Institute of Software Engineering

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Reference Architectures for MLOps: A Comparative Case Study

Samuel Reich

Course of Study:	Informatik
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Markus Haug
Commenced:	May 13, 2024
Completed:	November 13, 2024

Abstract

Machine Learning (ML) has become a huge topic in the field of Software Engineering in recent times. An important step in the development of a software system is the selection of a fitting reference architecture to base the system on. This also applies for software systems in the context of Machine Learning. The goal of this study was to collect reference architectures for ML-based systems and compare them based on a selected use case from the field of regression-based models. We performed a literature review to find reference architectures in scientific literature. We found some literature on the topic, from which we extracted six reference architectures. Some of the architectures are already well known and also used outside of ML topics. From the six architectures we found, we selected two based on their applicability for our use case and implemented a prototype system for each of them, both with the same functionality. Using all the information we gathered and our prototype systems, we then evaluated the benefits and drawbacks of the different architectures. We envision that our results can help practitioners in the process of choosing a reference architecture for their system.

Contents

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence. 15

ML Machine Learning. 3, 13, 15, 17, 19, 24, 25, 26, 30, 40, 41, 42, 43

MSA Microservice Architecture. 9, 37, 38, 39, 40, 41, 42, 43

SE Software Engineering. 13, 15, 43

TLA Three-layer Architecture. 9, 37, 38, 39, 40, 41, 42, 43

1 Introduction

Machine Learning (ML) is a rapidly developing topic and has quickly grown in importance in recent times. But Software Engineering (SE) for ML is still in an early state. Development of a ML system begs the question how to approach this process. One major part of designing a software system is choosing a reference architecture to base the system on. For regular SE there are plenty of reference architectures for various use cases. But what about SE for ML systems? Which reference architectures are there? How do they compare to each other?

Much work has been done in this field in recent years. There are many different approaches and ideas to SE for ML. Also, many reference architectures for ML-based systems have been proposed. This work aims at providing some answers to the questions mentioned above. We will provide an overview over different reference architectures and compare them based on different aspects to help with the selection of an architecture for different systems. As Machine Learning is a very broad topic, we chose to focus on systems around regression-based models. Of course, this restricts the generalizability of our findings on ML-based system in general, but in turn enables us to go more in-depth on this more specific topic.

"MLOps" is a term made up from "Machine Learning" and "DevOps". It means applying the concept of DevOps to ML-based systems. The focus of this work will be on the development part. We will also describe how we deployed our systems using Docker, but the main part will be about development.

To provide answers to the questions above, we first performed some literature research to find scientific literature about reference architectures for ML-based systems. After we found multiple architectures, we selected two of them and implemented a prototype system for each of them. Using the information we gathered and especially the prototype systems, we compared the found reference architectures while focusing on the two we selected for the implementation.

In the following we start by presenting our study design. We state our overall research questions and describe how we found and selected literature on the topic. Also, we explain our approach on implementation and how we did our final evaluation. Then we present the results of our literature research, describe the reference architectures we found and explain which ones we chose for implementation. Afterwards, we describe the use case we selected and discuss details of the implementation. Finally, we evaluate the reference architectures and implemented systems with regard to various aspects.

2 Related Work

There already are some papers that collect multiple reference architectures or architectural patterns for ML-based systems. For example, Washizaki et al. [WUKG20] conducted a systematic literature review and a did a survey among developers. As a result they present a list of different architecture patterns. This list also contains some of the architectures we will describe later on. They also present a list of design patterns and anti-patterns for ML. Finally, they analyze the connections between all the patterns and anti-patterns they collected to get an overview of how they are related to each other.

Martínez-Fernández et al. [MBF+22] performed a literature survey on Software Engineering for Artificial Intelligence (AI) to answer multiple questions. They collected work on characteristics, SE approaches and challenges of and for AI-based systems. This also includes architectures and design patterns for those systems. Concerning that, they name some studies that deal with this topics.

Lewis et al. [LOX21] present some challenges concerning software architectures for ML systems. For example, they state that when applying existing architectures to ML systems one may need to adapt them accordingly. They conclude that more work has to be done on this topic to find approaches to overcome new ML specific challenges.

Nahar et al. [NZL+23] also address challenges in the context of systems with ML components. In particular, they also summarize challenges evolving around the architecture of such systems. As the main challenges they state the transition "from a model-centric to a pipeline-driven or system-wide view", increasing complexity in design, difficulties "in scaling model training and deployment on diverse hardware" and a lack of taking "monitorability and planning for change" into consideration early enough [1].

For literature containing more detailed descriptions of a reference architecture, have a look at Section 4.1.1. There we list the papers in which we found the reference architectures described in Chapter 4.

Concerning MLOps, Matsui et al. [MG22] propose some useful steps to increase effectiveness in implementing systems. Those range from the way teams work to concrete tips for implementation and deployment. They emphasize that developers have to keep in mind that the shift towards MLOps brings new challenges and requires some adaption.

In contrast to the papers mentioned, we did not only collect reference architectures, but also directly compared them to each other and implemented prototype systems with two of the architectures. We do not only want to give an overview over some architectures, but also want to provide some results concerning the benefits and drawbacks of the architectures when implementing a system using them.

[1] p. 176 in [NZL+23]

3 Study Design

In this chapter, the study design is described. An overview is shown in Figure 3.1.

The study consists of two major parts: At first, some literature research has been done to find literature on the topic. From the found literature, multiple reference architectures were extracted for further steps. Secondly, two prototype systems using different architectures were implemented and afterwards evaluated regarding multiple aspects.

3.1 Research Questions

The overall research questions of this work are the following:

Which reference architectures for ML-based regression systems have been described in scientific literature? [RQ1]

How do these reference architectures compare to each other on our use case? [RQ2]

To answer these questions, a use case from this field of machine learning has been chosen. The research questions were then evaluated based on this use case which is described later in Section 5.1. To help answer RQ2, further research questions have been defined that can be answered for each of the reference architectures found during literature research:

What are the benefits of this architecture for our use case? [RQ2.1]

What difficulties does this architecture entail when applied to our use case? [RQ2.2]

By answering these questions for each reference architecture, RQ2 can be answered for the use case. Furthermore, the architectures can be compared to each other to get an indication of which one to use for a particular system.

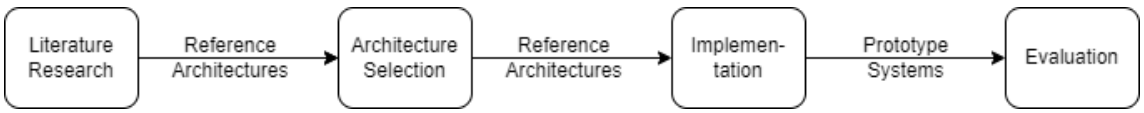

Figure 3.1: Overview of study design

3.2 Literature Research

To find literature on the topic, a lightweight literature review was performed. For this, an adaption of the Rapid Review methodology was used. A Rapid Review, as described by Cartaxo et al. [CPS20], is a secondary study that is performed in collaboration with practitioners working in the field of the study's topic. Compared to a Systematic Review, a Rapid Review is conducted in a shorter time omitting and simplifying some steps of a Systematic Review.

This methodology was adapted to the general conditions of this thesis so it can be applied by only one person and in the given time frame. The research questions have already been stated above. The search strategies used as well as inclusion and exclusion criteria will be described in the remaining part of this section. Reporting is done differently because the literature review is only one part of this work. The results are not summarized in an evidence briefing but in one of the following chapters.

Search Strategies

Two search strategies were used to find literature: At first, snowballing was performed starting from the proposal of this work. Additionally, a search using Scopus[1] was performed to find more literature. The following query was executed on June 19, 2024:

”MLOps” AND ”Reference Architecture” AND (LIMIT-TO (SUBJAREA , ”COMP”)) AND (LIMIT-TO (LANGUAGE , ”English”) OR LIMIT-TO (LANGUAGE , ”German”))

Inclusion and exclusion criteria

To decide on whether to further consider a paper, a set of inclusion and exclusion criteria was defined. For a work to be further considered it has to meet one of the inclusion criteria and none of the exclusion criteria.

These are the inclusion criteria:

[IC1] The work describes a reference architecture for ML-based systems

[IC2] The work contains at least a short summary of a reference architecture for ML-based systems

[IC3] The work contains a collection of reference architectures

So only papers about one or more reference architectures for ML-based systems are considered. For papers that meet IC2 or IC3, the search has to be extended onto work they reference concerning the mentioned reference architectures.

[1] https://www.scopus.com

The exclusion criteria are the following:

[EC1] The work is not written in English or German

[EC2] The work has been published before 2010

[EC3] The work does not fit the area of the chosen use case (see Section 5.1)

As Machine Learning is a rapidly developing field, papers that are too old might already be outdated and thus are excluded.

All papers found by applying the explained strategy were then evaluated further and the reference architectures described by them were extracted.

3.3 Implementation and Evaluation

After examining the found architectures, two of them were selected for implementation. For each architecture it is explained why it has been selected or not in its respective section in Chapter 4.

For implementation the goal was to build basic Machine Learning (ML) systems where a user can train, manage and use ML models using a simple web interface. One system was implemented for each of the two selected architectures. They both have the same requirements with the only difference being the underlying reference architecture to use.

After implementing the prototype systems, they were evaluated according to multiple criteria. Those were related to the development process, functional requirements and nonfunctional criteria of the implemented systems. In the following those three aspects of evaluation will be presented.

3.3.1 Development process

To evaluate the development process, we look at different aspects of the development of the two prototype systems. This includes the complexity of the systems, challenges that arose during development and how easy it was to test the systems during development.

3.3.2 Functional Requirements

The systems have to provide a web interface for the user to interact with them. Using this web interface, the user has to be able to complete the following tasks:

- Upload datasets for the specific use case in CSV format
- View the stored datasets
- Delete stored datasets
- Train models from the stored datasets
- View the stored models
- Edit the names and notes of stored models

- Delete stored models
- Evaluate stored models
- Make predictions from input data using the stored models

For this, the systems also need to store the uploaded data and trained models.

3.3.3 Nonfunctional Criteria

The nonfunctional criteria are not defined as strict requirements but rather as criteria to evaluate the implemented systems on. So we can compare the systems according to those criteria. Some of them have been inspired by ISO 25010 [ISO25010]. We will have a look at the following criteria:

- Response times: The time it takes for the systems to respond to specific requests
- Storage consumption: The storage used by the databases that hold the system's datasets and models
- Fault tolerance: How tolerant the systems are towards failure of single components
- Modularity: How strongly changes to one component affect other components of the system

4 Reference Architectures

In the following chapter, the process of finding reference architectures is described. Afterwards, the found reference architectures are presented. Also it is explained which of them were chosen to use for our implementation and why.

4.1 Metadata

In this section, the results of the conducted literature research are presented. The research consisted of two phases: Collecting literature and extracting reference architectures.

4.1.1 Collecting literature

As the first step of collecting literature, we performed snowballing from the proposal of this thesis. An overview of the results can be found in Figure 4.1. In the papers referenced by the proposal 62 further references were considered possibly relevant from their titles. After further inspecting those papers, 53 papers were excluded because they did not match any of the inclusion criteria, one was excluded because of EC1 and another one because of EC2. So after the first iteration, 7 papers were left.

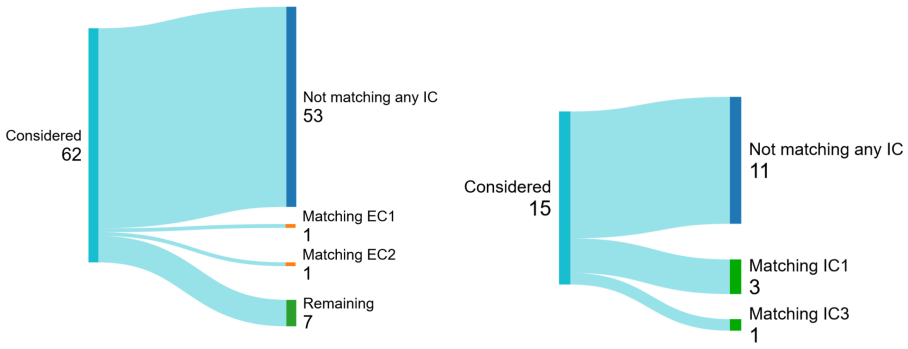

Figure 4.1: Results of Snowballing. Left: First step. Right: Second step.

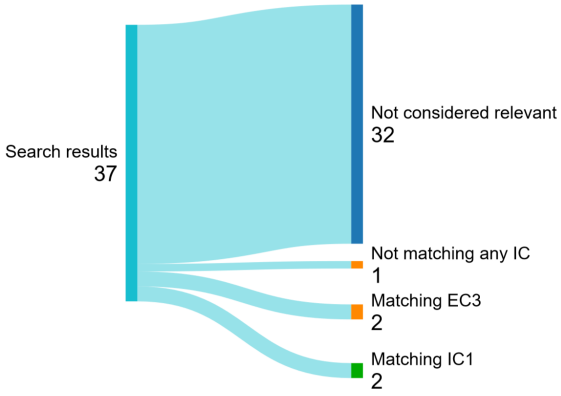

Figure 4.2: Results of Scopus search

A second iteration on these papers' references found 15 new possibly relevant papers. 11 of those were excluded after closer examination because they did not match any of the inclusion criteria. Of the four remaining papers, three contained descriptions of reference architectures. These are the following:

- "Machine Learning System Architectural Pattern for Improving Operational Stability" by Yokoyama [Yok19]
- "N-Version Machine Learning Models for Safety Critical Systems" by Machida [Mac19]
- "Solution Patterns for Machine Learning" by Nalchigar et al. [NYO+19]

The remaining paper by Serban et al. [Ser19] referenced multiple papers about reference architectures.

Additionally, we did some direct search using Scopus. A summary of the results can be found in Figure 4.2. It led to 37 results, of which 5 were considered possibly relevant by their titles and abstracts. One of those papers was excluded after closer examination because it did not match any of the inclusion criteria. Another two papers were excluded because they matched EC3. The two remaining papers both contain a description of a reference architecture:

- "Distributed artificial intelligence: Taxonomy, review, framework, and reference architecture" by Janbi et al. [JKM23]
- "An Efficient Microservices Architecture for MLOps" by Roh et al. [RJCH23]

So in total, six papers matching the criteria were found during literature research. Five of them contain a description of a reference architecture, and the remaining one contains references to architectures.

4.1.2 Extracting Reference Architectures

All but one of the five papers contain a description of one reference architecture. The paper by Yokoyama et al. [Yok19] contains two architectures: One rather simple and another that builds on top of the first one. Not all architectures had a specific name in the papers they were described in. For those we made up the names and chose them as descriptive as possible.

This are the six extracted reference architectures:

- Three-layer Architecture [Yok19]
- Multi-layer Architecture [Yok19]
- N-version Architecture [Mac19]
- Nine-part Architecture [NYO+19]
- Imtidad Reference Architecture [JKM23]
- Microservice Architecture [RJCH23]

They are described in more detail in the following.

The papers on further architectures referenced by [Ser19] were all excluded because they match EC2.

4.2 Three-layer Architecture

This architecture separates tasks into three layers. Those layers are the following: Presentation Layer, Logic Layer and Data Layer. An overview of this architecture's structure can be found in Figure 4.3.

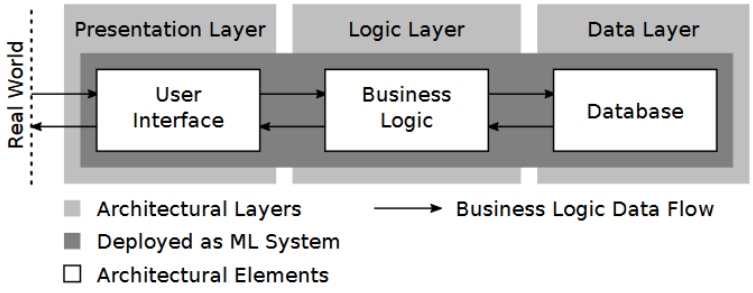

Figure 4.3: Three-layer Architecture. Source: [Yok19]

Presentation Layer The Presentation Layer is responsible for the connection to the real world. It provides a user interface for users to interact with the systems. For this, it can request information to show to the user from the Logic Layer.

Logic Layer This layer is the center of the system. It contains the core business logic and thus handles all calculations and operations. It has to handle requests coming from the Presentation Layer and can in turn request data from and send data to the Data Layer.

Data Layer The Data Layer manages all data handled by the system. It has to provide data access to the Logic Layer and store data persistently. Usually it is realized using a database.

The Three-layer Architecture has been chosen for implementation because of its simplicity and general applicability.

4.3 Multi-layer Architecture

The Multi-layer Architecture is an extension of the rather simple Three-layer architecture. The idea of separating the system into three layers stays, but each layer now consists of more than one component. While the general Three-layer Architecture can be used in many other fields besides Machine Learning, the Multi-layer Architecture is designed specifically for ML-based systems. Another big difference is that some of the additional components are only needed for creating models, which is assumed to be done during development by the authors who proposed the architecture. So those components are not included in the final resulting system. Figure 4.4 shows the overall structure of this architecture.

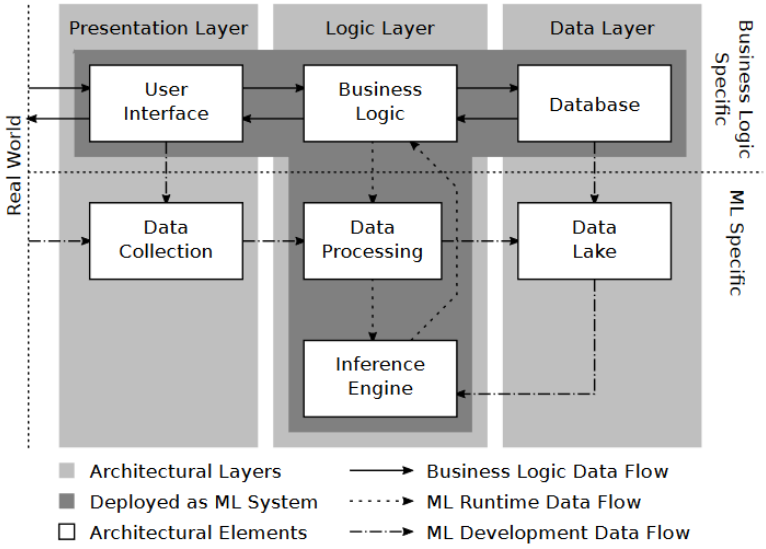

Figure 4.4: Multi-layer Architecture. Source: [Yok19]

The User Interface, Business Logic and Database components are very similar to those from the Three-layer Architecture. For the Machine Learning workflow the following four components have been added in comparison to the Three-layer Architecture:

Data Collection The Data Collection component is a part of the presentation layer. But it is not necessarily meant for the user to interact with it. Its purpose is to collect data from input data sources (e.g. existing datasets or a web crawler) or a user interface. As this only is a part of the development process of the system, the Data Collection component is not part of the final production system but only of the development system.

Data Processing This component processes incoming data and makes it ready to be used for machine learning purposes. That includes the data collected to create the inference engine as well as data provided by a user to get a prediction from. It is part of the logic layer.

Inference Engine The inference engine is the core of the Machine Learning part of the system and therefore belongs to the logic layer. It is created during development using the preprocessed data stored in the data lake. In the production system, it is then used to make predictions from data input by the user.

Data Lake The data lake belongs to the data layer. It receives all the data from the Data Processing component after processing and stores it so it can later be used to create the inference engine. As the data stored by the data lake are only used to create the inference engine, it is only part of the development system and not of the final production system.

Although Yokoyama et al. [Yok19] explain how this architecture is better for ML systems than the Three-layer Architecture, it was not chosen for implementation in this work. That is because the focus of this work is to implement simple prototype systems. For small systems the separation of each layer into multiple components adds more complexity than what would be necessary. Also, the Multi-layer Architecture does represent a slightly different approach on how to build the system compared to ours for this thesis. It assumes that models are trained during development which is not true for the systems developed here.

4.4 N-version Architecture

The idea of this reference architecture originates from N-version programming. In N-version programming, N different programs (with $N \geq 2$) are produced that all follow the exact same specification and provide the same functionality. This is mainly used in extremely critical applications where any error can be a huge problem. By using multiple different programs, the risk of the whole system yielding an error in a situation is lowered.

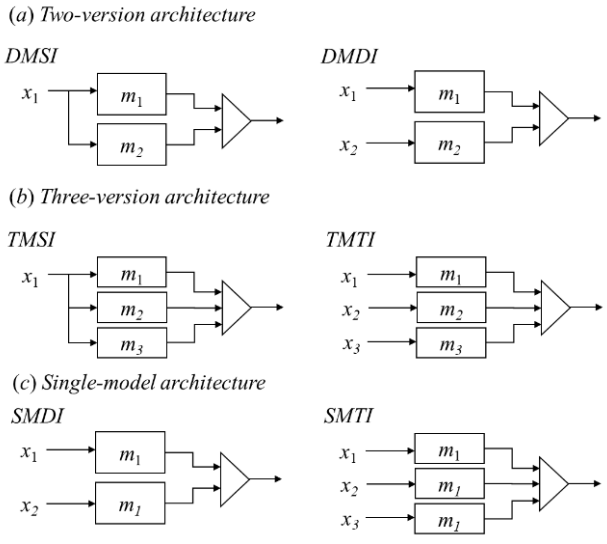

Figure 4.5: Multiple variations of the N-version Architecture. Source: [Mac19]

The N-version Architecture applies this idea to Machine Learning models. But as there are differences between ML models and general software components, it has to be modified. This leads to two parts that can be used in *N* different forms for better results. Firstly, instead of using only one model, you use multiple ones. So if one model produces an error, the system still will provide a valid result. Secondly, one can use different inputs either for different models or for only one model. This can mitigate issues caused by specific sets of input data. For further explanations and a reliability analysis refer to the work by Machida [Mac19].

Some of the possible variations of the N-version Architecture are shown in Figure 4.5. For example, there are two variants of a two-version architecture, one where both models receive the same input (Double Model with Single Input, short DMSI) and one where the models receive different inputs (Double Model with Double Input, short DMDI). Equivalent variants exist for a three-version architecture, they are called Triple Model with Single Input (TMSI) and Triple Model with Triple Input (TMTI). Other variants include single-model architectures with multiple inputs like Single Model with Double Input (SMDI) and Single Model with Triple Input (SMTI).

This architecture has not been chosen for implementation because it is a very general one that does not propose a concrete structure of the system but rather a concept that can be implemented in other architectures.

4.5 Nine-part Architecture

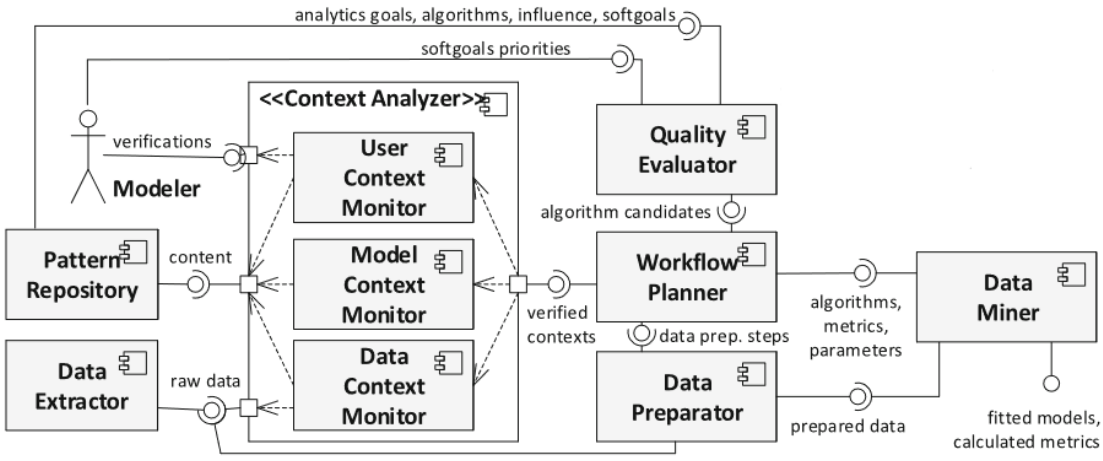

Figure 4.6: Nine-part Architecture. Source: [NYO+19]

This architecture consists of nine components that together make up a system for applying solution patterns for machine learning. Figure 4.6 shows a UML component diagram of its structure. In the following, the singular components will be explained:

Pattern Repository Stores multiple solution patterns in a standardized format

Data Extractor Extracts the relevant data from incoming raw data files

Content Analyzer Consists of three components that belong together: **User Context Monitor**, **Model Context Monitor** and **Data Context Monitor**. It can generate a content analysis workflow by parsing a given pattern's structure. The three Context Monitor components verify the data from the user, the data extractor and the model configurations respectively.

Quality Evaluator Evaluates algorithms based on non-functional requirements and makes recommendations on which ones to use. It receives requirements and knowledge on algorithms from the Pattern Repository component and requirement priorities from the user.

Workflow Planner Suggests an order of the actions to be executed by the Data Preparator and Data Miner components. Also it ensures that the required prerequisites are fulfilled.

Data Preparator Prepares data for further processing by the Data Miner. It requires the Workflow Planner to provide a list of steps to perform with the data.

Data Miner Executes machine learning algorithms on input data, stores models and calculates evaluation metrics. It also requires the Workflow Planner to plan its steps.

For more detailed explanations and background information refer to the work by Nalchigar et al. [NYO+19].

This reference architecture has not been chosen for implementation because it is rather complex and would be out of scope for our more simple use case.

4.6 Imtidad Reference Architecture

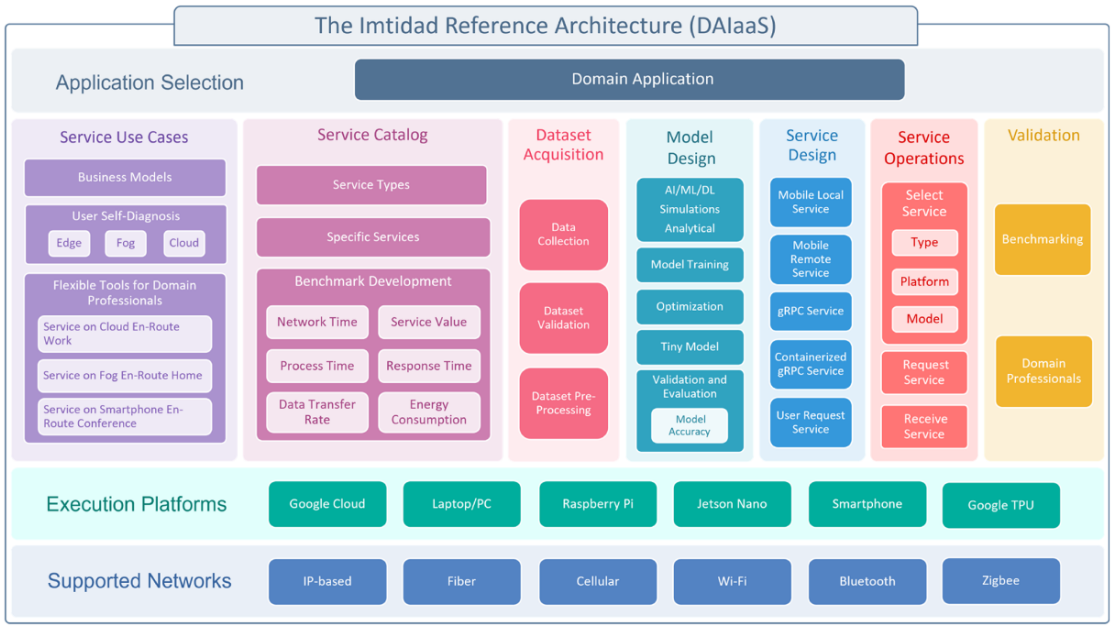

Figure 4.7: The Imtidad Reference Architecture. Source: [JKM23]

The Imtidad Reference Architecture is used for Distributed Artificial Intelligence (DAI) systems, more precisely DAI as a Service (DAIaaS). It consists of many services that can talk to each other independently. Figure 4.7 shows an overview of all the services and components included in the architecture. As this is a complex architecture and out of scope for this thesis, it will not be described in detail here. For more information refer to the work by Janbi et al. [JKM23] who proposed this architecture.

This architecture also has not been chosen for implementation due to the reasons already mentioned above.

4.7 Microservice Architecture

When using a Microservice Architecture, the system is split into multiple separate services where each service has a specific task. So for example, there could be one service that is responsible for training new models while another service is used to make predictions on those models. Also each service has its own database or even two: One for reading data and one for manipulating data (Create, Update and Delete operations). Because of multiple databases now holding the same data, they have to be synchronized regularly so each service receives each update on the data made by another service. For more detailed information and explanations consider the work by Roh et al. [RJCH23].

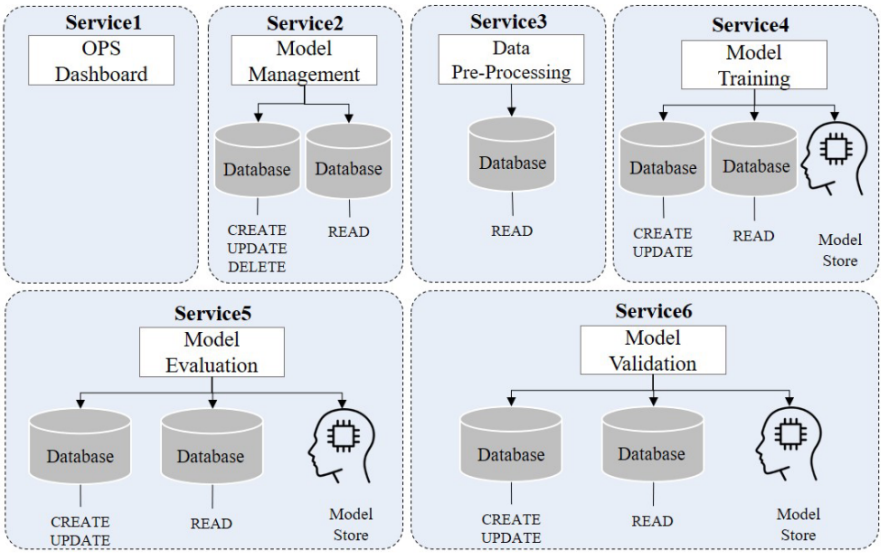

Figure 4.8: Microservice Architecture. Source: [RJCH23]

An exemplary MLOps system using a Microservice Architecture is depicted in Figure 4.8. It is split into six different services. Each service has its own database(s) and model store according to its needs. The OPS Dashboard service handles interactions with users while the other services perform different tasks concerning training and managing models.

The Microservice Architecture has been chosen for implementation because it is a common architecture and is rather simple to implement compared to most of the other presented architectures.

5 Implementation

In this chapter, details of the implementation are discussed. This includes a description of the chosen use case and the implemented systems. We will describe their structure, the technologies used for implementation and some other details.

5.1 Use Case

As the goal of this work is to compare reference architectures for systems based on regression models, the use case should be a problem solvable using basic regression models. So the findings can be generalized onto common regression problems.

The chosen use case is a simple prediction system that can estimate a ship's fuel consumption based on current weather and operational conditions. It can be found on GitHub[1].

The system has nine input features:

- wind speed ($\frac{m}{s}$)
- wind direction (°)
- significant wave height (m)
- peak wave period (s)
- wave direction (°)
- primary swell height (m)
- primary swell period (s)
- primary swell direction (°)
- course over ground (°)

The only output feature is the fuel mass flow in $\frac{kg}{s}$.

The authors of the GitHub repository provide a large example dataset which was used to test the implemented systems. They also provide code for training a neural network. Instead of the provided code, a simpler approach using a linear regression model was used for implementation.

[1] https://github.com/UH-MLOps/ship-performance-model

The implemented systems each provide a web interface and offer the following features to the user (according to the functional requirements stated in Section 3.3.2):

- Upload data: The user can upload CSV files containing datasets for the above use case
- Manage data: The user can view all stored data and delete them
- Train models: The user can train models using the stored data
- Manage models: The user can view all stored models, edit their metadata and delete them
- Evaluate models: The user can upload additional datasets to evaluate the stored models on
- Make predictions: The user can enter input data to make a prediction using one of the stored models

5.2 Underlying Technologies

The systems were implemented using Python for the back-end. To run the web servers, the Flask[2] and waitress[3] modules have been used. Further Python modules that have been used are:

- mariadb[4] to connect to databases
- pandas[5] for managing data
- scikit-learn[6] for all ML operations

and their respective dependencies.

Docker is used for deployment and running the systems. Full Docker configurations have been created for both systems. The databases have been set up using the mariadb image[7]. Each system also contains an instance of PhpMyAdmin which is instantiated from the phpmyadmin image[8]. All other containers run the implemented web servers and other required services. They are created from custom Docker images. All custom Docker images use a common base image which is derived from the ubuntu image[9]. Additionally, Python, pip, venv[10] and MariaDB Connector/C[11] have been added to this image.

Each of the custom images is configured using a Dockerfile. For each whole system we created a Docker Compose specification that sets up all required containers.

[2] https://pypi.org/project/Flask
[3] https://pypi.org/project/waitress
[4] https://pypi.org/project/mariadb
[5] https://pypi.org/project/pandas
[6] https://pypi.org/project/scikit-learn
[7] https://hub.docker.com/_/mariadb
[8] https://hub.docker.com/_/phpmyadmin
[9] https://hub.docker.com/_/ubuntu
[10] https://docs.python.org/3/library/venv.html
[11] https://mariadb.com/docs/server/connect/programming-languages/c/

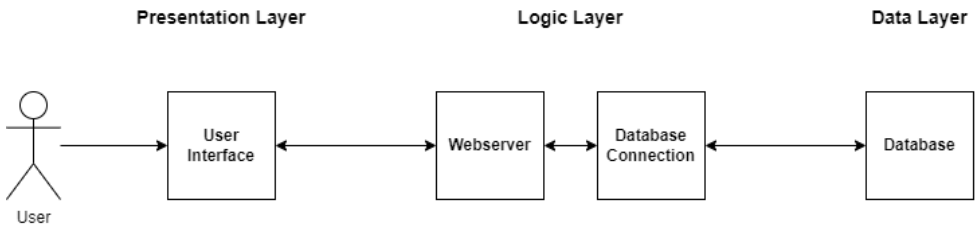

Figure 5.1: Structure of the Three-layer Architecture system

5.3 Three-layer Architecture

According to the reference architecture, this system is made up of three parts: Presentation layer, logic layer and data layer. The structure of the system is shown in Figure 5.1.

The presentation layer consists of the user interface which is provided as HTML and JavaScript code. It processes the user's inputs and sends according requests to the web server. On receiving the server's responses, it displays the results for the user. The sent HTTP requests and responses are the link between presentation layer and logic layer.

The logic layer mainly consists of the web server that receives requests from the presentation layer, determines a response and sends it back. The web server provides methods for all parts of the user interface as well as the system's funtionality. This includes adding data, managing existing data, training models, managing models, evaluating models and making predictions on existing models.

The logic layer is connected to the data layer using a DatabaseConnection component that handles all communication with the database. This database makes up the data layer. It contains two tables, one for the data and one for the models.

An exemplary interaction of a user with this system is shown in Figure 5.2. At first, the user enters the URL of the data overview page into their browser's address bar or clicks a link leading to that page. Then the browser sends a request to the webserver to get this page. When the webserver receives this request, it sends a response containing the user interface for the data overview page. Upon being loaded, the user interface sends another request to the webserver to get the data to display. To handle this request the webserver has to get the data from the database via the Database Connection component. After receiving the data, the webserver formats it into JSON format and sends it back as a response. Finally, the user interface displays the data in a table.

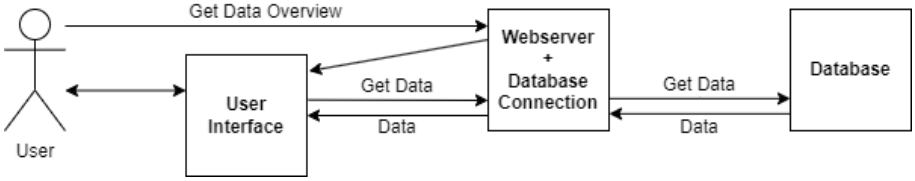

Figure 5.2: An exemplary interaction of a user with the Three-layer Architecture system

5.4 Microservice Architecture

In the Microservice Architecture system there is no central web server that handles all requests. Instead there are multiple ones for different tasks. The implemented system differs from the exemplary system shown in Section 4.7 in some aspects. For example, it does not use a Model Store but stores the models in a regular database table. Also, it does not separate read and write databases as that would have caused too much overhead for the rather small system.

Figure 5.3 shows the structure of the implemented system. The split into the services is really fine granular. This would not be absolutely necessary, one could also bundle all operations on datasets in one service and all operations involving models in another one. The reason for using this more fine granular approach was the description of the architecture by Roh et al. [RJCH23] who also split the system into more smaller services in their example.

In total, the system consists of eleven services which are described in the following.

Dashboard

The dashboard service provides the user interface. It is the only service the user themselves needs to know as all further services are only used by the JavaScript code provided from the dashboard. It has a similar task to the presentation layer in a Three-layer Architecture.

The service is a simple web server that only serves the requests to provide the user interface. To do this, it constructs the requested pages from HTML templates. It stores all the HTML, JavaScript and CSS code of the whole system.

Predict

This service is responsible for making predictions on existing models. It can load the model selected by the user from its database and make predictions on it.

The Predict service consists of a web server and a database manager. The web server provides a method to predict a value based on the input values from a form. The database manager is responsible for establishing a connection to the database and retrieving models to make predictions on.

Get Data

This service's sole responsibility is to read stored data from its database and provide it to the user. It is build similar to the Predict service with a web server for handling requests and a database manager to read data from the database.

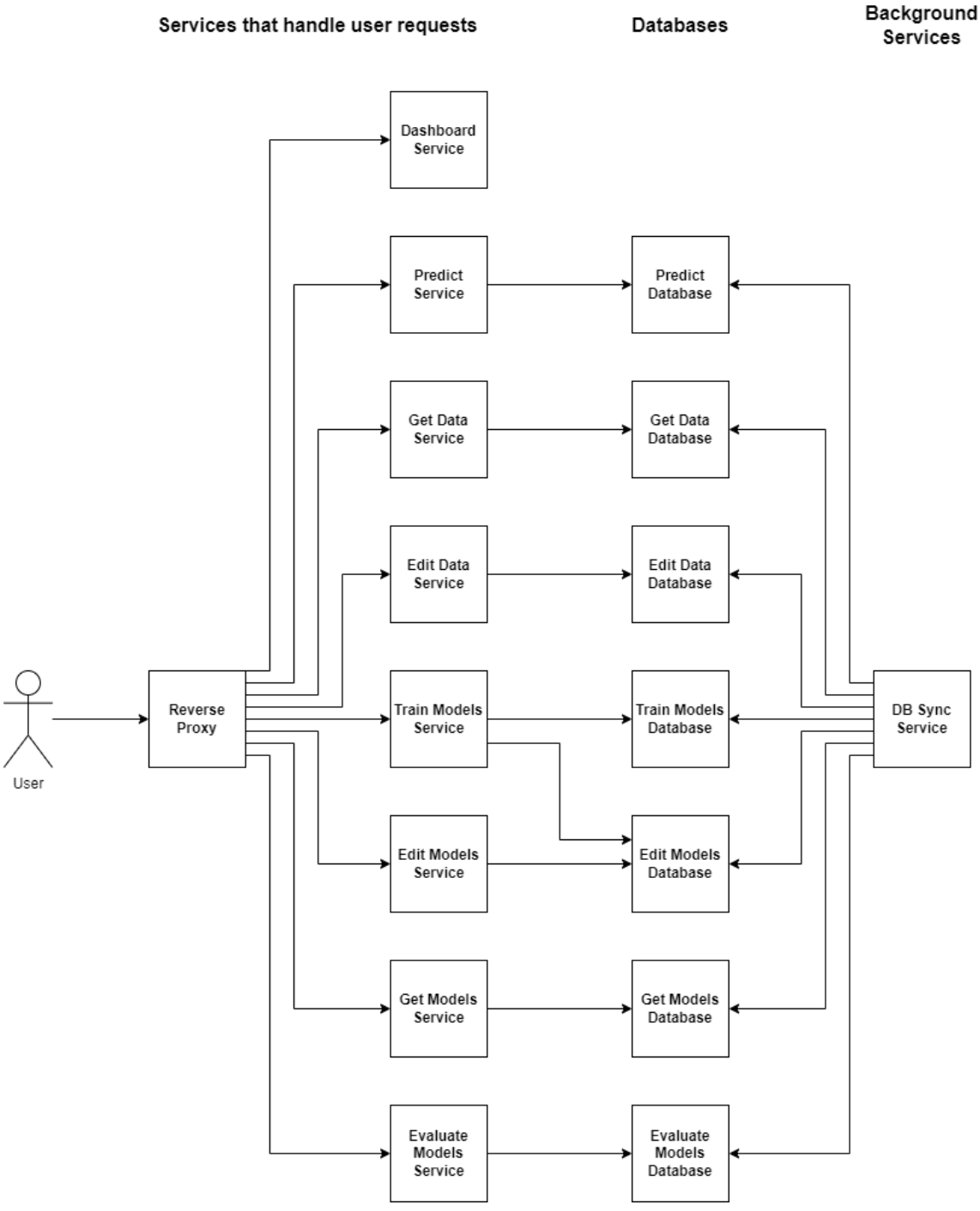

Figure 5.3: Structure of the Microservice Architecture system

Edit Data

As its name says, this service is used to edit the stored data. This includes adding data uploaded by the user and deleting existing data.

This service is a bit more complex than the previous ones as it provides functionality for adding data as well as deleting data. Those were combined in one service so there is only one service that has to write changes to the Data databases.

Like the previous services it consists of two major components: A webserver that handles incoming HTTP traffic and a database manager that is responsible for connecting to the service's database.

Get Models

Similar to the Get Data service, this service is responsible to read stored models from its database and return it to the user. Thus it also built similar to the Get Data service.

Edit Models

This service has two tasks: Editing and deleting existing models. Editing a model includes changing its name and notes to new values provided by the user. Concerning deletion, the service can either delete a set of specific models or all models at once. Like most other services it consists of a web server that handles incoming HTTP requests and a database manager that manages the service's connection to its database.

Train Models

The Train Models service is responsible for training new models. Its web server provides two methods: One for training a model from the existing data and one that requires an uploaded file with data to train the model on. It also has a database manager like the other services.

Unlike the other services it interacts with two different databases: It uses the Train Models Database to get the data to train a model from, but writes trained models to the Edit Models Database. That is due to our database setup with master and replica databases where changes may only be written to a master database. How this works in detail is described later.

Evaluate Models

This service is used to evaluate the stored models. For this, the user has to upload a new dataset to test the models on. This dataset is only used for evaluation one time and not stored permanently.

The service provides two methods: One for evaluating a specific model and one for evaluating all stored models. They calculate the R^2 score for the specific model or all models respectively. The R^2 score is a simple metric to measure the accuracy of a model's predictions. The best score a model can get is 1.0, but there is no lower boundary. We used the method `RegressorMixin.score`[12] of scikit-learn to calculate the scores.

Like all other services that interact with a database, this service has a database manager that handles all database interactions.

Reverse Proxy

The reverse proxy service ensures that all of the above services can be reached from the same address from the outside. It listens to this address and forwards each request to its responsible service. It is implemented using a nginx webserver that is configured to forward requests to specific URLs to their corresponding internal addresses. For example, all requests to URLs starting with `localhost:5001/get_data` are forwarded to the Get Data service which has the internal address `msa-get-data:80`.

Database

The database service is responsible for storing all data and models. It contains multiple databases so each of the above services can have its own database. To simplify the system and its deployment, there is only one container running an instance of MariaDB that contains all databases. But it would easily be possible to run them all in their own respective containers.

In total, there are seven databases. They are listed in Table 5.1.

Each database either belongs to the Data or Models category. For each category there is exactly one master database. It is the only database that the previously described services may write changes to. The remaining databases serve as replica databases which means that they just replicate the data stored in their respective master database. To synchronize the replica databases with their respective masters, an additional Database Synchronizer service is used which is explained next.

Name	Category	Type
Edit Data DB	Data	Master
Get Data DB	Data	Replica
Train Models DB	Data	Replica
Edit Models DB	Models	Master
Predict DB	Models	Replica
Get Models DB	Models	Replica
Evaluate Models DB	Models	Replica

Table 5.1: Databases

[12] https://scikit-learn.org/stable/modules/generated/sklearn.base.RegressorMixin.html#sklearn.base.RegressorMixin.score

Database Synchronizer

As each service has its own database, those databases have to be synchronized. There is exactly one master database for the stored data and one for the stored models. Both for data and models there exist multiple replica databases. This service synchronizes the replica databases regularly to reflect all changes made to their respective master database.

The synchronization is implemented by using an additional table called Queries in the master databases. It stores all queries executed on the database's Data or Models table that changed its state. The Database Synchronizer service regularly checks for entries in the Queries table of each master database. If there are any entries, they are retrieved and executed on all respective replica databases.

Additionally, the Database Synchronizer service creates and initializes all required databases on start up if necessary.

An interaction of a user with the implemented system could look like the one depicted in Figure 5.4. At first, the user enters the URL of the data overview page into their browser's address bar or clicks a link that refers to this page. This leads to a request being issued to the reverse proxy which forwards it to the Dashboard service. On receiving the request, the Dashboard service returns the code for the user interface which is then displayed by the user's browser. When displayed, the user interface sends a request to the reverse proxy to get the data to display. The reverse proxy forwards this request to the Get Data service which in turn queries its database for the data. After receiving the data from the database, the Get Data service formats them into JSON format and sends them back as a response. On receiving this response, the user interface creates a table containing the data and displays it to the user.

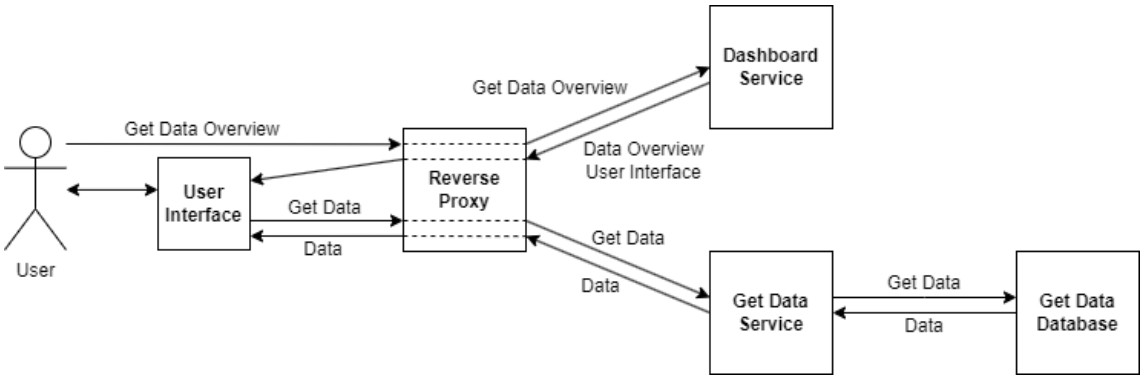

Figure 5.4: An exemplary interaction of a user with the Microservice Architecture system

6 Evaluation

In this chapter, the work done so far is evaluated with regard to various aspects. This includes the development process of the two implemented systems as well as functional requirements and nonfunctional criteria concerning those systems. Finally, the stated research questions (see Section 3.1) will be answered based on the findings of this work.

6.1 Development Process

In this section, we will have a look at various aspects related to the development processes of the two systems.

System Complexity

The development process of the Three-layer Architecture (TLA) system was rather straightforward: We set up a web server and a database and created a simple user interface. From this point we added the features one by one, always including its server code and the part of the user interface required to use it.

In turn, the development of the Microservice Architecture (MSA) system was more complicated. At first, we had to separate all functionality into services. Then we set up the database and reverse proxy. Afterwards, we were able to create the Dashboard service with a simple user interface. Only then we could start adding the real functionality by creating the services one by one. Additionally we had to implement the Database Synchronizer service at some time to be able to test all the other services properly.

Challenges

During the development of the TLA system we faced no architecture specific challenges, but only general ones, especially concerning lack of knowledge in some aspects of the tools we used. We did not face them anymore when implementing the MSA system, because we implemented the TLA system first.

The MSA system in turn provided some system specific challenges. One of them was to find a good separation of functionality into services. Another difficulty was the necessity to create a service that synchronizes the databases that, in some cases, has to handle very large amounts of data in short time.

Testing

Testing the TLA system was rather simple as the web server can easily be run outside the Docker configuration when the database container is configured to expose its port to the host system. This was not possible for the MSA system because it consists of more components and would require some changes to be able to run without Docker. Also, there were some prerequisites that had to be fulfilled before we were able to test some features at all. For example, the Database Synchronizer service had to be implemented before most of the other service could even be tested properly.

Summary

In total, the development of the MSA system was more difficult than the development of the TLA system. That is in line with what we would have expected, as the TLA is a very simple architecture that can be utilized for many different use cases but maintains a clear separation into its layers nevertheless. The MSA on the other hand is more complex and requires additional planning concerning the breakdown of a system into services.

6.2 Functional Requirements

The functional requirements for the systems are stated in Section 3.3.2.

Both implemented systems fully meet the stated functional requirements. They provide the same functionality with some differences in the structure of the user interface. These differences originate from the differing structures of the systems according to their reference architectures. But they do not alter the core functionality of the systems.

6.3 Nonfunctional Criteria

In this section, the two systems will be compared according to the nonfunctional criteria stated in Section 3.3.3.

Response Times

We measured the systems' response times to some specific requests to compare them on how fast they respond. We did not perform any in depth measurements but only some approximate ones to get a notion on whether one of them is considerably faster than the other in some regard. Five tasks were chosen to measure the response times for:

- Upload data: We uploaded a sample dataset containing 517500 rows of data (about 48.4 MB in size)
- Show data overview: Load an arbitrary page of data which contains 10 rows
- Train a model: Train a model on the uploaded sample dataset

- Make a prediction: Make an arbitrary prediction on the trained model
- Delete all data: Delete all stored data at once

The results can be found in Table 6.1. It is to remark that for the MSA system it takes some additional time until all data can be accessed from the overview and used for model training after the upload is completed. We measured this additional time to be around 170s for our sample dataset. This originates from the database separation in this system. The uploaded data have to be added to all databases before they can be fully accessed by the databases' corresponding services which takes some time. The same issue occurs when deleting all data. But the delay is much lower then.

	TLA system	MSA system
Upload data (517500 rows)	105s	125s
Load one page of data (10 rows)	1s - 1.5s	1s - 1.5s
Train a model	4s	5.5s
Make a prediction	Near 0.0s	Near 0.0s
Delete all data	10s	11s

Table 6.1: Approximate average response times of our systems

In general, all response times except the one when uploading data are about the same for both systems. The time it takes to upload the data is not that much higher for the MSA system either. But the delay until the data are fully synchronized between all databases more than doubles the total time required for the data to be fully accessible by every service. If the system does not have to handle that much data, this is no huge issue. But for large amounts of data, this is a clear disadvantage of our MSA system.

As we only implemented prototype systems that might not be optimized well, we can not say that the required synchronization has to be that much of an issue in every system using the MSA. If one uses a much faster approach to synchronize databases, this drawback could be eliminated completely.

Storage Consumption

To measure the database's storage comsumptions, we noted the size of the Docker volumes that hold the database files after each operation we performed.

We noticed that the stored models take up almost no storage space so the datasets are the only relevant aspect. To test their storage consumption, we uploaded some sample data which consisted of 517500 rows and was about 48.4 MB in size as a CSV file. The respective storage space taken up by the databases is listed in Table 6.2. The values were not exactly the same each time, so we can only give an approximation.

	TLA system	MSA system
Initial	160 MB	160 MB
Storing 517500 rows of data	400 MB	900 MB

Table 6.2: Approximate storage consumption of our systems

After deploying the systems, their databases took up about the same of storage space. But after uploading a large amount of data, the values differ alot. That is plausible because the MSA has multiple databases holding the same data which leads to a larger amount of required storage space.

Fault Tolerance

In the TLA system, the failure of one of the three layers leads to a dysfunction of the whole system as almost any request by the user goes through all layers. The failure of one the services of the MSA system however does not always lead to an error of the whole system. Whether this is the case depends on the affected service. There still are some services the system depends on to be able to operate. Those are the Reverse Proxy, Database, Database Synchronizer and Dashboard services. [1] If one of the other services failes, the remaining part of the system that does not use this service is still able to operate normally.

Thus, in general, the MSA is more tolerant against the failure of one of its components than the TLA system (at least for some components).

Modularity

Obviously the MSA system has a higher degree of modularity than the TLA system as modularity is a core property of the Microservice Architecture. While changes in one of the core services (Reverse Proxy, Database, Database Synchronizer and Dashboard) still have an impact on the rest of the system, changes in other services mostly do not affect the rest. A change in a part of the TLA system most likely also affects other parts of the system as all parts are more interconnected and also share more code.

6.4 Results

This section summarizes the evaluation results and provides answers to the research questions stated in the beginning.

6.4.1 What Reference Architectures are there?

The first research question of this work was: “Which reference architectures for ML-based regression systems have been described in scientific literature?”

During the performed literature research we identified six different reference architectures. Some of them are rather specific to a single use case. An example for this would be the Imtidad Reference Architecture. So some architectures can not be used for the general case of ML-based regression systems. On the other hand, some architectures are applicable in many different scenarios. This includes most of the other presented architectures.

[1] At least for the Database service, this could easily be changed if one would use multiple services for the individual databases.

Also, some of the architectures are ML specific while others can also be used for non ML systems. For example, the Three-layer Architecture and Microservice Architecture are also used in fields outside of ML. But there is potential for optimizing such architectures for ML by doing some adjustments. A good example for this would be the Multi-layer Architecture which is a ML specific variant of the more general Three-layer Architecture. Some reference architectures are not even a full architecture that alone can be used to implement a system but instead provide an idea that can be implemented in another architecture. The N-version Architecture is one of those as it just provides an idea on how to use models but not how to build a system around those models.

In conclusion, plenty of different reference architectures for ML-based regression systems are described in scientific literature. Some of them have been described in Chapter 4. Of course, this is no complete list, but it should provide some overview.

6.4.2 How do they compare?

The second research question was: "How do these reference architectures compare to each other on our use case?" It was stated to be evaluated utilizing some additional questions for each reference architecture concerning its benefits and difficulties in our use case.

Of the six reference architectures we described in Chapter 4, we found three not being suited well for our use case after closer examination. Those were the Multi-layer Architecture because it assumes that models are already trained during the system's development what is not the case in our application, the Nine-part Architecture because it is too complex for our rather simple use case, and the Imtidad Reference Architecture which also is too complex and does not quite fit our use case.

One of the remaining reference architectures, the N-version architecture, is no fully fledged architecture on its own but rather a concept of utilizing multiple models and input data variants. This idea can be applied whilst using another architecture as the basis. It would be possible to make use of this concept in our implemented systems, for example by letting the user choose multiple models to make a prediction on at the same time.

So there are only two fully fledged reference architectures left that are suited well for our use case. Both of them are general architectures that can be used in many different scenarios and that are not made specifically for ML systems. They also are the architectures we chose for implementation. In the following we will mainly compare those two architectures.

The main benefits and difficulties we found concerning our implemented TLA and MSA systems are summarized in Table 6.3 and Table 6.4 respectively.

Benefits	Difficulties
• Easier to implement • Less storage consumption	• Highly interconnected / low modularity

Table 6.3: Main benefits and difficulties of our Three-layer Architecture system

Benefits	Difficulties
• Higher fault tolerance in general	• Requires additional planning • Additional system requirements (e.g. reverse proxy, database synchronization) • Database synchronization takes some time for large amounts of data

Table 6.4: Main benefits and difficulties of our Microservice Architecture system

In summary, we see more benefits for our use case when using the TLA. First and foremost, it is easier and more straightforward to implement. But it also surpasses the MSA regarding the required storage space and does not need additional time to synchronize data between databases after adding or deleting data. For smaller amounts of data, the difference in storage consumption and required time is rather small and no big issue, but of course the more difficult development process remains. As we already stated earlier, our database synchronization process surely is not the most efficient one. There might be approaches that can reduce the additionally required time significantly which would eliminate this drawback of the MSA.

The TLA also showed some drawbacks in our prototype systems. Generally, they are attributable to the more interconnected, non-modular structure. This makes components more dependant on each other and reduces the overall fault tolerance of the system. But this drawbacks are minor compared to the benefits the TLA has over the MSA in our use case.

For larger systems the benefits of modularity might weigh much higher than they do for small systems like the ones we implemented. This can move the decision on the architecture towards the MSA but does not necesssarily have to as the easier development process always is a huge advantage of the TLA.

6.5 Threats to Validity

All of the work has been done by one person which might lead to a bias concerning the evaluation. Knowing all internals of the implemented systems might influence how they are perceived as a whole during evaluation. Prior knowledge or experience might also influence how the system are seen because one already has a preferred architecture.

The finished systems were evaluated according to fixed criteria that can be assessed objectively where personal bias should not play too big of a role. The evaluation of the systems' development process has to be done by someone who has worked on the development theirselves which leaves no choice but to accept some potential bias there. But the overall influence of a potential bias by the author should not be that high.

Also, we only implemented systems for one specific use case of regression-based ML. But we would argue that this is no issue, because a huge part of our systems can generally be used for this type of application. One would only have to change some specifics like column names and column count everywhere in the systems to use them for another regression-based use case.

7 Conclusion

The goal of this work was to identify possible reference architectures for Machine Learning based systems and evaluate their benefits and challenges with regard to a chosen use case. For this we performed a literature review to find scientific literature about the topic. Then we selected two of the found architectures for the implementation of two prototype systems. Finally, we compared the reference architectures with regard to various aspects while focusing on the two ones we chose for implementation. We presented multiple suitable reference architectures and did a comparison between some of them to aid in the selection of an architecture for some ML-based systems.

We found that there already is some literature about reference architectures for ML-based systems, but some of the architectures are only applicable for a rather specific use case. Many ideas from general Software Engineering have been transferred to the ML context. Thus many of the proposed reference architectures for ML are also used generally or derived from architectures for general software systems.

The main architectures we had a look on were the Three-layer Architecture and the Microservice Architecture, which both are also used outside the field of ML. Comparing them, we found the TLA to have more benefits for systems like the ones we implemented. Our TLA system was generally easier to implement and confronted us with less challenges than the MSA system while we found its main drawback to be the high amount of interconnection between system parts. But for other systems the different aspects of our evaluation might weigh differently so the MSA might be more beneficial. To find out more about that, further research is needed.

Outlook

As other authors have already stated, there still is much work to be done to achieve today's standards in Software Engineering also for systems using ML. One possible approach would be to further research the applicability of existing reference architectures in the context of ML or how existing architectures can be adapted to ML systems. An example for an architecture that has been adapted to ML-based systems is the Multi-layer Architecture we presented in Section 4.3 which is based on the Three-layer Architecture. Similar approaches could also be explored for other existing reference architectures.

We only evaluated some architectures based on a single use case. So there is much potential to continue our work. One could evaluate the same architectures based on different use cases to check whether our results can be generalized. Further research could also include more architectures for the same use case to find out whether there are ones that might be better suitable than those we found. Especially the implementation of more prototype systems with other architectures would be helpful in that regard.

Bibliography

[CPS20] B. Cartaxo, G. Pinto, S. Soares. "Rapid Reviews in Software Engineering". In: *Contemporary Empirical Methods in Software Engineering*. Ed. by M. Felderer, G. H. Travassos. Cham: Springer International Publishing, 2020, pp. 357–384. ISBN: 978-3-030-32489-6. DOI: 10.1007/978-3-030-32489-6_13. URL: https://doi.org/10.1007/978-3-030-32489-6_13 (cit. on p. 18).

[ISO25010] *ISO/IEC 25010:2023: Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model*. International Organization for Standardization, 2023. URL: https://www.iso.org/standard/78176.html (cit. on p. 20).

[JKM23] N. Janbi, I. Katib, R. Mehmood. "Distributed artificial intelligence: Taxonomy, review, framework, and reference architecture". In: *Intelligent Systems with Applications* 18 (2023), p. 200231. ISSN: 2667-3053. DOI: https://doi.org/10.1016/j.iswa.2023.200231. URL: https://www.sciencedirect.com/science/article/pii/S266730532300056X (cit. on pp. 22, 23, 27, 28).

[LOX21] G. A. Lewis, I. Ozkaya, X. Xu. "Software Architecture Challenges for ML Systems". In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2021, pp. 634–638. DOI: 10.1109/ICSME52107.2021.00071 (cit. on p. 15).

[Mac19] F. Machida. "N-Version Machine Learning Models for Safety Critical Systems". In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2019, pp. 48–51. DOI: 10.1109/DSN-W.2019.00017 (cit. on pp. 22, 23, 25, 26).

[MBF+22] S. Martínez-Fernández, J. Bogner, X. Franch, M. Oriol, J. Siebert, A. Trendowicz, A. M. Vollmer, S. Wagner. "Software Engineering for AI-Based Systems: A Survey". In: *ACM Trans. Softw. Eng. Methodol.* 31.2 (Apr. 2022). ISSN: 1049-331X. DOI: 10.1145/3487043. URL: https://doi.org/10.1145/3487043 (cit. on p. 15).

[MG22] B. M. A. Matsui, D. H. Goya. "MLOps: five steps to guide its effective implementation". In: *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*. CAIN '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 33–34. ISBN: 9781450392754. DOI: 10.1145/3522664.3528611. URL: https://doi.org/10.1145/3522664.3528611 (cit. on p. 15).

[NYO+19] S. Nalchigar, E. Yu, Y. Obeidi, S. Carbajales, J. Green, A. Chan. "Solution Patterns for Machine Learning". In: *Advanced Information Systems Engineering*. Ed. by P. Giorgini, B. Weber. Cham: Springer International Publishing, 2019, pp. 627–642. ISBN: 978-3-030-21290-2. DOI: https://doi.org/10.1007/978-3-030-21290-2_39 (cit. on pp. 22, 23, 26, 27).

[NZL+23] N. Nahar, H. Zhang, G. Lewis, S. Zhou, C. Kästner. "A Meta-Summary of Challenges in Building Products with ML Components – Collecting Experiences from 4758+ Practitioners". In: *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*. 2023, pp. 171–183. DOI: 10.1109/CAIN58948.2023.00034 (cit. on p. 15).

[RJCH23] S. Roh, K.-M. Jeong, H.-Y. Cho, E.-N. Huh. "An Efficient Microservices Architecture for MLOps". In: *2023 Fourteenth International Conference on Ubiquitous and Future Networks (ICUFN)*. 2023, pp. 652–654. DOI: 10.1109/ICUFN57995.2023.10201181 (cit. on pp. 22, 23, 28, 32).

[Ser19] A. C. Serban. "Designing Safety Critical Software Systems to Manage Inherent Uncertainty". In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 2019, pp. 246–249. DOI: 10.1109/ICSA-C.2019.00051 (cit. on pp. 22, 23).

[WUKG20] H. Washizaki, H. Uchida, F. Khomh, Y.-G. Guéhéneuc. "Machine learning architecture and design patterns". In: *IEEE Software* 8 (2020), p. 2020 (cit. on p. 15).

[Yok19] H. Yokoyama. "Machine Learning System Architectural Pattern for Improving Operational Stability". In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. 2019, pp. 267–274. DOI: 10.1109/ICSA-C.2019.00055 (cit. on pp. 22–25).

All links were last followed on November 6, 2024.

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted hard copies.

---

place, date, signature