'15, Montreal, Canada: MIT Press, 2015, pp. 2503–2511. [Online]. Available: http://dl.acm.org/citation.cfm?id=2969442.2969519 (cit. on pp. 1, 9, 12).

[3] P. Dewayne and A. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, Sep. 2000. DOI: 10.1145/141874.141884 (cit. on p. 3).

[4] A. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, no. 3, pp. 210–229, Jul. 1959. DOI: 10.1147/rd.33.0210 (cit. on p. 3).

[5] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds., *Structured Programming*. London, UK, UK: Academic Press Ltd., 1972, ISBN: 0-12-200550-3 (cit. on p. 3).

[6] M. Fowler. (2006). Continuous integration, [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html (cit. on p. 3).

[7] H. v. Vliet, *Software Engineering: Principles and Practice*, 3rd. Wiley Publishing, 2008, ISBN: 0470031468 (cit. on p. 3).

[8] J. Humble, C. Read, and D. North, "The deployment production line," in *AGILE 2006 (AGILE'06)*, Jul. 2006, 6 pp.-118. DOI: 10.1109/AGILE.2006.53 (cit. on p. 3).

[9] C. Renggli, B. Karlas, B. Ding, F. Liu, K. Schawinski, W. Wu, and C. Zhang, "Continuous integration of machine learning models with ease.ml/ci: Towards a rigorous yet practical treatment," *CoRR*, 2019. arXiv: 1903.00278. [Online]. Available: http://arxiv.org/abs/1903.00278 (cit. on p. 3).

[10] N. Pentreath. (2018). Deploying machine learning models in practice, [Online]. Available: https://qconsp.com/sp2018/system/files/presentation-slides/qconsp18-deployingml-may18-npentreath.pdf (cit. on pp. 4, 5).

[11] A. Géron, *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2017, ISBN: 978-1491962299 (cit. on p. 5).

[12] R. Pressman, *Software Engineering: A Practitioner's Approach*, 7th ed. New York, NY, USA: McGraw-Hill, Inc., 2010, ISBN: 0073375977, 9780073375977 (cit. on p. 5).

[13] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*, 1st. O'Reilly Media, Inc., 2016, ISBN: 1491956259, 9781491956250 (cit. on p. 5).

[14] I. Niaz, "Automatic code generation from uml class and statechart diagrams," Jan. 2005 (cit. on p. 8).

[15] C. Thornton, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Auto-weka: Automated selection and hyper-parameter optimization of classification algorithms," *CoRR*, vol. abs/1208.3719, 2012. arXiv: 1208.3719. [Online]. Available: http://arxiv.org/abs/1208.3719 (cit. on p. 9).

[16] L. E. Li, E. Chen, J. Hermann, P. Zhang, and L. Wang, "Scaling machine learning as a service," in *Proceedings of The 3rd International Conference on Predictive Applications and APIs*, C. Hardgrove, L. Dorard, K. Thompson, and F. Douetteau, Eds., ser. Proceedings of Machine Learning Research, vol. 67, Microsoft NERD, Boston, USA: PMLR, Nov. 2017, pp. 14–29. [Online]. Available: http://proceedings.mlr.press/v67/li17a.html (cit. on pp. 9, 34).

[17] J. Dunn. (2016). Introducing fblearner flow: Facebook's ai backbone, [Online]. Available: https://engineering.fb.com/ml-applications/introducing-fblearner-flow-facebook-s-ai-backbone/ (cit. on p. 9).

[18] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, C. Y. Koo, L. Lew, C. Mewald, A. N. Modi, N. Polyzotis, S. Ramesh, S. Roy, S. E. Whang, M. Wicke, J. Wilkiewicz, X. Zhang, and M. Zinkevich, "Tfx: A tensorflow-based production-scale machine learning platform," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17, Halifax, NS, Canada: ACM, 2017, pp. 1387–1395, ISBN: 978-1-4503-4887-4. DOI: 10.1145/3097983.3098021. [Online]. Available: http://doi.acm.org/10.1145/3097983.3098021 (cit. on p. 9).

[19] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *International Conference on Software Engineering (ICSE 2019) - Software Engineering in Practice track*, IEEE Computer Society, May 2019. [Online]. Available: https://www.microsoft.com/en-us/research/publication/software-engineering-for-machine-learning-a-case-study/ (cit. on p. 9).

[20] S. Schelter, F. Bießmann, T. Januschowski, D. Salinas, S. Seufert, and G. Szarvas, "On challenges in machine learning model management," *IEEE Data Eng. Bull.*, vol. 41, no. 4, pp. 5–15, 2018. [Online]. Available: http://sites.computer.org/debull/A18dec/p5.pdf (cit. on pp. 9, 11, 18).

[21] M. Kim, T. Zimmermann, R. DeLine, and A. Begel, "Data scientists in software teams: State of the art and challenges," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1024–1038, Nov. 2018. DOI: 10.1109/TSE.2017.2754374 (cit. on p. 10).

[22] C. W. Danilo Sato Arif Wider. (2019). Continuous delivery for machine learning, [Online]. Available: https://martinfowler.com/articles/cd4ml.html (cit. on pp. 10, 11).

[23] S. Schelter, J.-H. Böse, J. Kirschnick, T. Klein, and S. Seufert, "Automatically tracking metadata and provenance of machine learning experiments," 2017 (cit. on p. 12).

[24] I. Sommerville, *Software Engineering*, 9th ed. Addison-Wesley, 2010, ISBN: 978-0-13-703515-1 (cit. on p. 13).

[25] B. Goodman and S. Flaxman, "European union regulations on algorithmic decision-making and a "right to explanation"," *AI Magazine*, vol. 38, no. 3, pp. 50–57, Oct. 2017. DOI: 10.1609/aimag.v38i3.2741. [Online]. Available: https://www.aaai.org/ojs/index.php/aimagazine/article/view/2741 (cit. on p. 19).

[26] H. Dam, T. Tran, and A. Ghose, "Explainable software analytics," May 2018. DOI: 10.1145/3183399.3183424 (cit. on pp. 19, 20).

[27] Google. (2018). Rules of machine learning - best practices for ml engineering, [Online]. Available: https://developers.google.com/machine-learning/guides/rules-of-ml#training-serving_skew (cit. on pp. 20, 21).

[28] D. S. Kester Tong and G. Katsiapis. (2017). Preprocessing for machine learning with tf.transform, [Online]. Available: https://ai.googleblog.com/2017/02/preprocessing-for-machine-learning-with.html (cit. on p. 21).

[29] M. Loukides. (2012). What is devops? [Online]. Available: http://radar.oreilly.com/2012/06/what-is-devops.html (cit. on p. 23).

[30] A. Trends. (2018). Why mlops (and not just ml) is your business' new competitive frontier, [Online]. Available: https://www.aitrends.com/machine-learning/mlops-not-just-ml-business-new-competitive-frontier/ (cit. on p. 23).

[31] Microsoft. (2020). Machine learning operations maturity model, [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/example-scenario/mlops/mlops-maturity-model (cit. on p. 23).

[32] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "What's your ml test score? a rubric for ml production systems," 2016 (cit. on p. 23).

[33] M. Zeller, R. Grossman, C. Lingenfelder, M. Berthold, E. Marcadé, R. Pechter, M. Hoskins, W. Thompson, and R. Holada, "Open standards and cloud computing : Kdd-2009 panel report," *Publ. in: Proceedings of the 15th ACMKDD International Conference on Knowledge Discovery and Data Mining : June 28 - July 1, 2009, Paris, France / sponsored by ACM SIGMOD and ACM SIGKDD. New York, NY, 2009, pp. 11-18*, Jan. 2009. DOI: 10.1145/1557019.1557027 (cit. on p. 27).

[34] N. H. Airbnb Engineering and A. Keys. (2014). Architecting a machine learning system for risk, [Online]. Available: https://medium.com/airbnb-engineering/architecting-a-machine-learning-system-for-risk-941abbba5a60 (cit. on p. 27).

[35] E. Boyd. (2017). Microsoft and facebook create open ecosystem for ai model interoperability, [Online]. Available: https://azure.microsoft.com/en-us/blog/microsoft-and-facebook-create-open-ecosystem-for-ai-model-interoperability/ (cit. on pp. 28, 33).

[36] L. Patruno. (2017). Docker for machine learning, [Online]. Available: https://mlinproduction.com/docker-for-ml-part-1/ (cit. on p. 29).

[37] C. Boettiger, "An introduction to docker for reproducible research.," *Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015 (cit. on p. 29).

[38] A. Jaleel. (2020). Startup - rest api with flask, [Online]. Available: https://www.kaggle.com/ahammedjaleel/startup-rest-api-with-flask (cit. on p. 32).

[39] I. Hellström. (2020). A tour of end-to-end machine learning platforms, [Online]. Available: https://www.kdnuggets.com/2020/07/tour-end-to-end-machine-learning-platforms.html (cit. on p. 34).

[40] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, F. Xie, and C. Zumar, "Accelerating the machine learning lifecycle with mlflow," *IEEE Data Eng. Bull.*, vol. 41, pp. 39–45, 2018 (cit. on p. 34).

[41] Apache Software Foundation. (2020). Kafkaproducer 2.0.0 api, [Online]. Available: https://kafka.apache.org/20/javadoc/org/apache/kafka/clients/producer/KafkaProducer.html (cit. on p. 47).

[42] Apache Software Foundation. (2020). Kafkaconsumer 2.0.0 api, [Online]. Available: https://kafka.apache.org/20/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html (cit. on p. 47).

[43] Confluent. (2020). Ksql and kafka streams, [Online]. Available: https://docs.confluent.io/current/ksql/docs/concepts/ksql-and-kafka-streams.html (cit. on p. 48).

[44] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," 1973 (cit. on p. 49).

[45] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Chichester, UK: Wiley, 2000, ISBN: 978-0-471-60695-6. [Online]. Available: https://www.safaribooksonline.com/library/view/pattern-oriented-software-architecture/9781118725177/ (cit. on p. 54).

[46] R. Kohavi and R. Longbotham, "Online controlled experiments and a/b testing," in. Jan. 2017, pp. 922–929. DOI: 10.1007/978-1-4899-7687-1_891 (cit. on p. 59).

[47] J. Kervizic. (2019). Overview of different approaches to deploying machine learning models in production, [Online]. Available: https://www.kdnuggets.com/2019/06/approaches-deploying-machine-learning-production.html (cit. on p. 62).

[48] R. T. Fielding, "Architectural styles and the design of network-based software architectures," AAI9980887, PhD thesis, 2000, ISBN: 0-599-87118-0 (cit. on p. 64).

[49] R. T. Fielding. (2016). Hypertext transfer protocol – http/1.1 - methods definitions, [Online]. Available: https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html (cit. on p. 65).

[50] A. Tsymbal, "The problem of concept drift: Definitions and related work," May 2004 (cit. on p. 70).

[51] C. Samiullah. (2020). Monitoring machine learning models in production - a comprehensive guide, [Online]. Available: https://christophergs.com/machine%20learning/2020/03/14/how-to-monitor-machine-learning-models/ (cit. on p. 70).