SOFTWARE ENGINEERING PRACTICES IN DEVELOPING DEEP LEARNING MODELS: AN INDUSTRIAL CASE VALIDATION

Adnan Mahinić
amc22003@student.mdu.se

Examiner: Federico Ciccozzi
Mälardalen University, Västerås, Sweden

Supervisor: Antonio Cicchetti
Mälardalen University, Västerås, Sweden

Company supervisor: Nicolas Leberruyer,
Volvo Construction Equipment AB, Eskilstuna

October 1, 2023
Acknowledgements

I would like to express my deepest gratitude to my dedicated supervisors, Nicolas Leberruyer and Antonio Chiccheti, whose unwavering guidance, insightful feedback, and continuous support were instrumental in the completion of this master thesis. Your expertise and encouragement have enriched my understanding and shaped the quality of this work.

I am indebted to Volvo Construction Equipment for providing me with the invaluable resources and real-world insights that greatly contributed to the depth and relevance of my research.

To Malardalens University, I extend my appreciation for fostering an environment of academic excellence and growth, which has been pivotal to my development throughout this journey.

My heartfelt thanks go out to my beloved family for their unending encouragement, belief in my abilities, and for standing by me during both the triumphs and challenges of my education journey. Their belief in me has been my constant motivation, and I am truly blessed to have them by my side. This achievement would not have been possible without their sacrifices and belief in my dreams. Thank you for being my pillars of strength and for always standing by me.

Lastly, but certainly not least, I want to extend a special acknowledgment to my girlfriend, whose unwavering support, patience, and understanding have been my constant motivation. Your presence in my life has been an anchor of strength.

To all those who have played a part, no matter how small, in this endeavor, thank you for being an integral part of this achievement.
Abstract

The widespread of machine learning and deep learning in commercial and industrial settings has seen a dramatic up-rise. While the traditional software engineering techniques have overlap between machine learning model development, fundamental differences exist which affect both scientific disciplines. The current state-of-the-art argues that most challenges in software engineering of deep learning applications stem from poorly defined software requirements, tightly coupled architectures and hardware-induced development issues. However the majority of the current work on this topic stems from literature reviews and requires validation in an industrial context. The work aims to validate the findings of the academia through the development of the autoencoder model for gearbox fault detection. The model has been developed as a part of the ongoing campaign from Volvo Construction Equipment towards introducing AI-based solution in quality control and production. Findings of the work are mostly aligned with the current state-of-the-art, where poorly defined software requirements and hardware-induced issues have been experienced, but the tightly-coupled architecture did not characterize the final product. Along with the confirmation of the previous findings, the work presents a recommendation for practitioners of software engineering for deep learning models in the form of technological rule which addresses the hardware-induced issues of development through the contribution of a method for calculating the memory requirements of the model and batch during the training phase.

Keywords: Software engineering, deep learning, autoencoder, design science, technological rule, industrial case validation
# Table of Contents

1. Introduction ........................................... 1

2. Background ............................................ 3
   2.1. Software Engineering ................................. 3
   2.2. Software Development Lifecycle ...................... 3
   2.3. Deep Learning .................................. 4
   2.4. Autoencoders ........................................ 5
      2.4.1 Overview of the Architecture .................. 5
      2.4.2 Application in Fault-Detection .................. 6
   2.5. Machine Learning Development Lifecycle .......... 6
   2.6. Quality Attributes ................................ 7

3. Related Work ......................................... 8

4. Problem Formulation ................................. 10
   4.1. Research Questions ................................ 10
   4.2. Limitations ...................................... 10

5. Method ................................................. 11

6. Ethical and Societal Considerations .............. 14

7. Implementation .......................................... 15
   7.1. Requirements ...................................... 15
   7.2. Architectural Overview ............................. 16
   7.3. Development ...................................... 17

8. Comparison to State of the Art .................. 21

9. Technological Rule .................................. 21

10. Discussion ............................................ 23

11. Conclusions and future work ..................... 25

References ............................................... 27
List of Figures

1. AI vs. ML vs. DL [1] ................................................................. 4
5. System Architecture and Data Flow ........................................... 16
6. Preprocessing Pipeline class diagram ........................................... 17
7. Autoencoder Class Diagram ...................................................... 18
8. Encoder Memory Requirements .................................................. 19
9. Decoder Memory Requirements .................................................. 20
10. Total Model and Features Memory Requirements .......................... 20
1. Introduction

Machine learning (ML), and its sub-discipline deep learning (DL), has seen wide adoption in all forms of industry, and has achieved commercial use. [5] Machine learning has evolved as the preferred way for constructing usable software systems for computer vision, speech recognition, natural language processing, robot control, and other industrial use-cases. Machine learning skills can be introduced to a system in a variety of ways, including software systems that include ML components and ML frameworks, tools, and libraries that enable ML functionality. A common tendency has emerged: while designing and deploying machine learning systems is generally quick and inexpensive, maintaining them over time is challenging and costly due to technical debt. ML systems have all of the challenges associated with non-ML software systems, plus a collection of ML-specific issues.

Software practitioners are still struggling to operationalize and standardize ML-based system software development techniques. For the cost-effective development of high-quality and reliable ML systems, operationalization and standardization of software development techniques are required. There have been several attempts to identify the existing challenges in software engineering of deep learning models, and most of them were related to the requirements elicitation, quality attributes of the architecture and development stages. However, the majority of the studies currently available are of the systematic literature review type. The validity of these researches still need to be proved, as the findings from the academia might not translate that well to the real-world, and with the machine learning development being such a young branch of science, it is crucial to perform industrial case validations.

Autoencoders are neural networks that can learn meaningful features and representations from data, making them an attractive tool for simplifying the feature engineering process in machine learning projects. Furthermore, autoencoders can be utilized for data denoising, dimensionality reduction, generative modeling, and even pretraining deep learning neural networks. [6] More recently, the industry has adopted autoencoders in fault detection as a solution to the problem of increased complexity of process monitoring. [7]

Volvo Construction Equipment (VCE) in Eskilstuna produces gearboxes and axle components for wheel loaders, excavators and haulers. At the end of the production line, gearboxes go through a quality inspection cell, where an audio file of the gearbox sound is recorded. VCE wants to implement an AI based solution for gearbox fault detection, similarly to the previously mentioned notion of supplementing existing monitoring and testing methods being validated through machine learning solutions. Thus, this presented itself as an opportunity to examine the current state-of-the-art in software engineering when it comes to developing deep learning models and validate the findings of academia through an industrial case. Hence, the work seeks to contribute both to the fields of data science and software engineering, by devising an autoencoder based fault detection tool and implementing software engineering in the development of the tool.

However, while validation of academic findings was the primary goal of the work, far too often new findings stem from the industrial settings. Guided through the design science research paradigm, solutions to real-world problems within the software domain are created, evaluated, and provided, in turn improving software engineering practices, and contributing to the development of innovative artifacts that benefit both researchers and practitioners. The work achieves this by providing a technological rule regarding the development of the testing stage of a deep learning model, thus benefiting both the practitioners of software engineering and data science. The technological rule proposed in this work advises a use of a method that calculates the memory requirements of a deep learning model.

The thesis is sectioned into chapters as follows: Section 2. presents a theoretical background relevant to both software engineering and machine learning, necessary to understand the work conducted. In Section 3. state-of-the-art regarding software practices in deep learning is given. The problem formulation and research questions, along with the limitations of this work are found in section 4. The design science paradigm and the setup of the research is explained in section 5. Section 7. describes the implementation of the autoencoder fault detection tool through the software engineering process, while section 8 compares it to the state-of-the-art. The novelties of the work are presented in the form of technological rule in section 9. The work identified difficult requirements elicitation, high-coupling of the components and hardware limitations as the primary
SE challenges in developing DL models. Excluding high-coupling of the components, the work validated these challenges in an industrial environment, as well as the use of open-source libraries and modules as the characteristics of software engineering of deep learning projects. This, along with the practicality of the technological rule, is discussed in Section 10, while Section 11 covers the summarization of findings and prospects for future endeavors.
2. Background

In this section the fundamental concepts and knowledge required for through understanding of the research are presented. The initial focus is on presenting the standard practices for traditional software development as the work is directed at software engineering of deep learning projects. Naturally, the second part will contain definitions and applications of deep learning, and more specifically autoencoders, as they are at the core of the work. Finally, machine learning development lifecycle will be explained in order to allow parallels to be draw to standard software engineering. Finally, as non-functional requirements of the project are encapsulated using quality attributes, they will also be addressed and defined.

2.1. Software Engineering

Software engineering is defined as a discipline encompassing methodologies, and practices for designing, developing, testing, deploying, and maintaining high-quality software systems. [8] A direct consequence of the rapid evolution of technology has caused an omnipresent integration of software into day-to-day aspects of modern line, such as personal devices, industry or critical infrastructure. Software engineering’s primary goal is to improve the reliability, efficiency, and scalability of software systems. This goal is achieved through careful adherence to foundational concepts of software development process, which includes:

- **Requirements Engineering**: Gathering, analyzing, and documenting the needs and specifications of the software system to be developed.
- **Design and Architecture**: Creating a blueprint for the software system’s structure, components, and interactions to ensure scalability, maintainability, and flexibility.
- **Coding and Implementation**: Transforming design into executable code through programming languages, adhering to coding standards and best practices.
- **Testing and Quality Assurance**: Systematically verifying and validating the software to identify and rectify defects, ensuring its functionality, security, and reliability.
- **Deployment and Maintenance**: Releasing the software to users and continuously maintaining and updating it to address issues, incorporate new features, and adapt to changing requirements.
- **Project Management**: Applying project management techniques to plan, schedule, allocate resources, and monitor progress throughout the software development life cycle.

2.2. Software Development Lifecycle

The Software Development Life Cycle (SDLC) is a foundation or framework for organizing, planning, and carrying out tasks associated with the development of an information system. [9] The majority of software development is done using various SDLC models. These models include the waterfall model, the spiral model, the V model, the rapid application development model, the agile software development model, and others. All of these models are primarily configured with requirement specifications, design, implementation, testing, deployment, and maintenance in mind. These concepts are applied and followed in several SDLC models, each with its own operational mode. Every stage of the SDLC model is based on different principles and concepts.

When selecting a software development technique for a deep learning application, one must take into account the project’s complexity, team size, flexibility, and client collaboration. An iterative and flexible methodology, such as Agile, can be advantageous for deep learning projects, allowing for constant revisions to models and requirements. However, including DevOps aspects can ensure that DL models are integrated and deployed smoothly. A more formal approach, such as as Waterfall, may be sufficient if the project has well-defined needs and a stable environment. Finally, it is critical to adjust the technique to the unique constraints of combining DL with software development while emphasizing adaptability and collaboration. [9]
2.3. Deep Learning

Referred to as a branch of machine learning (ML) (see Figure 1) [10], deep learning (DL) aims to mimic the patterns of information processing as close as possible to the human brain. What separates deep learning from the rest of the subbranches of machine learning is the ability to map a given input to a concrete label by leveraging large amounts of data without any human-designed rules. The mechanism which enables this is based upon layers of algorithms making up an artificial neural network (ANN). Each of the layers in ANNs is responsible for providing a particular interpretation of the data. [11]

![Figure 1: AI vs. ML vs. DL](image)

Traditionally, machine learning requires the steps of pre-processing, feature extraction, learning and classification steps in order to achieve the classification task. In particular, manual feature selection nature of ML is often considered as being biased [12] leading to data leakage. On the contrary, deep learning possesses the ability to learn important feature sets automatically, removing the necessity for feature extraction. Further more, often the other steps such as pre-processing and feature extraction can be omitted, allowing for DL model to become "single-fit" and completely black-box in nature. [13][11] The applications of deep learning can in some situations perform better than their human counterpart [14], and that’s why it is applied in:

- Instances where there is a lack of human experts
- Learning of underlying patterns exceeding human perception (speech recognition, healthcare, sentiment analysis)
- Time-variant environments (price prediction, stock exchange)
- Instances where there is a demand for high-adaptability (personalized content)
- Complex and large data inference (fault detection, system reliability)
2.4. Autoencoders

An autoencoder is a type of a deep learning algorithm whose primary purpose is learning of an "informative" data representation that can be used for different applications. This is achieved by learning how to reconstruct a set of input observations well enough. [15] Autoencoders have found their applications in image compression[16], as autoencoders represent images in compressed format by retaining the most important features. Similarly, autoencoders have been used for text summarization.[17]. However, more recently, autoencoders started appearing as proposed solutions to a fault detection problem. [18] This work also leverages the fault detection nature of autoencoders and provides an implementation for an industrial case. Thus, this portion of the paper will be dedicated to explaining the mechanism behind autoencoders, and section 2.4.2 will give a brief overview on how autoencoders are used for fault detection purposes.

2.4.1 Overview of the Architecture

Autoencoders can be visualized as in Figure 2, which represents a typical autoencoder architecture. Main components of the autoencoder are encoder, latent representation and the decoder. Encoder and decoder are functions, while the latent representation is a tensor of real numbers, containing the most important features of the input. The main goal of an autoencoder is to reconstruct the original input as best as possible.

![Figure 2: Illustration of autoencoder model architecture][2]

Encoder and decoder functions are implemented using neural networks. Expressed mathematically, encoder is written as a function \( g \) depending on some parameters \( x_i \):

\[
z_i = g(x_i)
\]  

(1)

Where \( z_i \in \mathbb{R}^q \), the latent feature representation symbolizes the output of the encoder block, given an input \( x_i \). Thus formally, we have \( g: \mathbb{R}^n \rightarrow \mathbb{R}^q \). Similarly, the decoder and its output \( x'_i \) can be written as a generic function \( f \) of latent features \( z_i \) as:

\[
x'_i = f(z_i) = f(g(x_i))
\]  

(2)

where \( x'_i \in \mathbb{R}^n \). An autoencoder is considered trained when the functions \( g(\cdot) \) and \( f(\cdot) \) which satisfy the following condition:

\[
\arg \min_{f,g} < [\Delta(x_i, f(g(x_i)))] >
\]  

(3)

where \( \Delta \) is a measure of a difference between the input and the output of the autencoder i.e. the loss function, while \( < \cdot > \) indicates the average overall observations. In order to avoid creating an
identity function between $f$ and $g$, a so called "bottleneck" is added, so that the dimension of the latent features is reduced.

### 2.4.2 Application in Fault-Detection

As mentioned in the previous section, autoencoders are often used for fault detection. The idea behind this is that autoencoders are only good at reproducing inputs similar to the ones it was trained on. Let’s imagine a scenario where we train an autoencoder on a set of handwritten digits. If a handwritten digit is provided as the input for the autoencoder, it will reproduce it with minimal reconstruction error, since it is already "familiar" with such type of input. However, if an autoencoder receives a triangle for input, it won’t be able to reconstruct it and that will be evident with a high reconstruction error. Thus, autoencoders become a suitable tool for outlier detection.

More generally speaking, implementing an autoencoder for fault detection is performed in these discrete steps:

1. Provide the entire dataset with minimal outlier to autoencoder for training
2. Calculate the reconstruction error for the portion of the dataset containing the outliers
3. Sort the observations by reconstruction error
4. Classify the observations with high reconstruction error as outliers

Important thing to note is that autoencoders cannot provide a clear cut answer to the outlier problem, but rather they make a strong suggestion that an observation might be an outlier. In order to be certain of the classification done by the autoencoder, the problem and the results need to be analyzed by someone who is an expert on the data and the problem itself.

### 2.5. Machine Learning Development Lifecycle

The lifecycle of the machine learning project is somewhat similar to that of a traditional software engineering. A research conducted by Microsoft [3] defines the machine learning workflow in 9 stages as in Figure 3:

![Figure 3: Machine Learning Workflow as depicted in [3]](image)

In stage 1, model requirements, the designers make key decisions on which features are useful for the product and what model is the most appropriate for the given problem. Stage 2, data collection aims to gather all the data available (internal and external) used to train the model. This data is then processed and all inaccuracies are removed during stage 3, the data cleaning. In the data labeling step, the engineers assign truth labels or classes to each record, as labels are required to implement supervised learning. Even in unsupervised methods, this step is somewhat required as you need to understand what data you possess. Following the data preparation step, in stage 5, feature engineering, the data scientists aim to extract and select features that contribute to the model the most. Stage 6, model training aims to train the chosen models on the collected data, based on the labels provided, after which in model evaluation, stage 7, the engineers evaluate the model based on pre-defined metrics or in critical domains, extensive human evaluation. After the model has been evaluated and approved, stage 8 follows, as the model is deployed on target devices. During the final stage 9, the model is continuously monitored during execution and any possible errors are reported.
Even though it may seem linear, a machine learning project often contain lots of feedback loops and there is back-and-forth between stages, sometimes to the point of requiring to collect new data during the model evaluation. This does run in parallel with the Agile way of developing software, however ML workflow requires a considerably higher amount of experimentation to that of traditional software engineering. Thus, there exists a gap between the two that needs to be bridged, and that is one of the contributions of this work.

2.6. Quality Attributes

In software engineering, and particularly when discussing software architectures, quality attributes represent a *measurable or testable propery of a system that is used to indicate how well the system satisfies the needs of its stakeholders* [19]. Quality attributes (QAs) are used to measure the "goodness" of software along a dimension that is of interest to a stakeholder.

Quality can be defined in a variety of ways. Quality qualities and features are critical in the overall design of software systems. The product’s quality helps to better user requirements and satisfaction, clearer software design, and, ultimately, higher end product quality.

Quality attributes are diverse and often interrelated, influencing each other and sometimes presenting trade-offs. When expressing non-functional requirements (quality attributes), that is requirements that do not elicit what the software should do, they need to be measurable and testable. Moreover, quality attributes are orthogonal to functional requirements, meaning that the choice of functionality does not impact the level of quality of such functionality. [19]

Since the aim of the work is to measure certain quality aspects from the software side of the implemented fault detection tool, the following are the definitions for the quality attributes that will be analyzed in the work, as defined in [19]:

1. **Performance** - systems ability to meet the timing requirements
2. **Modifiability** - the ease which changes can be made to the system
3. **Portability** - the ease which software that was built to run on one platform can be changed to run on a different platform
3. Related Work

This section focuses on the presentation of previously conducted research on software engineering (SE) practices for deep learning (DL) models. The goal is to provide a general idea what the academia has presented so far as current problems and how this paper fits into the existing realm of research.

When comparing the development of traditional SE systems to that of machine learning (ML) ones, there exist fundamental differences. In a work by Arpteg et. al [20], the authors have performed interpretative research on 7 industrial DL projects in order to identify the main challenges and best practices for building DL systems. Unlike this thesis project which focuses on fault detection, the projects described in the work by Arpteg et. al focus on machine learning models for predicting weather, user retention, oil and gas recovery potential. Nevertheless, the key takeaways from the software development side of things are transferable to a degree on this project as well. The authors have managed to identify three categories: development challenges, challenges related to production, as well as organizational challenges. Since the scope of this project hasn’t reached the production phase, only the development challenges will be presented.

Among the development challenges, the authors have cited experiment management, limited transparency due to DL-specific abstractions, troubleshooting issues caused by complicated frameworks, resource limitations, especially for single machine solutions, and testing. This work has addressed the resource limitations, as the autoencoder framework has been tailored to estimate the required computing resources and time for training the instance of the model on the specific dataset before the training has been initiated. When it comes to experimentation, authors mention that reproducible results can only achieved if the hardware, platform, source code, configuration, training data and model state are version controlled.

However, the work described in the previous paragraph has overlooked one key component of software development, and that is requirements gathering. A careful consideration for ML software requirements has been given by Wan et. al [21] in their research, where they aimed to uncover how machine learning changes software development practices. Their interviewees noted that ML software systems tend to source the requirements from the data that’s being processed and the stakeholders understanding of the data, as well as quantitative measures such as accuracy, precision and recall. Additionally, authors mention that the high-level architecture for ML systems is mostly fixed, following the collection, cleaning, feature engineering, execution and deployment template, something which has also been exhibited on this project. A direct consequence of that is a system of highly coupled components, discouraging concurrent development of software components, another trait of this project.

Another work which outlines software practices in development of DL is that of Zhang et. al [22]. Just like the previously described work, the authors note the difficulty in eliciting requirements, citing that they are sourced from the overall understanding of the data by the stakeholders. Participants of the study have mentioned that as the datasets grow larger, a higher focus of the development is pushed toward concurrent processing, when coupled with the requirement of handling requests in parallel, shifts away the focus from addressing the functional requirements. The authors do note that popular DL frameworks enable for easier prototyping, which is in line with this work, as it relies on the popular TensorFlow library. Additionally, one of the strong findings of the research is the notion that ensuring the quality and correctness of DL applications is mostly done through model evaluation against benchmark, and there is a low emphasis on traditional software testing methods, such as unit testing or code review.

Similar to the findings of Wan et. al [21], Serban et Visser [23] share common views on software requirements for ML project, particularly in the conclusion that the components lack functional requirements. What Serban et Visser expand upon is the design phase of ML projects. They note that it is essential to lower the uncertainty of ML components via the use of n-versioning and monitoring of uncertainty metrics. The participants of the research concluded that the most important decision drivers of their project were hardware and scalability, which was exhibited on this project, as hardware limitations impose heavy compromises when it comes to parameters of the model. The scope of the research extended into quality attributes of the respective software architecures. Interestingly, most of the participants gave a low importance to privacy and security of the project, despite several known breaches of ML components in practice. [24]
In a survey involving more than 80 practitioners of machine learning, Saidur et. al [25] provided 17 findings about common practices in developing ML applications, both from the AI and software engineering side. Particularly related to this work, a great majority of the participants (93.18%) answered that they used existing libraries and frameworks for their ML model implementations. Moreover, when it comes to testing, just like in [22] and on the notes of [21] the focus is shifted towards accuracy of the model and the performance, rather then traditional software testing methods. The participants cite the black-box nature of ML and divergence of data as main challenges of testing. When asked on what are the primarily monitored factors for deployed models, practitioners cited performance, business parameters and overall user acceptance.

Alshangiti et. al [26] conducted a study on StackOverflow Posts with the aim to uncover the most common pitfalls when designing ML applications. With regards to software engineering problems, environment setup and model deployment phase is considered the most challenging, as 50% of the posts lacked an answer on those topics. The reliance on existing libraries and frameworks for development of AI and ML models has its drawbacks however. Authors have inferred that proper documentation is lacking for most of AI libraries, as considerable amount of questions on StackOverflow were centered around the appropriate use of methods or difference between methods in a class. Among the contributions of this project is a comprehensive documentation of all modules, moving towards bridging of the gaps between AI projects and software engineering.

In a 2019 Microsoft case study conducted by Amershi et. al [3], a nine-step development workflow for AI-applications was analyzed using the data gathered from Microsoft teams. They too outlined the differing characteristics of ML modules compared to traditional software engineering ones, citing complex component entanglement affecting stages such as specification, debugging and testing. In terms of customization and reusability of the ML modules developed in the projects, the authors note in their observations that for similar domains, applying the same model can take a lot of modifications, taking as much time as developing the original. Just like previously reviewed work, component entanglement is cited as one of the challenges when attempting to achieve modularity within ML/AI applications.

When reflecting on the related work that was reviewed, it can be inferred that the most of the works are either statistical literature reviews or data mining methods, whose findings often need to be validated. Thus, a gap in the research has been identified, and this work will serve as an industrial case validation of the common software engineering challenges and practices in developing deep learning models.
4. Problem Formulation

This section is dedicated to identifying the problem and the consequential research questions that drive this work. In the final parts of the chapter, limitations of the research shall be outlined.

4.1. Research Questions

This research aims to investigate various aspects of software engineering in the context of developing deep learning models. The initial focus is on identifying the current challenges in this domain, shedding light on the difficulties that arise during the software development process for such complex models. The work seeks to explore how findings from recent research in deep learning software engineering translate to practical applications in industries, investigating whether characteristics highlighted in academic research are evident in real-world industrial settings. Finally, the work delves into potential strategies for addressing the challenges identified in the state-of-the-art for software development in deep learning, aiming to propose solutions or mitigation techniques that could enhance the development of deep learning models by improving their software engineering practices.

Formally, the thesis aims to answer these research questions:

- What are the existing software engineering challenges in developing deep learning models?
- Which characteristics from contemporary research on software engineering of deep learning can be observed in a real-world industrial scenario?
- How can some of the software engineering challenges in developing deep learning models be mitigated?

4.2. Limitations

This section discusses the limitations of the work, which are important to understand how the thesis could be improved upon. They are as follows:

1. Partial completion of the software engineering development lifecycle
   The primary limitation of this study is the incomplete software engineering development lifecycle of the autoencoder model. The testing, deployment, and monitoring stages are an essential part of the SDLC, and would introduce new challenges. Furthermore, they would impact the overall design and architecture, as these stages would need to be incorporated in the planning phases. So in order to fully compare the work to the state-of-the-art in software engineering, future research would need to incorporate them.

2. Lack of strong empirical validation for the technological rule
   Even though it is reasoned that the technological rule has been empirically validated through the application in industrial environment, the design science pattern emphasises that the validation comes through other forms of research methodology such as experiment, case study, or another industrial case validation. Thus, the contribution of the technological rule is yet to be proved.
5. Method

This section contains the description of methodologies which were used to answer the research questions outlined in the section 4. The first research question was addressed via a literature review. As it comes to RQ2, a comparison was carried out between the state-of-the-art and the implementation in an industrial context. Lastly, research question 3 was answered via the design science paradigm and technological rules were presented that target resolving issues currently present in the software engineering of deep learning models.

To address the first research question, it is essential to investigate the state-of-the-art in deep learning model software engineering. This involves examining databases of the popular digital libraries such as IEEE Xplore, ScienceDirect and Springer, and conducting a literature review to identify the key challenges and problems in designing deep learning solutions. Additionally, the research aims to determine the design principles that facilitate fault detection accuracy, robustness, scalability, and adaptability in real-world software systems.

The second research question focuses on validating the findings of the first research question in an industrial context. A comparison will be carried out between the state-of-the-art and an implementation of the software engineering process for AI based fault detection tool, and similarities and differences will be pointed out.

The third research question addresses the design of deep learning models and how it could incorporate software engineering needs. To accomplish this, the research aims to explore various strategies for tailoring deep learning tools to meet different software engineering scenarios and in turn improve the current state of software development. [27] Given the state of empirical software engineering [28] and evidence-based software engineering [29], the main focus on the research is now primarily oriented towards empirically informed understanding of practice and solution proposals.

One such paradigm that can be used as a frame for presenting and analyzing software engineering research is design science. The main goal of design sciences is to understand and improve upon human-made designs in an area of practice. Given the nature of software engineering, the majority of studied phenomena represent designed artifacts, thus allowing for the research to be framed as design science. [27]

The design science research paradigm in software engineering is an approach that focuses on creating and evaluating innovative artifacts to address specific problems in the field. It is a problem-solving-oriented research methodology that aims to produce knowledge through the construction and evaluation of novel artifacts rather than purely theoretical analysis.

In the context of software engineering, the design science research paradigm emphasizes the creation of software artifacts, frameworks, methodologies, techniques, or models that can be practically applied to solve real-world problems. These artifacts are designed to improve the development, management, or usage of software systems and are evaluated based on their effectiveness, efficiency, usability, and impact.

One can think of design science as spanning across two major dimensions: the problem-solution and theory-practice dimensions. Observing the figure 4, the two quadrants in the bottom represent the practical contribution of the research, i.e. both the problem and its solutions. On the other hand, the theoretical contribution, visualized by the two top quadrants, are in line with the technological rules and the corresponding constructs. [27]
The key characteristics of the design science research paradigm in software engineering include:

- **Problem conceptualization**: The research is driven by identified problems or challenges in software engineering practice or theory. The activity primarily concerns with describing the problem.

- **Solution design**: activity of mapping the problem to a general solution

- **Abstraction**: identification of key design decisions under the given scope of the solution’s validity.

- **Instantiation**: implementing the solution within a context

- **Empirical validation**: The artifacts are evaluated through empirical studies, experiments, case studies, or other forms of evaluation to determine their effectiveness and impact. The research involves an iterative process of refinement and improvement based on evaluation results.

Design science is closely related with the term technological rule, as it serves as a mean to transfer knowledge between contexts. A technological rule itself can be defined as a mapping between the instances of the problem and the solution. The research outcome of design science needs to be framed in terms of effects of interventions, and a technological rule achieves exactly that. [27] One of the ways of expressing technological rules is as follows:

**To achieve Effect in Context apply Intervention**

A technological rule contains the design knowledge, which helps the software engineers in designing customized solutions for their particular problems. Given a technological rule, there needs to be a generalization of software engineering problems tailored to stakeholder’s desired effect, so that it can be applied as a potential intervention based on the specifics of the context. The intervention itself is formulated in various way. These may include a referral for and against the use of a tool, application of a practice, technique, framework or a set of guidelines. [27]

However, the validity of technological rules can be questioned depending on the level of abstraction at which they are expressed. Given a very high abstraction level, technological rules may
seem trivial or easy to disprove, while for a low abstraction level, technological rules start losing the relevance for most software engineers.

With regards to theoretical knowledge, the design science paradigm also produces the constructs on which the technological rules are built. More specifically, this refers to the conceptualization of the problem and the solution domains. An example of a construct can be a taxonomy used for classification of a set of technological rules. [27]

With regards to the validation of a technological rule, it must be first instantiated, so that it’s validity could be reasoned about. With the To achieve Effect in Context apply Intervention way of expressing technological rules, we can capture the validation activities. The intervention represents the object of the validation study, while the context is the place where research is conducted. Finally, the expected effect defines the validation criteria. [27]

As described by Runeson et. al in [30], explanatory sciences rely on statistical generalizations, while the design science paradigm gets its basis from the theoretical/analytical generalizations. The scope of validity for a technological rule is extended by application of the intervention to new contexts, or by comparing two contexts and reasoning about the similarity and applicability of the intervention in the other context. This constitutes case-based generalization [27]

The design science research methodology is highly suitable for this research on building a deep-learning-based AI fault detection tool for gearboxes. This methodology focuses on creating innovative artifacts to solve specific problems, which aligns perfectly with the objective of improving the existing software engineering processes for deep learning. By following the design science research paradigm, key components of the deep learning framework can systematically be designed, implemented, and evaluated. The assessment of these RQs will lead to a contribution to the development of knowledge and best practices in designing deep learning models for fault detection, providing guidance to future researchers and practitioners.

Secondly, the design science research methodology encourages iterative and practical research cycles, which is well-suited to the project. As the project progresses through data preprocessing, model implementation, and UI development, the artifacts can be iteratively refined and improved based on evaluation feedback. This iterative process aligns with the principles of design science, where knowledge is gained through the construction and evaluation of practical solutions. By conducting evaluations using appropriate metrics and benchmarks, the effectiveness and efficiency of current techniques for developing deep learning models will be assessed and informed improvements will be made.

By addressing these research questions, this study seeks to contribute to the development of a deep learning tools for fault detection in software engineering that is based on sound design principles identified in state-of-the-art and validated in an industrial context, and innovative technological rules which can cater to different software engineering contexts.
6. Ethical and Societal Considerations

Despite the fact that this study is being conducted in an industrial setting, it does not involve any interactions or interventions from the real-world because it does not use live data, but rather a recording of data collected from January 2023 to April 2023. Moreover, the study is organized in such a way that no personal, confidential, or sensitive information is used. As a result, it is critical to stress that this thesis work raises no ethical or societal problems.
7. Implementation

This section contains the extensive description of the system architecture and design, as well as
the breakdown of the autoencoder based fault detection. While it does describe a specific problem
solution, this implementation can be viewed as an instance of a more general problem of developing
a fault-detection DL model based on waveforms. For example, a work conducted by Ahmad et. al
[31] implemented an autoencoder-based monitoring and anomaly detection for rotating machines
based on vibrations. Just as sound, vibrations are mechanical waves that can be represented by
their respective spectrograms. The implementation follows the traditional software development
life cycle, and showcases how a deep learning software project is conducted.

7.1. Requirements

As mentioned before, this project has been conducted as part of ongoing campaign from Volvo Con-
struction Equipment (VCE) to incorporate machine learning/artificial intelligence based software
solutions into the production. VCE factory at Eskilstuna manufactures gearboxes and transmission
axles components. At the end of the production line, the gearboxes go through a quality inspection
test where an audio file is recorded. VCE understands that AI algorithms can analyze data from
various sensors and sources to provide insights into the health of gearboxes. While manual data
analysis methods exists, this data-driven approach can lead to better decision-making and further
validate existing fault detection techniques. Thus, the goal of the project is defined as follows:

**Develop an anomaly detection model for the wheel loader gearbox based on audio
recordings**

The first month of the project was spent on understanding the existing methods and the target
system, along with eliciting requirements. The data acquisition (DAQ) system itself is a rig that
collects sound from two onboard microphones and the rotational speed of the ingoing and outgoing
shaft of the gearbox. The data is recorded during a 90 second test using a LabVIEW DAQ
application and it is stored in a binary format on a local industrial computer. The application
itself is a proprietary one, meaning that there was no existing documentation or source code to
gain further understanding or modify the DAQ process. Furthermore, while the data is stored in
a binary format and used by existing analysis software, the only way to convert a DAQ file to a
human-readable is manually via another LabVIEW proprietary application which converts a single
file to a Universal File Format 58 (.uff).

After the UFF file was obtained, data analysis was performed on it using the pyuff module in
Python. Based on the analysis, it was concluded that the Universal File Format data set is used
import and export data from a transducer located at a certain node and pointed in a specific
direction. It contains metadata about the acquisition, but the most important fields were channels
0, 1, 2 and 3 which contain the raw left audio microphone recording, right audio microphone
recording, ingoing and outgoing speed tacho signals, respectively. The file itself is 300 MB in size
due to high sampling frequency (50000 Hz), and unfortunately files are not stored longer than 7
days on disk, as the infrastructure is a bit dated and the memory would fill up quickly.

Given what was outlined previously, and following further discussion with the stakeholders,
these are the requirements that were set upon the project:

- The application shall implement deep learning algorithms for fault detection
- The DL model shall use the data that is acquired by the test rig
- The application shall preprocess raw sensor data by filtering noise, removing outliers, and
  performing data normalization
- The system shall extract relevant features from preprocessed data to represent the gearbox’s
  operational state
- The application shall provide visual diagnostic information about detected faults
- The user interface shall be user-friendly and intuitive, allowing operators and maintenance
  personnel to interact with the application easily.
7.2. Architectural Overview

Following the requirements elicitation stage, the project was moved to the architectural design phase. The architecture of the system is depicted in figure 5, along with the data flow.

Figure 5: System Architecture and Data Flow

The only constrain which was imposed on the system architecture from a data point-of-view was that the files are stored in the binary format on the local industrial PC, as mentioned in section 7.1. Thus, a decision was made to create an Extraction, Transform and Load (ETL) pipeline which consists of a batch-processor built on top of the existing UFF file converter in LabView and a loader which sends the data to Azure Blob Storage. The choice of Azure Blob Storage and Azure in general as the platform was intentional, as it facilitates fast development and abstracts the hardware from the development environment. Abstracting the hardware allows for parallel development of the model and training, and increased computing resources. Following the ETL the data is preprocessed as follows:

1. **Loader** loads the UFF58 files one by one to the processing node in Azure
2. Audio signals from both channels are extracted from the UFF file into arrays.
3. The arrays are padded from the left if the signal is shorter than 90 seconds or cut if the signal is longer than 90 seconds by the **padder**
4. **Spectrogram extractor** extracts the logarithm short-time Fourier transform amplitude (spectrogram) and stores it in a two-dimensional array, where values are a function of frequency and time.

5. **Normalizer** is responsible for taking the spectrogram and normalizing the values between 0 and 1 using MinMax normalization, as DL models produce better results with normalized data.

6. After the data is normalized, **saver** uploads the data back to Azure Blob Storage, saving the min-max values and the normalized array.

Once all the files have been preprocessed, custom **batch generator** loads files in batches of two and they are fed to the **autoencoder** model. The decision to load the files in batches instead of all at once was made due to the hardware constraints, as each preprocessed spectrogram contains more than 300 MB of data. The autoencoder is trained on the entire dataset, and subsequently the parameters and weights are stored in Azure Blob for further reuse.

### 7.3. Development

The preprocessing pipeline and the autoencoder model, along with the user interface, were developed in Python, in order to facilitate portability and leverage the existing numerical and deep learning libraries such as Numpy and Keras. Object-oriented approach was chosen as the programming paradigm, as it allows for increased readability and decoupling of components. The project was carried out between a pair of remote developers and the choice was made to use the virtual environment (venv) within Python in order to ensure consistency of the dev environment and packages. Additionally, the list of packages generated by the virtual environment enabled an easier migration to the cloud from the local development, while aiding in uniformity between the environments. The initial single-file processing pipeline and parts of the preprocessing pipeline were developed on local machine, while the autoencoder model was developed and tested on Azure Databricks.

---

Figure 6: Preprocessing Pipeline class diagram
The Preprocessing Pipeline and the including components, which implement the Batch Preprocessor from figure 5, are shown in figure 6. Important feature of the preprocessing pipeline is that all of its components are highly modifiable in order to support different implementations. For example, during the initial stages of the project the LogSpectrogramExtractor was developed using the Librosa package, however, since the package is tied to specific versions of Python, a design choice was made to implement the class via the Scipy module, but the same interface was retained. Normalizer class supports arbitrary min-max normalization. The preprocessing pipeline is loosely coupled, since the only information known between the components is that all of them expect numpy array/ndarray as input.

The autoencoder is depicted in the class diagram in figure 7. The autoencoder is implemented in a modular way which follows the architecture illustrated in figure 2, in order to facilitate easier modifiability, readability and debugging. Both the encoder and decoder parts of the autoencoder model are built with layer-based granularity. The autoencoder requires the input data dimensions, number of filters, kernels and strides in each layer and the build method will handle the connection of each of the layers.

```
Autoencoder
- input_shape : n-tuple of int
- conv1 filters : n-tuple of int
- conv1 kernels : n-tuple of int
- conv1 strides : n-tuple of int
- latent_space_dim : int
- pooling : bool
- interpolation : str
- encoder : Keras.Model
- decoder : Keras.Model
- model : Keras.Model
```

```
BatchGenerator(Keras.utils.Sequence)
- file_names : list of string
- batch_size : int
```

Figure 7: Autoencoder Class Diagram

The autoencoder supports all of the standard functionalities of a deep learning model, such as compilation, training, prediction (reconstruct method) and saving to and loading from disk. The training method in most autoencoder implementations requires the entire dataset to be loaded in memory and the training is performed on the entirety of the dataset at once. However, for this implementation, such approach was not possible due to large record size (8192x2048 matrix of floats). Thus, the choice was made implement a BatchGenerator class which inherits the base object keras.utils.Sequence for fitting to a sequence of data. [32]. This allows for the dataset to be
loaded in batches instead of at once, seemingly avoiding all memory problems.

However, the development of the training methods proved to be the most challenging part of the project. The previously mentioned large record size was due to 50000 Hz sampling frequency and the 90 second length of the audio recordings. Since there was no clear indication at which frequencies the noise in the gearboxes occurred, further downsampling was not possible, as the 50000 Hz sampling frequency is just above the Nyquist frequency[33] which covers the human-hearing spectrum. Loading individual files was not the problem, however, the autoencoder weights needed to be stored in memory as well during the training. As the record propagates through the encoder, it is downsampled in powers of 2 using the max-pooling method which preserves the most important features, but convolution filters add a third dimension to the data. The biggest factor in memory consumption from the autoencoder, however, are the dense layers which bridge the final convolution layer of the encoder and the bottleneck, and the bottleneck and decoder, respectively, as each data entry in the convolution layer is connected to every entry in the bottleneck array.

Thus, rapid prototyping of the autoencoder was made difficult as a slight change in the size of a convolution filter or downsampling in a layer would have enormous consequences on memory consumption, in some cases crashing due to memory exceptions during instantiation. Further compounding, it couldn’t be known if a configuration of the autoencoder was going to successfully train even if the model is instantiated, because there was no way to know if the configuration will leave enough memory for the batch to be loaded and other process variables to be stored. In order to resolve this problem, one of the contributions of this work is a method for calculating the memory requirement to store an autoencoder instance with arbitrary number of layers and the features of a batch. To serve as an example, an autoencoder model has been instantiated with the following parameters:

```
input_shape = (2048, 8192, 1)
conv_filters = (512, 128, 32, 16)
conv_strides = (4, 4, 4, 2)
conv_kernels = (3, 3, 3, 3)
latent_space_dim = 1024
```

The following is the output of the `get_model_memory_usage` for an instance of the autoencoder model.

![Figure 8: Encoder Memory Requirements](image-url)
The method displays first the memory usage for each of the layers of the encoder, and finally the total memory required for the component. Then, similar is done for each of the layers of the decoder. In the final summary, the total memory for the features is calculated as $(\text{encoder memory} + \text{decoder memory}) \times \text{batch size}$, since the model needs to store both files propagated through the weights in order to readjust each iteration. The parameters are in reference to the actual values for the weights of the autoencoder model in each layer. Without this method, instantiating the autoencoder would require a lot of guessing and it would severely prolong the development time of the autoencoder. Following this, due to the ability to rapidly prototype autoencoder instances, a satisfactory accuracy of the model was achieved and the autoencoder development, training and evaluation was finished. Lastly, an UI was developed using the `tkinter` module in Python in order to run the demonstration of the model, however, due to time constraints on the project, the final requirement was not fulfilled, which allows for customization of the options for visualization and display of data.
8. Comparison to State of the Art

As mentioned in section 3, most of the works done on the topic of software engineering in machine learning have been interpretative studies or systematic literature reviews, and the goal of this work is to put those findings into a context of an industrial setting. Thus, in this section, the implementation will be compared to what was described in section 3.

Regarding the requirements gathering and elicitation stage, primary focus of the stakeholders was to achieve a high-accuracy of the model on the existing dataset. During the later stages of the development however, the stakeholders realized that additional requirements should be imposed on the software side of things, such as the deployment platform and method. Thus, it can be concluded that the requirements have been sourced mostly from the data and the stakeholders’ understanding of the data, which is in line with the findings of Zhang et. al [22] and Wan et. al [21].

Wan et. al [21] concluded that high-level architecture of ML-projects is fixed and often leads to highly coupled components, as did Amershi et. al [3]. While it can be agreed upon that the high-level architecture does follow the traditional extract-preprocess-train architecture, one of the disagreements with the state-of-the-art in this case is that the components are loosely coupled, allowing for high modifiability. Additionally, the developed autoencoder model can be tailored to any fault-detection use-case which relies on spectral analysis, such as vibrations and radio signals, with minimum modifications, opposing the opinion of Amershi et. al on reusability.

The development part of the implementation has the biggest overlap between the state-of-the-art and this work. Aligned with the findings of Zhang et. al [22], Saidur et. al [25] and Alshangiti et. al [26] this project too relies on the open-source library Tensorflow as the basis of the model. Arpteg et. al [20] noted that the reproducibility of the results depended on the consistency of the platform, hardware, source code and training data. This has been achieved by keeping the hardware (Azure Databricks), platform (Python and Tensorflow), training data and model state tracked along the iterations. The only slight deviation from state-of-the-art is with the findings of Zhang et. al related to parallel processing. Since the application did not suffer from performance issues, there was no focus put towards concurrency, allowing the functional aspects to be in the forefront of development concerns.

Heavy restrictions on the development caused by hardware constraints were experienced on this project as in [23] and [20]. Moreover, this issue was encountered late in the development and the method for model memory usage was developed out of necessity, otherwise the project would have taken a lot longer than it did. Finally, the testing stage consisted mostly of data science related testing (benchmarking, accuracy, etc.), while the traditional software testing was ignored due to time constraints of the project.

9. Technological Rule

In this section, the technological rule will be presented, which is the outcome of following the design science paradigm in software engineering. First the characteristics which make the work suitable for design science will be outlined, followed by the technological rule.

It is argued that the work falls in the domain of design science as it posses these characteristics:

- **Problem conceptualization:** Hardware imposed restrictions cause heavy compromises and make the development of the ML/DL models difficult
- **Solution design:** The problem would be solved if a method of calculating the hardware requirements for the model features and parameters before training phase existed
- **Abstraction:** The solution is applicable only if it does not compromise other aspects of the model. The solution is not dependent on the specific architecture of the deep learning model and does not impose any restrictions on the selection of parameters of the model.
- **Instantiation:** The autoencoder deep learning model has been developed for the purposes of fault detection in an industrial context. The autoencoder model posses the `get_model_memory_usage` method which allows for the model’s memory to be calculated before the training phase.
• **Empirical validation:** There are two notable periods of the model development, the phase before the method was implemented and the period after the method had been implemented. It was observed that rapid prototyping and better understanding of the model itself was enabled by the implementation of the method.

Thus, since the work satisfies the requirements by the design science paradigm, a technological rule is proposed as follows:

To achieve faster development, prototyping and better understand the hardware compromises which are necessary to make in the context of developing deep learning models with large datasets include a method or function which will calculate the feature and parameter memory requirements of the training stage.
10. Discussion

This section aims to discuss the key takeaways from the implementation and comparison of the implementation to the state of the art and how they contribute towards answering the first two research questions. Additionally, the discussion is provided on how the technological rule presented in section 9 fits into the context of the final research question in solving some the challenges identified in the related work.

The implementation of software engineering practices in developing the autoencoder model for fault detection has proved to be a successful case of validating current state-of-the-art challenges and practices in research of software engineering of deep learning models. As mentioned before, no industrial case validations could be found in the existing literature, which serve as an important benchmark for the generalizations that are made. Most of the authors of related work have noted the difficulties in eliciting requirements, citing that most of them are data-driven and centered around the stakeholders understanding of the data. Far too often in such kind of projects, the traditional software requirements are neglected for the sake of achieving maximum accuracy of the model, and they are understood well deep into the projects development lifecycle. This was the case for this project, as problems in the development arose from loosely defined software requirements, which were resolved way later than they should have been. Despite the findings of academia regarding the highly coupled components in the architecture of most deep learning implementations, careful planning of the architecture and the pipeline has allow for this project to be characterized by loosely coupled components. Lastly, a high overlap is noted in the development stage of the project between the literature and the implementation, most notably in the platform and package selection, and the overall omission of standard software testing. It can be observed, however that one of the limiting factors of this project is the incomplete software engineering lifecycle, due to the missing deployment and monitoring stages. Regardless, when taking into consideration the time-span in which the project was carried out, the end result is deemed satisfactory and can provide a guideline on raping deep learning model prototyping.

By far the most limiting factor in rapid prototyping and development of the deep learning model proved to be the shrouded hardware restrictions. The hardware restrictions arose in the development of the training pipeline of the deep learning model, and required major redesign if they weren’t precisely identified and mitigated. These problems were identified both in the previous works of academia and this implementation, however, they have not been addressed up to this point, or at least, no mention of them was found in the existing literature. Thus, the proposition of the technological rule from section 9 provides a contribution to future development of deep learning models and ensures a faster, more precise way to identify and mitigate hardware-induced development issues.

Formally, these are the answers to the research questions proposed in the work:

- **RQ1:** What are the existing software engineering challenges in developing deep learning models?
  
  As it was inferred from section 3, the key challenges of software engineering of deep learning models lies in difficult requirements elicitation due to dependency on stakeholders’ understanding of data, high coupling of components in the architecture, and obscured hardware limitations hindering rapid development.

- **RQ2:** Which characteristics from contemporary research on software engineering of deep learning can be observed in a real-world industrial scenario?
  
  Observed characteristics include difficult requirements elicitation, fixed and predefined architecture, use of open-source modules and libraries, and hardware-induced development issues. Despite several mentions of high-coupling of components, the project posses a loosely coupled architecture.

- **RQ3:** How can some of the software engineering challenges in developing of deep learning models be mitigated?
  
  One of the addressed challenges of the contemporary work were the hardware-induced development issues in developing deep learning models. The solution to the concrete problem of
rapid autoencoder prototyping and development was the implementation of a method that calculated the memory requirements of training the model in \( n \) amount of batches. Thus a technological rule is proposed: "To achieve faster development, prototyping and better understand the hardware compromises which are necessary to make in the context of developing deep learning models with large datasets include a method or function which will calculate the feature and parameter memory requirements of the training stage". Practitioners of deep learning will observe faster prototyping and clearer limitations imposed on the project with the application of this rule. The rule is to be implemented as an extension of the summary method of deep learning models. Thus, it is a contribution to future software engineering project involving deep learning models.
11. Conclusions and future work

The section contains all the conclusions that were drawn from the implementation, comparison to the state of the art, and the proposed technological rule for the software engineering of deep learning models. Lastly, future work is discussed and how the work can be improved and built upon.

The implementation of software engineering practices in developing an autoencoder model for fault detection has been successfully validated against current state-of-the-art challenges and practices in the research of software engineering for deep learning models. The absence of industrial case validations in existing literature is highlighted. It’s noted that many related works face challenges in requirements elicitation due to data-centric nature and stakeholder data understanding, often sidetracking traditional software requirements. While academic findings often point to tightly coupled components in deep learning architectures, this project managed to achieve loosely coupled components through careful planning. There’s substantial overlap between literature and implementation in stages like platform and package selection, with a notable omission of standard software testing. A significant obstacle in rapid prototyping is hardware limitations, identified both in academia and this implementation. This issue hasn’t been adequately addressed in existing literature, making the proposed technological rule, which involves calculating memory requirements for training deep learning models, a valuable contribution for addressing hardware-induced development issues in future projects. The project’s completion is considered satisfactory for providing guidance in deep learning model prototyping.

Regarding the future work, the first possible improvement on this work is for the project to be carried out in all stages of software engineering lifecycle, as this project stopped just before the deployment and monitoring stages. These could identify new challenges and produce new technological rules, further benefiting the software practices in engineering deep learning models. A slight change of context could also further this research as inclusion of other types of deep learning models would serve as a validation, especially for the proposed technological rule. Given the noted omission of standard software testing in the project, future work could delve into incorporating robust testing methodologies for deep learning models. This might involve exploring techniques such as unit testing, integration testing, and automated testing tailored to deep learning workflows.
References


